

Conceptual Models

and

Computing

Mícheál Mac an Airchinnigh

This thesis is submitted for the Ph.D. in Computer Science at the University of Dublin, Trinity College, Department of Computer Science, wherein the research was conducted.

October 1990

# Declarations

I declare that this thesis has not been submitted as an exercise for a degree at any other University.

I declare that all of the material contained herein is entirely my own work.

I declare my consent to the Library of Trinity College, Dublin, that I agree that the Library may lend or copy this thesis upon request.

Signature: M. Mac an Airchinnigh

Date: 3 October 1990

## Summary

Human beings form conceptual models of their world and behave accordingly. I have focused my research on those aspects of conceptual models which are peculiar to the act of computing and with particular reference to the specification, design and development of software which is intended to be executed on a computer.

The conceptual models of mathematicians and computing scientists have much in common. I argue that software engineers should develop a similar *kind* of conceptual model, one which is founded on discrete mathematics and mathematical method. I have identified the central rôle that abstract notation, which is a “tool of thought”, ought to play in the formation of the software engineer’s conceptual model of computing.

That I might “stand on the shoulders of giants” and add to the considerable body of learning in the field of computing, I chose to develop my theory of conceptual models using the the Vienna Development Method (*VDM*) for software engineering and its accompanying notation or metalanguage, *Meta-IV*. The dialect of *Meta-IV*, which seemed to me to be the most promising as a basis for my research, was that of Dines Bjørner and the Danish School of the *VDM*.

Having developed an operator calculus for the *VDM* and extended the method to conform with traditional mathematical practice in problem solving, I have laid a new foundation for the study of constructive discrete mathematics as viewed from the perspective of software engineering. In particular, the conceptual model that I propose, opens a new chapter on computing pedagogy.

# Dedication

To the three women in my life:

my wife, *Anne-Isabelle*,

and my daughters, *Eloïse* and *Thérèse*,

who supported me with their love.

## Acknowledgements

I owe a debt of gratitude to my supervisor, Professor John G. Byrne, who supported and encouraged me in my research work. His open mind and critical attitude to my ideas allied with his searching questions on important points forced me to be precise where I would have preferred to be vague.

In matters of philosophy and critical thinking, I have relied on the technical expertise of my wife, Anne-Isabelle, whose Cartesian mindset and French cultural background have proved to be an inspiration.

Without question, I must acknowledge Professor Dines Bjørner, as my mentor in the Vienna Development Method (*VDM*). His enthusiasm for the field conveyed by his numerous publications, his lectures and conversations, as well as those of the Danish School of the *VDM*, have had a strong influence on the overall direction. Although I much preferred Bjørner's style to that of Professor Cliff B. Jones and the English School, I also owe him a debt of gratitude, both with respect to his technical writings and his particular conceptual viewpoint of software engineering. Only lately had I met Professor Andrzej Blikle, whose *VDM* work in the Polish School I admire, and which had considerable influence in the thrust of my arguments.

Research is not conducted in isolation. I thank my former colleague, Hans-Jürgen Kugler, for innumerable discussions on all aspects of software technology over a period of ten years. Without his companionship and depth of learning in the field, I would sorely have been at a loss.

My final word of thanks is to a departed colleague, Dr. Robert Friel, *Requiescat in pace*, who introduced me to T<sub>E</sub>X and who is, therefore, directly responsible for the quality of the technical writing in the thesis.

# Table of Contents

Chapter 1: Conceptual Models	1
1. Introduction	1
1.1. Language and Notation	3
1.2. Structure	8
1.3. Definitions, Hypotheses, and the like	10
2. The <i>MRC</i> Notation	11
2.1. The Origins of the <i>MRC</i> Notation	13
2.2. The Meaning behind the Notation	16
3. Models	18
3.1. The Essence Thereof	19
3.2. Ockham's Razor	21
3.3. Conceptual Models	23
3.4. Mental Models	25
4. Representations	26
4.1. Algorithms	30
5. Encodings	33
5.1. Function and Algorithm	33
5.2. The Concept of Type	35
5.3. On the Inadequacy of Encodings	37
6. Summary	37
Chapter 2: An Operator Calculus for the <i>VDM</i>	39
1. Introduction	39
1.1. The <i>VDM</i> Schools	41
1.2. Currying/Schönfinkeling	43
2. The Domain of Sets	48
2.1 Set Operations	50
2.2 Conceptual Expressiveness	59

Chapter 2: An Operator Calculus for the <i>VDM</i> continued.	
3. The Domain of Sequences	60
3.1 Sequence Operations	62
3.2 Conceptual Expressiveness	70
4. The Domain of Maps	71
4.1. Map Operations	73
4.2. Conceptual Expressiveness	81
5. The Domain of Trees	82
5.1. Projection	84
5.2. Recursive Trees	85
5.3. Operations	86
6. Summary	89
Chapter 3: Monoids and Homomorphisms	91
1. Introduction	91
2. Monoids	94
2.1. The Domain of Sets	96
2.2. The Domain of Sequences	96
2.3. The Domain of Maps	99
2.4. Other Algebraic Structures	100
3. Monoid Homomorphisms	102
3.1. The Domain of Sequences	106
3.2. The Domain of Sets	117
3.3. The Domain of Maps	121
3.4. Monoids of Endomorphisms	125
3.5. Admissible Submonoids	128
4. Summary	130

Chapter 4: Tail-Recursive Algorithms	131
1. Introduction	131
1.1. Conceptual Errors	133
1.2. A Selection of ‘Error-correcting’ Algorithms	134
1.3. The Generic Solution	140
2. Basic Numerical Algorithms	142
2.1. The Exponential Function	142
2.2. An Alternative Version of Exp	149
2.3. Factorial Function	151
2.4. Tail-recursion and While loops	154
2.5. Combinatorial Functions	155
2.6. Multiplication	156
3. The Free Monoid Homomorphisms	158
3.1. Properties of the Tail-recursive form	161
3.2. PROLOG Implementation	162
4. Reduction	164
4.1. Other Tail-recursive Versions of Reduction	167
4.2. PROLOG Implementation	168
4.3. Imperative Encodings	169
4.4. Extension of the Reduction Operator	171
4.5. Application to Polynomial Evaluation	172
4.6. PROLOG Implementation	172
4.7. The Restricted Reduction Operator	173
4.8. Imperative Encoding	174
5. Application	175
5.1. PROLOG Implementation	176
5.2. Tail-recursive Form	176
5.3. Imperative Encoding	177
6. Summary	178

Chapter 5: The Method of the Irish School	181
1. Introduction	181
1.1. Conjecture, Proof and Counter-example	182
2. Invariants	184
2.1. Subinvariant 1	187
2.2. Subinvariant 2	188
2.3. Subinvariant 3	193
3. The Modified File System Model	195
3.1. The Put Command	197
3.2. Theorem and Proof	202
4. The Domain of Bags	208
4.1. Operations on Bags	210
4.2. Applications	214
5. The Domain of Relations	224
5.1. Transfer Operations	226
6. Summary	230
Chapter 6: Discovering Specifications	231
1. Introduction	231
2. Requirements Model	233
3. A Requirement	237
3.1. The Deus ex machina Requirements Test	240
3.2. The Communications Requirements Test	242
3.3. Constraints, Context Conditions, and Invariants	244
4. Bill of Materials	246
4.1. The Operations	247
4.2. Bill of Materials as Product	253
4.3. Addition Chains	255
5. Material Requirements Planning	262

## Chapter 6: Discovering Specifications continued.

6. Scheduling	268
6.1. Johnson's Scheduling Algorithm	271
6.2. An Alternative Specification	274
6.3. The GANTT Diagram	275
6.4. Johnson's Algorithm for 3 Machines	277
6.5. A Heuristic Algorithm for n Machines	278
7. Summary	279

## Chapter 7: Graphics

1. Introduction	281
2. Line Drawing Algorithms	282
2.1. Simple DDA Algorithm	283
2.2. The Bresenham Algorithm	285
2.3. Implementations in PROLOG	286
3. Polygon Clipping	288
3.1. Sutherland-Hodgman Algorithm	289
3.2. A Refinement	292
3.3. A Refinement — Skeletal Imperative Encoding	292
4. Polygon Fill Algorithm	294
4.1. The Build Phase	295
4.2. The Process Phase	300
5. Curves	302
5.1. Recursive Definition of Bézier Curves	305
5.2. de Casteljau Algorithm	306
5.3. Construction of the Polyline	310
5.4. The Subdivision Algorithm	313
5.5. Derivative of the Bézier Curve	315
6. Surfaces	319
6.1. Construction of the Wire-frame	321
7. Summary	326

Chapter 8: Communications & Behaviour	329
1. Introduction	329
1.1. Communicating Sequential Processes	331
2. Communicating Agents	335
2.1. Entities and Structure	336
2.2. Agents and Entities	337
2.3. The User as Agent	339
2.4. Context Sensitive Communication	341
2.5. The User's Context	344
2.6. History	347
3. Communicating Results	352
4. Directions for Future Research	354
Appendix A: The Dictionary	357
1. Introduction	357
2. Model 0 — Set	358
2.1. The Operations	358
2.2. Development	360
3. Model 1 — Set Partition	362
3.1. The Invariant	363
3.2. The Retrieve Function	366
3.3. The Operations	366
4. Formal Relationship of Model 1 to Model 0	369
4.1. The Retrieve Function	370
4.2. The New Operation	371
4.3. The Enter Operation	371
4.4. The Remove Operation	372
4.5. The Lookup Operation	373
4.6. The Size Operation	374

## Appendix A: The Dictionary continued.

5. Model 2 — Sequence	375
5.1. The Invariant	375
5.2. The Retrieve Function	376
5.3. The Operations	377
5.4. Further Developments	380
6. Formal Relationship of Model 2 to Model 0	381
6.1. The Retrieve Function	381
6.2. The New Operation	382
6.3. The Enter Operation	382
6.4. The Remove Operation	383
6.5. The Lookup Operation	384
6.6. The Size Operation	385
7. Model 3 — Sequence Partition	386
7.1. The Invariant	386
7.2. The Retrieve Functions	387
7.3. The Operations	388
8. Formal Relationship of Model 3 to Model 0	389
8.1. The Retrieve Function	389
8.2. The New Operation	390
8.3. The Enter Operation	391
8.4. The Remove Operation	393
8.5. The Lookup Operation	395
8.6. The Size Operation	396
9. Model 4 — Map	398
9.1. The Operations	399
10. Model 5 — Map	403
10.1. The Operations	403

Appendix B: The Environment and Store	407
1. Introduction	407
2. Model 0: The Store	408
2.1. The Invariant	408
2.2. The Operations	408
3. Model 1: Environment and Store	410
3.1. The Invariant	410
3.2. The Operations	411
4. Summary	420
Appendix C: The File System	421
1. Introduction	421
2. File System — Version 0	423
2.1. Semantic Domains	423
2.2. The Invariant	424
2.3. Syntactic Domains	429
2.4. Semantic Functions	429
2.5. Summary	441
3. File System — Version 1	442
3.1. Semantic Domains	442
3.2. The Retrieve Function	444
3.3. Syntactic Domains	444
3.4. Semantic Functions	445
4. Conclusion	455
References	457

# Chapter 1

## Conceptual Models

### 1. Introduction

In this work, I propose the thesis that a theory of conceptual models is central to Computing. By ‘computing’ I intend to mean that branch of constructive algorithmic mathematics over finite structures, against a framework of computing machinery characterised by the notion of state. Now what I have just written may be interpreted as a sort of working definition. But caution in interpretation is called for. A word such as ‘state’ triggers different meanings depending on the context. While I may permit one to suppose that it has the connotations of ‘memory’, ‘storage’, etc., yet I am also conscious of its use in denotational semantics where it denotes ‘a world populated by specific entities’. The domain of discourse of the thesis is limited to the so-called discipline of software engineering. Specifically, I will demonstrate that formal software development is the keystone of said discipline and provides a sound foundation for same. Now I wish to liken the process of formal software development to that of the construction of a classical Greek tragedy, the characteristics of which have been set down by Aristotle in his *Poetics* (Hutton 1982, 50). Formal software development consists of a) a formal specification derived from requirements, and b) a formal method by which one proceeds from the specification to the ultimate concrete reality of the software. A formal specification may be considered to be the outline of the plot; the method is the procedure by which details of the plot are filled in.

The term ‘conceptual model’, which I distinguish carefully from that of ‘mental model’ also triggers all sorts of connotations, especially within disciplines other than computing: logic, cognitive psychology, philosophy, mathematics and science. I will argue that the basis of computing is the construction and manipulation of a conceptual model. Alas, there does not seem to be any way in which I can avoid

## CONCEPTUAL MODELS

circularity or self-reference in speaking about such a conceptual model. For, the very approach by which such construction and manipulation takes place and which is the matter of this thesis is, of course, an elaboration of my understanding—my conceptual model. By that I do not wish to imply any tendency towards *solipsism*. I do believe that the conceptual model which I have, has a conceptual basis, shared to a greater or lesser degree, by all those who are ‘involved’ with computing. The particulars are, of course, unique to me alone in respect of my background experience and beliefs. The former being grounded in the programming languages that I have employed and the authors that I have read; the latter is determined by the particular philosophy that is elaborated herein.

The ultimate goal of the thesis is to exhibit a specific philosophy of education in computing, one which is in broad agreement with the sort of principles behind quotations such as:

1. “As long as we work ourselves into such excitement over these languages [Algol 60, Cobol, Basic, ...], we are apt to forget that they are only a means of expression and that the essential thing is to have something to say, not how to say it.” (Arsac 1985, xv). I would add Ada, C++, PROLOG, etc.
2. “Given a terse and readable abstract model of some software item ... we see it as the foremost and almost solely distinguishing task of programming to carefully turn [sic.] the abstraction ... into an efficient realization.” (Bjørner and Jones 1978, 33).
3. “The design of such computer systems should employ development methods as systematic as those used in other engineering disciplines” (Bjørner and Jones 1982, ix).
4. “operational reasoning about programming is ‘a tremendous waste of mental effort’ ” (Dijkstra 1989, 1403). I can not agree more with these words of Dijkstra having spent enormous amounts of energy trying to reason about computer graphics algorithms.

By ‘education’ in the phrase ‘philosophy of education in computing’, I wish to draw on the original meaning of *educare*—‘to lead out’. That is to say, there is more involved than a simple pedagogical theory. I wish to propose a conceptual model of computing, one that is firmly grounded in and dependent on philosophy. From this philosophy, a specific method in formal software development is derived, one that

has significant pedagogical import. I claim that the science of Computing is in a pre-paradigmatic state in the sense that Kuhn ([1962] 1970) employs that term.

### 1.1. Language and Notation

Why is there such a proliferation of computer programming languages? That there is, is indisputable. On the one hand, with respect to certain languages, there is a clear relation between the concepts that they embody and the machinery upon which they execute. For example, the assignment statement clearly reflects the concrete reality of von Neumann storage update. There is, on the other hand, a sense in which the programming language itself is a conceptual model of the machinery on which programs written in that language execute (compare this with Wegner's statement that "a programming language can be thought of as a specification of a class of computers, where each computer in the class corresponds to a program of the language" (Wegner 1971, 226)). Thus one may speak of a 'C machine', a 'Pascal machine', a 'LISP machine', etc. This a notion frequently employed in computing pedagogy. That one should seek to turn a pedagogical notion into a reality—build a real 'LISP machine' to support a virtual 'LISP machine'—is a natural scientific progression that fits analogously with the paradigm of normal science (Kuhn [1962] 1970). The real language machine is the counterpart of the laboratory/experimental equipment.

Programming languages belong to families much in the same way that natural languages do and one can trace the ancestry of members within such families and between such families. Both Ada and C belong to the family of 'imperative von Neumann' languages and reflect many of the characteristics of Algol 60 which, in a sense, is the original ancestor. But C++ and Smalltalk, members of the 'object-oriented' family, have features which may be traced to Simula 67, a language which also has Algol 60 as ancestor. Language development seeks to capture non-machine oriented concepts within its syntactic forms: Ada encapsulates many of the so-called software engineering principles necessary for programming by teams; Smalltalk encapsulates the notion of inheritance.

A parallel historical development sprang from the notion of programs as mathematical entities. In particular, the view that a program was a mathematical (recursive) function led inexorably to the development of the family of 'functional'

## CONCEPTUAL MODELS

programming languages, of which, it may be said, LISP is the father. Now the conceptual model behind such languages is different in essence from those of the ‘imperative’ family, as is indeed the notation employed and their mode of use. In fact, with respect to the description of mode of use, terms such as ‘conceptual programming’ (Sandewall 1977) and ‘exploratory programming’ were employed. It is not surprising that from such mode of use the whole subdiscipline of Artificial Intelligence originated.

Out of such ‘primordial soup’, the concept of programming language independent formal specification languages evolved. Once again, one observes the same sort of diversity ranging from those that focused on structure/architecture such as the *VDM Meta-IV* (Bjørner and Jones 1978) and those which focused on behaviour such as CSP (Hoare 1985). As will be made clearer later on, the qualifier ‘programming language independent’ will need to be modified somewhat. For even these languages retain vestiges of their computing machinery origin much in the same way as highly evolved natural species such as the human being retain vestiges of their origins. Still I observe the same sort of pre-paradigmatic chaos. But I will argue that the emergence of a computing paradigm is emerging.

Now what is the nature of formal specification and how does it differ from that of program? Can one say that the act of formal specification is a mathematical activity? I will demonstrate that whereas mathematics does form an integral part of such activity, there is also an essential conceptual modelling component that underlies both the activity of formal specification and programming. The distinction between formal specification and programming is to be found in the type of notation employed as much as in the ultimate purpose of each activity. To give a brief indication of what I understand to be such a distinction I would like to mention the place of poetry in literature in contrast to prose (cf., Aristotle’s remarks on the matter in *Poetics* (Hutton 1982, 54)).

One of the essential characteristics of poetry is its terseness of language. It is not necessary to qualify poetry with the attribute ‘good’; but one may speak of good and bad verse. The poem speaks of a truth that transcends time and space. But in the case where the language and culture of the reader is totally different from that of the poet, much effort is required to recapture a glimpse of that truth. Moreover, said truth, though originally emanating from an individual, may be said to take on a

life of its own through the word. It is not necessary for the reader to reconstruct the original world picture in detail to grasp that truth. Indeed, it is often the case that subsequent readers reinterpret the words in their own world picture that presents a fresh enriched truth. A formal specification is the poetry of computing.

### 1.1.1. Notation in Mathematics

Mathematical notation is the ultimate language of poetry in that it is ideally space and time invariant. Naturally, it has also evolved over time. Some symbols have endured. Others have perished (see Cajori's work cited in (Davis and Hersch 1980) and in (Iverson 1980, 444)). Even today there are still variants in widespread use. For example, consider the 'trivial' matter of the decimal point; two commonly used variants are 3.14159 and 3,14159. Mathematics without symbols would be inconceivable. Of course, it was not always so. In early times the language of mathematics was practically symbol free and precisely for that reason was very limited.

That notation may signify more than was originally intended may be illustrated by the choice of notation for the differential of a variable  $x$  with respect to time  $t$ . Newton used  $\dot{x}$ ; Leibnitz used  $dx$  (Boyer 1968, 435; 441). It was the latter that led to the operational calculus (Davis and Hersch 1980, 125). See remarks by Whitehead of a similar nature (Whitehead [1911] 1948, 168). Moreover, although the differential is to be considered as an atomic entity, the suggestiveness of separability in the notation  $dx/dt$  may be exploited to introduce the whole field of differential equations. Thus, from  $dx/dt = f(t)$ , one derives  $dx = f(t) dt$ , and, thence the integral,  $x = \int f(t) dt$ . Note also how the introduction of the symbolic wedded Geometry and Algebra. Although the Greeks used Geometry to solve algebraic problems and made extensive use of coördinates in the treatment of conics, it was the algebraic notation of Viète and Descartes which provided the expressive power; see (Boyer 1968, 333–42; 368–80) on Viète and Descartes, respectively, and (Dieudonné 1972).

That notation may be employed effectively to solve problems, without having a rigorous foundation at the time, has been attested to over and over again in the history of mathematics. Consider once more the use of infinitesimals. Berkeley attacked the whole position rather successfully (Davis and Hersch 1980, 244): "If it be said that several theorems undoubtedly true are discovered by methods in

## CONCEPTUAL MODELS

which Infinitesimals are made use of ... I answer that ... [it is not] necessary to make use of or conceive infinitesimal parts of lines ...” (Berkeley [1710] 1962, 133). It was only in the second half of this century that a rigorous justification for their use was given by Abraham Robinson, albeit in the context of nonstandard analysis (Enderton 1972, 164 *et seq.*; Davis and Hersch 1980). The summation of infinite series is another case in point. Mathematicians such as Euler employed infinite series to great effect and sometimes obtained strange results (Boyer 1980, 486). It was not until the time of Cauchy (*ibid.*, 566), and still later Weierstrass (*ibid.*, 606), that the theory of convergence of infinite series established their validity. A third is the theory of quaternions (Altmann 1986).

That notation is not to be regarded as an end in itself has been attested to. Consider Ulam’s comment in his autobiography:

“I am turned off when I see only formulas and symbols, and little text. It is too laborious for me to look at such pages not knowing what to concentrate on” (Ulam 1976, 275).

signifying that the symbolic ought to be a succinct illuminating adjunct to the descriptive text and vice-versa. The example of the style of  $\mathcal{Z}$  specifications is a case in point. However, it would be ridiculous were the notation to be somehow redundant with respect to the text. One complements the other. I am entirely in agreement with Whitehead (Whitehead [1911] 1948, 41), where he asserts that “It is a profoundly erroneous truism, repeated by all copybooks and by eminent people when they are making speeches, that we should cultivate the habit of thinking of what we are doing”. It is by acquiring skills that we are freed from thinking. The notation is the tool by which this is accomplished in mathematics.

Now, it seems to me, that I ought to mention here also the significance of the rôle that the picture or diagram has played in mathematics. As for Geometry, at least ‘classical’ Geometry, the diagram was almost an essential adjunct in mathematical problem solving activity. It conveyed at a glance the nature of the problem in question and frequently suggested the solution. It is only rather recently, since *circa* the late nineteenth century, that its rôle diminished and more emphasis was placed on the purely symbolic.

### 1.1.2. Notation in Computing

The era of abstract symbolic notation in computing was heralded by the introduction of the programming language APL, a lucid presentation of which is given in Iverson's ACM Turing Award Lecture of 1979, *Notation as a Tool of Thought* (Iverson 1980). Although used originally as a language for communication between the designers and programmers of a complex system in 1957, i.e., as a specification language, (Falkoff and Iverson, 1981, 662), the provision of an appropriate keyboard and IBM golf ball printer led inexorably to the use of the notation as a programming language. Iverson, in his Turing Lecture merely states that it originated in an attempt to provide clear expression in writing and teaching (Iverson 1980). APL succeeded precisely because the tool-builders abandoned the conventional wisdom of the use of ASCII-like character codes. Programs in APL consist of 'pure' notational forms. Consequently, such programs were considered unintelligible by other non-APL programmers, their complaint being much the same as that of Ulam cited above. But such programs, like their formal specification counterparts, are not intended to be read, a common misunderstanding by 'outsiders'. They are intended to be redone! A rationale for the strange evaluation of APL expressions from right to left was given by Iverson in his work *Elementary Functions* (Iverson 1966). He sought to overturn centuries of mathematical tradition in the use of operators, for a rational approach that facilitated APL tools and style of use.

However, at the symbolic level, all programming languages are essentially nothing more than notation. For this reason, the argument that programs must be well-documented is given. Such advice is nothing more than a restatement of the essential complementarity of notation and descriptive text. However, where it is asserted that the program text is to be self-documenting, an argument often made for the use of Ada, for example, then one must begin to suspect the wisdom of the advocates. Verbosity in programming is inherently self-defeating.

That advocates of formal specification languages also fall into the same trap would seem to be due to the desire to build ASCII-oriented computer tools. Loathe to repeat the imaginative solution of the APL promoters, a minimum cost attitude, or principle of least effort, would appear to prevail. For instance, the verbosity and entire unwieldy machinery of the RAISE specification language (Prehn 1987), a proposed successor to the *VDM Meta-IV* (Bjørner and Jones 1982), is likely to do

## CONCEPTUAL MODELS

more harm than good for the development of the science of formal specification. The use of an ASCII syntax for the BSI/VDM standard, even though a mathematical syntax is also provided, is very likely to lead to the proliferation of tools for same and widespread adoption of a style of verbose specifications that will obscure rather than clarify the essentials.

I will demonstrate that it is precisely the very nature of the abstract notation used in formal specification that leads to expressive power and suggestiveness, that said nature resembles that of mathematical notation, but differs from it to the extent that formal specification is as much about the conceptual modelling of behaviour as it is about structure.

### 1.2. Structure

“A structure contains [...] certain unifying elements and connections, but these elements cannot be singled out or defined independently of the connections involved [...] structures defined in this way may be considered independently of the elements that go to make them up [...] there exist some structures of various logical ‘types’; that is to say, one must be prepared to envisage structures of structures [...]” (Piaget 1971, 139).

One must distinguish between essential or inherent structure and the notation or language for expressing structure. Conceptual structure is that by which one manages complexity. Consider the human body and its underlying skeleton. The skeleton is the supporting structure or framework by which the human body exists. Without the skeleton there can not be a human body.

#### 1.2.1. Structure in Mathematics

The *discovery* of structures, and morphisms between structures has been one of the great unifying forces in the history of mathematics. I have chosen the word ‘discovery’ to signal that in a certain sense mathematicians only reveal what already exists. Much the same attitude of ‘discovering’ must be adopted by computing professionals. Napier discovered (natural) logarithms—the logarithm being an isomorphism between certain groups of real numbers, one with a law of multiplication, the other with a law of addition. Though, as is common in mathematical discovery, a swiss clockmaker Jobst Bürgi may have actually preempted him (Boyer 1980, 346).

Might not one argue rather that Napier *invented* logarithms? I would say no. Napier invented a mechanism—his rods—which made manifest his discovery. But it can rightly be argued that his ‘discovery’ was not the ‘discovery’ of logarithms with respect to their essential nature, as an isomorphism between structures. Indeed, as Knuth remarks, Napier did not make the connection between his logarithms and their inverses (Knuth 1973, 1:23). For that kind of discovery one had to await the development of the Theory of Groups by Évariste Galois (Boyer 1980, 641).

### 1.2.2. Structure in Computing

Originally, in programming, structure was taken to be ‘data structure’. Operations on the structure were syntactically independent of the syntax of the structure itself. By that I mean the syntax for ‘describing’ structure, say types, was distinct from that of operations on the structures, as given by say, functions or procedures. There was, of course, a certain syntactic correspondence between basic structures, such as the array, and appropriate operations, such as selection of an element of the array. Abstract data types introduced the concept of encapsulated structure, binding data structure and operations together within one syntactic form hereinafter called a module, and brought the computing notion into line with that understood in mathematics. But, it is to be well-noted that use of modules does not guarantee anything about (intrinsic) structure. For example, in Ada the syntactic form package corresponds to what has been described as a module and one frequently finds the advice that it be used to group things together—even where such things have no relation one to the other. Designers of new programming languages are always loathe to accept the names of syntactic entities that are already well established.

That there should be an immediate correspondence between the notion of structure as understood in mathematics and that in formal specifications seems to me to be obvious. But, it appears that the sort of structures that one finds in computing are incredibly more complex than those of mathematics. Now I would like to flag the important distinction that must be made between structure and module within the context of specifications. Introduction of a syntactic encapsulating form does not solve the problem of discovering structure; where such structure has already been identified, the module may, of course, be employed to the benefit of tools.

## CONCEPTUAL MODELS

### 1.3. Definitions, Hypotheses, and the like

“For this reason I have stressed that I am not interested in definitions; since all definitions must use undefined terms, it does not, as a rule, matter whether we use a term as a primitive or as a defined term” (Popper 1972).

The main purpose in the use of definitions is to give a definitive structure (and often a name) to a collection of undefined terms. Definitions function similarly to axioms in this respect. There are several possible interpretations that one might give to the term hypothesis. One which I find on occasion to be particularly useful in philosophical work on conceptual models is that which interprets hypothesis as an educated guess, which must be subsequently verified by experimentation and/or proof. In the more formal part of the thesis which deals with the verification of specifications, I use hypothesis in the sense of Pólya to denote an essential part of a mathematical theorem.

In the remainder of this Chapter I present a very brief overview of my early work on a User’s Conceptual Model which I began in the Summer of 1983. At that time my primary objective was to use some form of symbolic notation to try to express the relationships between concepts and words in the domain of Computing. The study of programming languages played a major rôle in that early research and I argued that were one to consider a programmer as a user, then the programmer actually acquired a particular conceptual model of computing as a direct result of programming in a particular language. Having graduated in pure mathematics and having subsequently transferred to computer science, the relationship between the activity of solving problems in mathematics and that of solving problems by programming intrigued me. I wondered how such a relationship might be clearly exhibited. The *MRC* notation, which I now present, was developed precisely for that purpose.

## 2. The *MRC* Notation

“Further development [of any science], therefore, ordinarily calls for the construction of elaborate equipment, the development of an esoteric vocabulary and skills, and a refinement of concepts that increasingly lessens their resemblance to their usual common-sense prototypes”. (Kuhn [1962] 1970, 64)

Kuhn’s observations on Science, as understood in the traditional sense, seem to me to be equally applicable to Mathematics and Computing. The esoterism of mathematical vocabulary and skills is apparent to everyone; to those acquainted with mathematics, the explosive growth of such vocabulary and skills in this century has been startling, even to the extent of locking out practising mathematicians in one field from other fields in mathematics (Davis and Hersch 1980, 194). The same phenomenon is also observed in Computing.

Once, Mathematics was accessible to the ‘layman’, especially in the field of Euclidean Geometry. Computing is still accessible to the layman, specifically in the domain of programming. Indeed, one of the peculiarities of Computing lies in just this, the bridging between the professional and the layman. Whereas one may adopt the narrow view of a Theory of Computing that is independent of computing machinery—so-called theoretical Computer Science which shares many of the same characteristics as Mathematics—it is precisely in the behavioural aspects of said machinery that distinguishes the field from Mathematics. For then, not only may the layman dabble in programming, but may also experience the phenomenon of computing as ‘user’.

One of the most significant developments in the field of Computing is the concern about meaning, or semantics, and correctness. In its origins, Computing was primarily concerned with the numerical solution of problems for which mathematicians could not provide closed-form solutions and in the simulation of systems and phenomena which required computations beyond the capability of the human being (Ulam 1976, 227). As the range of problems that could be solved by Computing widened, a pressing need was felt for the development of appropriate programming languages that adequately addressed the problem domain rather than focusing on the machinery. Now concern about meaning and correctness focused on the new

## CONCEPTUAL MODELS

languages themselves and on their translators or compilers. Operational semantics sought to give meaning to programs against the framework of abstract or virtual machines, the work of Landin (in his development of the SECD machine) being the classical example (Wegner 1971, 216; Brady 1977, 222). The birth of conceptual modelling in Computing was taking place. Here I must distinguish carefully between a model such as the SECD machine and the Turing machine, both of which are computing models or abstract machines. It is that the former was specifically employed in the context of the development of programming languages—to give meaning to Algol 60—that it deserves to be called a conceptual model.

The term ‘conceptual model’ is justified by virtue of the fact that focus was switching to the concepts embodied in the programming languages which had an impact both on language developers and the programmers. A parallel development in the *design* of software was also taking place. Flowcharts were used to express algorithms. With the development of Algol 60, algorithms expressed in that language became the norm, at least among the members of the professional computing community.

That Algol 60, in spite of its simplicity and elegance, was conceptually inadequate (for indeed, programs in that language could do anything that machine coded programs could do), soon became apparent and led to the explosive growth of other languages which sought to remedy the deficiencies. Conceptual modelling developed apace. But there was also a corresponding explosion in complexity, a complexity which manifested itself in the efforts to build compilers. For this thesis, it is important to note the historical transition from operational semantics to denotational semantics in the context of the IBM Vienna Laboratory effort to build a compiler for PL/1: “The attempts to use VDL definitions [operational semantics] as a basis for systematic development of implementations [such as the PL/1 compiler] was . . . providing indications that a change of definition style might be worthwhile” (Bjørner and Jones 1978). That change became *VDM*—the Vienna Development Method—founded on denotational semantics.

But now there is a noticeable widening of the conceptual modelling activity. The very same techniques employed to provide semantics for programming languages, could equally well be employed to address other ‘computing languages’, to wit the so-called command languages of operating systems. The notion of an end-user’s

conceptual model was being delineated. Such an end-user group consisted of programmers who needed to manipulate tools such as editors, compilers, linkers, etc., in order to do their job, and system operators whose responsibility was to provide the requisite services. There was another class of end-users emerging, as a result of the technological breakthroughs in computer graphics and Computer-aided Design (which cynics would say still properly ought to be called computer-aided drawing).

Although attempts were made to capture such graphical interaction by programming language oriented techniques, it became quite evident that such end-user conceptual models could not easily or readily be expressed by same. It was in order to describe the conceptual modelling issues of such interactive users that the *MRC* notation was first employed.

### 2.1. The Origins of the *MRC* Notation

The *MRC* notation was developed at the IFIP WG5.2/EUROGRAPHICS Workshop on User Interface Management Systems, Seeheim, FR Germany, November 1983 (Pfaff 1985) to facilitate analysis and presentation of a user's conceptual model, denoted *UCM*. The letters  $\mathcal{M}$ ,  $\mathcal{R}$ ,  $\mathcal{C}$ , represented 'model', 'representation', and 'encoding', respectively. Its origins are to be found in a paper I presented at that workshop (Mac an Airchinnigh 1985b) and a summary of which is recorded in a workshop report (Mac an Airchinnigh 1985a). The most recent published account of my use of the notation is to be found in (Mac an Airchinnigh 1986, 109). Coincidentally, a similar notation was employed by Norman in his analysis of mental models (Norman 1983). Although the basic *MRC* notation is very simple, it does help one to express one's ideas more clearly. For example, let  $u$  denote a user and  $c$  denote a computer system. Then,  $\mathcal{M}(u, c, t)$  denotes the *UCM* that  $u$  has of  $c$  at some time  $t$ .

A Conceptual model  $\mathcal{M}$  as such is not directly perceivable. One can only infer its existence and nature by observing both the behaviour of and the expressions used by the one possessing it. For reasons of tractability I considered that focus on forms of expression might lead to an understanding of conceptual models that could in turn be expressed in some formal manner. Consideration of behaviour and how that might be represented, I considered to be a more complex problem, one that I have now resolved to my satisfaction and an account of which is sketched in the Chapter

## CONCEPTUAL MODELS

on *Communications and Behaviour*. Suppose that, for some formalism  $\mathcal{F}$ , one forms the representation  $\mathcal{R}(\mathcal{M}(u, c, t), \mathcal{F})$  which is interpreted as the expression by  $u$  of his understanding (conceptual model) of  $c$  at time  $t$ . Now consider the expression

$$\mathcal{M}'(u, \mathcal{R}(\mathcal{M}(u, c, t), \mathcal{F}), t')$$

This may be interpreted as an introspective analysis  $\mathcal{M}'$  by  $u$  of the *UCM*  $\mathcal{M}$  that  $u$  has of the computer system  $c$ , where  $t < t'$ . In practice, I viewed the formation and development of  $\mathcal{M}(u, c, t)$  to arise as a result of applying some inverse mapping to  $\mathcal{R}(\mathcal{M}(u, c, t), \mathcal{F})$ ; that this inverse mapping should involve, among other things, perception—both of the real entity in question experienced personally, and other expressions of that or similar entities. In other words, broadly speaking, the *UCM* is influenced by user manuals, training and use, previous experience, peer comment, and by available literature, etc.

Representations are fundamental to the theory of conceptual models. In particular, the set of representation languages may be said to be the universe of discourse of the conceptual model. Here I intend that the notion ‘language’ encompasses more than just some form of written or spoken language. Of all possible representations, those which can be executed by machine are distinguished. Such representations are called encodings in this work. The original idea behind this approach sprang from a desire to find a way in which a user’s conceptual model might be ‘stored and executed’ on a machine, an alternative view one might say to the same sort of goal set by those working in Artificial Intelligence, though a more modest one I believed. Now consider

$$\mathcal{C}(\mathcal{R}(\mathcal{M}(u, c, t), \mathcal{F}), \mathcal{P})$$

where  $\mathcal{P}$  is some programming language. One may interpret this as the computer system’s model of the *UCM*  $\mathcal{M}(u, c, t)$ . Were it possible to find  $\mathcal{F}, \mathcal{R}, \mathcal{P}, \mathcal{C}$  such that  $\mathcal{M}(u, c, t)$  was an invariant under  $\mathcal{R}$  and  $\mathcal{C}$ , then one would have an intelligent computer system in human terms. But, of course, a computer program itself could be interpreted as a representation of a programmer’s conceptual model of some small domain; and again, executable formal specifications might be regarded in a similar light. But was there not a difference in nature between these forms of representation as encodings and the ‘real’ representation intended? Could one not infer the distinction from admonitions such as ‘programs should be documented’?

It seemed obvious that an approximation relation was needed.

Denoting ‘approximates’ by  $\sqsubseteq$ , I hypothesised that inevitably an encoding could not be *more* than an approximation of a representation which in turn was not more than an approximation of the conceptual model  $\mathcal{M} \sqsupseteq \mathcal{R} \sqsupseteq \mathcal{C}$ . This seemed to me to be self-evident. To test the theory I used the concept of real-number. The set of real numbers  $\mathbf{R}$  is composed of two disjoint subsets—the rationals  $\mathbf{Q}$  and the irrationals. For instance,  $\sqrt{2}$  is irrational, a real number that caused Pythagoreans much dismay (Boyer 1980, 79). Rationals (at least of a certain magnitude) may be represented exactly on a computer. Polynomials with rational coefficients may also be represented exactly on a computer, again with the proviso that the coefficients not be too big. For instance, the polynomial  $x^2 - 2 = -2 \cdot x^0 + 0 \cdot x^1 + 1 \cdot x^2$  may be represented by the sequence of its coefficients  $\langle -2, 0, 1 \rangle$ . Consider the set of numbers which consists of the roots of such polynomials. For example, for the given polynomial, its roots are  $\{-\sqrt{2}, \sqrt{2}\}$ . Thus the irrational  $\sqrt{2}$  may be said to be exactly representable. It is *not* representable as a floating point number. Numbers which are the roots of such polynomials with rational coefficients are called algebraic numbers which include a large subset of both irrationals and complex numbers. By Cantor’s diagonal argument it may be shown that there are irrationals which are not algebraic (Birkhoff and Mac Lane 1965, 381). Such numbers are transcendental. Both  $\pi$  and  $e$  are transcendental. Now if we agree to identify a real number with an algorithm for computing it, whether the algorithm be terminating or not, then all real numbers are representable. That such algorithms exist follows from the definition of real number by the Dedekind cut (Boyer 1980, 608). Thus, with respect to the concept of real number, it could be said that strict equality exists between conceptual model, representation, and encoding.

Now for many programmers, given the representation and encoding for reals as suggested, it would be clear that their conceptual model was inadequate. For some, ‘real’ would be synonymous with ‘floating point’. But even here, it is doubtful that they would be fully aware of the realities behind floating point numbers such as those expressed in (Knuth 1981, 213). It is also highly unlikely that the representation of reals as algorithms would be encoded as such on a computer. An individual real such as  $\pi$  might be. I reached the conclusion that the domain of representations  $\mathcal{R}$  was the most fruitful starting point in studying conceptual models.

# CONCEPTUAL MODELS

## 2.2. The Meaning behind the Notation

As explained earlier, the  $\mathcal{MRC}$  notation was adopted purely as a matter of convenience in discussing conceptual models. Originally, no thought was given to the matter of semantics of the notation other than that supplied by descriptive text. However, the mere fact of adopting the notation led to the possibility that the notation might take on a ‘a life of its own’, so to speak. Further, there was the matter of the form of the notation, one largely dictated by the text processing facilities available to me at the time. My early papers used a Roman font exclusively. Experimentation with other fonts (via  $\text{\TeX}$ ) allowed me to distinguish elements selected from different domains.

Consider the expression  $\mathcal{M}(u, \varepsilon)$ . The form suggests that  $\mathcal{M}$  is a relation over the cartesian product  $USER \times ENTITY$ . That a set of users  $U = \{u_1, u_2, \dots, u_n\}$  may have the same model of some entity  $\varepsilon$  is given by  $\mathcal{M}(U, \varepsilon)$ . Choosing  $\varepsilon$  as a fixed-point of discussion, then  $\mathcal{M}_\varepsilon$  may be used to denote  $\mathcal{M}(U, \varepsilon)$ . That an individual user  $u$  has a model of some set of concepts  $E = \{\varepsilon_1, \varepsilon_2, \dots, \varepsilon_n\}$  may be expressed by  $\mathcal{M}(u, E)$  abbreviated to  $\mathcal{M}_u$ .

### 2.2.1. The domain of users

Originally, the term ‘user’, as used in the phrase ‘user interface’, was taken to denote any human being who interacted with a computer. Then it was widened to embrace the notion of rôle with respect to specific computer resources—command language interpreter, editor, compiler, debugger, etc., all of which could be said to be entities. Thus, so far, a user was one who used tools. However, upon further conceptual analysis of ‘tool’ and ‘use of a tool’, it was realized that there were nuances of ‘use’ that might further discriminate ‘user’. To signal such an important shift in emphasis, I suggested adopting the French words *usager* and *utilisateur*.

*Usager* was considered to denote one who merely used a tool, one who adapted to the tool in question. On the other hand, *utilisateur* denotes one who is able to adapt/modify the tool to one’s own needs. With respect to some computer tools, such as a compiler, it is easy to see that most users are *usagers*. But a command language interpreter, especially one such as a Unix shell, lends itself to manipulation, modification by a *utilisateur*.

In all cases, it was apparent that the notion of user was not really independent

of the domain of entities. Taking said domain as starting point and invoking the ‘use’ relation to determine ‘user’ leads to an even wider class of entities so denoted. For example, one computer might use the resources of another. Thus, could not one say that a computer was a user in this sense? In fact, it is not so much the bits of hardware that constitute a computer; rather, it is the suite of software in its entirety that determines a computer (cf., Wegner’s remarks on a ‘programming language’ computer cited earlier). A program might use the resources of another program. Is not a program a user?

Naturally, given such a domain, a re-examination of what was meant by ‘conceptual model’  $\mathcal{M}_u$  was required. Clearly, there seemed to be a conceptual model continuum from that of the human being to that of a program.

### 2.2.2. The domain of entities

In the first instance, I hypothesised that users possessed conceptual models of things, i.e., entities.

DEFINITION 1.1. *An entity is that thing which exists and which may clearly be distinguished from any other thing, not by considering any attributes that it might possess, nor in any particular relation to other things, but in its own essential being.*

Clearly, things such as computers, computer programs, etc., could be considered entities. However, in developing a theory of an Ada program methodology, it was obvious that there were user rôles which were determinable by the notions of package specification, package implementation, and package use (Mac an Airchinnigh 1984a; 1984b). Should not the things denoted by such notions be deemed to be entities? At first glance, one might be ready to accept such an argument were it the case that the terms denoted syntactic units. For there is, indeed, a compilation unit called package specification. But, in the proposed methodology, a package specification was in reality only the visible part. The private part, together with the package body, constituted the package implementation. Such thinking led to a widening of the notion of entity.

Effectively, entities were being determined by the particular user class under consideration. Moreover, they were distinguishable from abstract notions such as ‘justice’, ‘peace’, ‘love’, etc., all of which belonged to the domain of *CONCEPT*. Now it is quite clear that ontologically a user is an entity,  $USER \subset ENTITY$ ,

## CONCEPTUAL MODELS

and one may proceed to introduce the conceptual models  $\mathcal{M}(u, u)$ , and  $\mathcal{M}(u_j, u_k)$ . Now, whereas it has been demonstrated that entities may characterise elements of the domain *USER*, it is also clear that users may determine elements of the domain *ENTITY*.

### 3. Models

In the previous section the term ‘conceptual model’ has been applied rather loosely and I have already indicated that I consider a conceptual model to be different from a ‘mental’ model. I have implied that the user’s conceptual model ‘resides in the user’s mind’. What then, one might ask is the nature of such a model and in what particular way can it be said to be different from a mental model? First let me deal with the notion of model itself.

Models abound in and are fundamental to our world. They are essentially artifacts that we use to guide our decisions and behaviour. With respect to the discipline of Software Engineering, I will use *VDM* models as the basis for requirements, specification, design, and implementation. Thus one particularly distinguished class of models that are central to this work are the so-called formal mathematical models. To understand what the term model denotes, it is useful to examine how it is used. I select three particular examples.

- Consider the importance of Ptolemy’s geometrical cosmological model, which was later supplanted by that of Copernicus (Popper 1972, 173). The ptolemaic model is still adequate for some purposes, for example, in navigation. But the copernican model, which resulted in a major religious revolution, “was not more accurate than that of Ptolemy until drastically revised by Kepler more than sixty years after Copernicus’ death” (Kuhn 1977, 323).
- Euclidean geometry was deemed to be the only model of our world until the nineteenth century. According to Boyer, the discovery of non-Euclidean geometry may be considered to have “dealt a devastating blow to Kantian philosophy”. (Boyer 1968, 586). Penrose (1990, 158) makes a similar observation

— Symbolic logic is a mathematical model of deductive thought (Enderton 1972, 151). But is it not the only model. Even within the domain of logic the term model takes on a very particular meaning.

Now one might suppose that logic bears some sort of direct relation to a conceptual model. Logicians, it would appear, are guided by a ‘logical’ grammar in their thinking, said grammar being different from that of any natural language (Anderson and Belnap 1975, 473). In the domain of *formal* methods, it is generally supposed that logic *must* play a rôle. I will demonstrate just the opposite in this thesis.

### 3.1. The Essence Thereof

DEFINITION 1.2. *A model  $\mathcal{M}$  is an abstraction or simplification. Let  $\varepsilon$  denote some entity. then  $\mathcal{M}(\varepsilon)$  denotes a model of  $\varepsilon$ .*

EXAMPLE 1.1. Consider a simple von Neumann computer architecture. A model of its memory may be represented by the domain equations

$$MEMORY = ADDRESS \xrightarrow[m]{} INSTRUCTION \mid DATA$$

$$ADDRESS = \mathbf{N}$$

$$INSTRUCTION = \dots$$

$$DATA = \dots$$

I have chosen to represent (i.e., to model) the memory as the set of all possible mappings from the domain of addresses to entities which are to be modelled as elements of the disjoint domains of instructions and data. The information in this model is partial. I have chosen to ignore as irrelevant for the moment, the precise nature of *INSTRUCTION* and *DATA*. The actual memory of von Neumann computer architectures is more complex than that specified by the model. Note the inconsistency in naming with respect to *DATA*. Properly speaking I ought to use ‘*DATUM*’; but current convention dictates otherwise.

By definition, it would seem to be impossible that a model of some entity contain as much information as is implied by the entity itself. However, it is desirable to weaken the definition to include the possibility that an entity and its model are identical. For example, let us consider the set of all words,  $w$ , over some alphabet (usually denoted  $\Sigma^*$ , where  $\Sigma$  is a finite alphabet of letters). This may be adequately modelled by the domain equation  $w \in \mathbf{N}_1 \xrightarrow[m]{} \Sigma$ , subject to the constraint that the

## CONCEPTUAL MODELS

length of the word,  $w$  is  $|dom w|$  and the  $i$ th letter in the word is  $w(i)$ ,  $1 \leq i \leq |dom w|$ . In this case, both the entity and its model contain the same information. The notion of ‘as much information as’ is to be understood in the same way that a similar phrase is used by Stoy (1980):

“ $x \sqsubseteq y$  means  $x$  approximates  $y$ ;  $y$  contains more (or equal) information than  $x$ , but no information in  $y$  contradicts anything in  $x$ ”.

This approximates relation is exactly that used earlier in the  $MRC$  notation.

PROPOSITION 1.1. *For all entities  $\varepsilon$ , a model of  $\varepsilon$  is an approximation of  $\varepsilon$ , which may be written  $\mathcal{M}(\varepsilon) \sqsubseteq \varepsilon$ .*

Given a model  $\mathcal{M}$  of some entity  $\varepsilon$ , it may be possible to find a second model  $\mathcal{M}'$  of  $\varepsilon$  such that  $\mathcal{M}(\varepsilon) \sqsubseteq \mathcal{M}'(\varepsilon) \sqsubseteq \varepsilon$ , in which case  $\mathcal{M}'$  is said to be a refinement of  $\mathcal{M}$ .

EXAMPLE 1.2. A dictionary may be modelled as a set of words

$$DICT_0 = \mathcal{P}WORD$$

Certainly such a model does not contain as much information as a ‘real’ dictionary.

Now consider the model

$$DICT_1 = \mathcal{P}AGE^*$$

$$\mathcal{P}AGE = \mathcal{P}WORD$$

which introduces the notion of pagination and includes more information than the original model  $DICT_0$ . Then we have

$$DICT_0 \sqsubseteq DICT_1 \sqsubseteq \textit{dictionary}$$

Also, there may exist models  $\mathcal{M}'(\varepsilon)$  and  $\mathcal{M}''(\varepsilon)$  such that  $\mathcal{M}(\varepsilon) \sqsubseteq \mathcal{M}'(\varepsilon)$  and  $\mathcal{M}(\varepsilon) \sqsubseteq \mathcal{M}''(\varepsilon)$  and neither  $\mathcal{M}'(\varepsilon)$  nor  $\mathcal{M}''(\varepsilon)$  are approximations of the other, in which case they are said to be incomparable.

EXAMPLE 1.3. Consider the entity dictionary again. Let us model it more precisely:

$$DICT_2 = \mathcal{W}ORD \xrightarrow{m} \mathcal{P}DEF$$

$$\mathcal{D}EF = \dots$$

Then we have

$$DICT_0 \sqsubseteq DICT_2 \sqsubseteq \textit{dictionary}$$

and  $DICT_1$  and  $DICT_2$  are incomparable.

Such a hypothetical collection of models of  $\varepsilon$  forms a lattice of models of  $\varepsilon$  where  $\sqsubseteq$  is the partial order,  $\varepsilon$  is the ‘top’ ( $\top$ ) element and  $O$ , the ‘empty’ model, as the ‘bottom’ ( $\perp$ ) element. But conceptually, what exactly is this entity which I have called a dictionary? I hold the thesis that it doesn’t really matter! It is precisely the collection of models that determines the nature of the dictionary, not any particular dictionary that one might have in mind. To facilitate the formulation of the following proposition,  $\varepsilon$  itself will be called a model of  $\varepsilon$ .

PROPOSITION 1.2. *For all entities  $\varepsilon$ , the set of models of  $\varepsilon$  forms a lattice with  $\sqsubseteq$  as the partial order.*

### 3.2. Ockham’s Razor

In constructing a (conceptual) model of some entity, it is crucial that one begin with the essentials. Now it is quite possible, indeed very likely that, say in the reverse engineering of some piece of software, one arrives at a model that is already more complex than is needed. This model may in turn be considered to be *the* entity that needs to be modelled. In this fashion one retrieves the essence to be captured. Such considerations may be summed up by the following principle of ontological economy which I consider to be fundamental:

PRINCIPLE 1.1. *(Ockham’s Razor) Entities are not to be multiplied beyond necessity (Entia non sunt multiplicanda præter necessitatem).*

Occam is a variant of Ockham. The wording is taken from *A Dictionary of Philosophy* (Flew 1979, 253), and apparently the actual words are **not** to be found in the extant works of William of Ockham. Curiously, Kant’s interpretation of this principle has also a particular relevance for my approach to conceptual modelling which I espouse wholeheartedly: “that rudiments or principles must not be unnecessarily multiplied (*entia præter necessitatem non esse multiplicanda*” ([1781; 1787] 1929, 538). I have deliberately chosen a spelling of ‘Ockham’ to distinguish from that which has been appropriated by a well-known programming language. The following proposition on the application of Ockham’s Razor follows naturally:

PROPOSITION 1.3. *Where there are two models which adequately cover some entity, then by the principle of Ockham’s Razor, the simpler of the two is to be chosen.*

## CONCEPTUAL MODELS

One of the most interesting examples of the application of this principle which is exhibited in the thesis is the treatment of structures that are adequately modelled by the sequence type of the *VDM*. With respect to the specification of algorithms over such structures, one has the choice to use ‘head’, ‘tail’ and ‘concatenation’ operators, or indexing. I prefer to use the former and my preference may be justified by Ockham’s Razor. Let  $\mathcal{M}_1(\varepsilon)$  and  $\mathcal{M}_2(\varepsilon)$  be two competing models that adequately cover  $\varepsilon$ , such that  $\mathcal{M}_1(\varepsilon) \sqsubseteq \mathcal{M}_2(\varepsilon)$ . Then  $\mathcal{M}_1(\varepsilon)$  is preferred. Of course, the notion of adequate covering is critical. Moreover, preferring one model over another does not necessarily mean rejection of the other. For instance, in the construction of an abstract model  $\mathcal{M}_1$  of a software system in the *VDM Meta-IV*, the reification process will give rise to a less abstract model  $\mathcal{M}_2$  where  $\mathcal{M}_1(\varepsilon) \sqsubseteq \mathcal{M}_2(\varepsilon)$ .  $\mathcal{M}_1$  would only be accepted if it were adequate, i.e., was able to capture the essential details of the system being modelled. In the philosophy of the *VDM*,  $\mathcal{M}_2$  is to be derived from  $\mathcal{M}_1$  by introducing more information. The preference of  $\mathcal{M}_1$  over  $\mathcal{M}_2$  is only valid with respect to certain criteria. Upon derivation of  $\mathcal{M}_2$  and subsequent application of reification, the application of Ockham’s razor is once more invoked. In this manner a chain of models is constructed leading from the original abstract model to the ultimate implementation:

$$\mathcal{M}_1 \sqsubseteq \mathcal{M}_2 \sqsubseteq \dots \sqsubseteq \mathcal{M}_j \sqsubseteq \dots \sqsubseteq \mathcal{M}_n$$

To reject, say  $\mathcal{M}_2$ , out of hand due to an application of Ockham’s razor and affirm that neither it nor its successors add anything to  $\mathcal{M}_1$  would be sheer folly in the *VDM* or in any other formal development method of a software system. An analogous argument is given by Popper (1972) *contra* the philosophers of the ‘radical behaviourist’ or ‘physicalist’ school who object to the introduction of mental states or events to explain human behaviour.

### 3.3. Conceptual Models

What is it that distinguishes conceptual models from non-conceptual models? We have already presented a case for conceptual model in the section on the  $\mathcal{MRC}$  notation. Essentially, the passage from model to conceptual model might be achieved by introducing the human element into the equation:

DEFINITION 1.3. *A conceptual model is an abstraction or simplification of reality. Let  $h$  denote a human being and  $\varepsilon$  denote some entity. Then  $\mathcal{M}(h, \varepsilon)$  denotes the conceptual model  $\mathcal{M}$  that  $h$  has of  $\varepsilon$ . An alternative notation that has some advantages is  $\mathcal{M}_h(\varepsilon)$ .*

But to what extent can a model be said to exist at all without the human element? For is it not the human element that determines the very existence of models? To elaborate further upon the nature of conceptual models, one might pose questions such as the following. Do dogs form/have models? Do dolphins form/have models? Do programs ...?

One is familiar with Descartes' famous 'Cogito, ergo sum'—'I think, therefore I am'. We use it to emphasise our difference from other animals. But what exactly does it mean 'to think'? Hadamard notes that Max Müller observes that the latin verb 'cogito' etymologically means to 'shake together' (Hadamard 1945, 29). Thinking and intelligence seem to go hand in hand. Hadamard further notes that St. Augustine had already noticed *that* [about the meaning of the verb 'cogito'] and also observed that 'intelligo' means to 'select among'. Now from these simple observations one can argue a good case for the soundness of artificial intelligence. For does one not shake 'facts' together in a computer? And what is the purpose of a rule-based system, if not to make a selection among possible choices? On such grounds is one able to include a computer system in the domain of entities possessing a conceptual model.

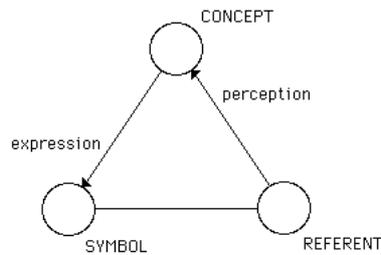
Consider a model aeroplane. Is it an aeroplane or a model of an aeroplane? Certainly, a prototype of a 'yet to be built' aeroplane is a concrete model. Yet, the very concreteness is derived from a conceptual model of the designer. Again, a piece of software has a real existence. It is a concrete manifestation of the conceptual model of its designer. Even a fashion model is a realisation of a projection of somebody's conceptual model of the 'ideal' man or woman. Thus, it is to be observed

## CONCEPTUAL MODELS

that there is a direct correspondence between conceptual models and concrete realizations or representations of conceptual models. Moreover, said representations have an independent existence, are distinguishable from other entities in the real world, and are never perfect!

### 3.3.1. Formation of Conceptual Models

A fruitful starting point is to posit the Aristotelian hypothesis (Sowa 1984, 11) that a concept is formed as a result of the perception of real world entities (the referents). The word (or symbol) is an expression of the concept. But the link from symbol to referent is indirect. Following Saussure, one would assert that the link is entirely arbitrary. One may sum up these relationships by the meaning triangle:



The Meaning Triangle

where ‘the left corner’ is the *symbol* or *word*; ‘the peak’ is the *concept*, *intension*, *thought*, *idea*, or *sense*; and ‘the right corner’ is the *referent*, *object*, or *extension* (Sowa 1984, 11).

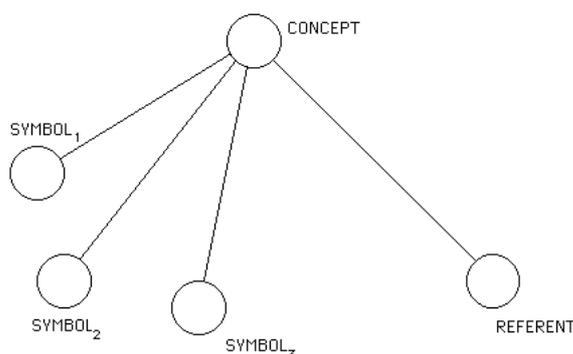
Sowa’s *Conceptual Structures* is probably the most comprehensive treatise on conceptual models and proved to be a major formative influence in my own thesis, as already acknowledged earlier. However, whereas his point of departure is natural language, I have chosen to focus exclusively on the symbolic, in the abstract sense of that term. By abstract symbol I wish to mean notation, as distinct from the word. In particular, as shall be discussed in the sections on representations and encodings, my starting point lies at the left corner of the triangle in the worlds (plural!) of symbols as shown in the extended meaning triangle on the following page.

But there is no reason why one should not cycle anticlockwise around the meaning triangle, why a particular collection of symbols should not in themselves be regarded as the referent, the subject of perception. The expressed takes on an independent existence of its own and may therefore be deemed to be an entity. Such an

approach is very much at the heart of all computer science. A simple example may suffice to illustrate this point.

Consider the entity ‘scenario’, which the Bart Veth Group of CWI used in the context of Intelligent CAD systems (ICAD) (Veth 1987), and which is considered to be the key concept of the Tokyo IDDL (Tomiyaama, Xue and Ishida 1989). To the average (!) educated native English speaker, it conjures up the notion of a ‘scene’ in a ‘play’. There are ‘actors’ involved who play particular ‘rôles’, dialogue ‘scripts’ which determine what they say and how they ‘act’, and ‘stage props’ (which may be minimal) to support the drama. Let us denote a (conceptual) model of such a scenario by  $\mathcal{M}_\mu(\text{‘scenario’})$ .

One can say that a person understands the meaning of ‘scenario’, i.e., has a conceptual model of ‘scenario’ by the language they use in speaking about it. Thus, the author has effectively exposed a part of his understanding of the meaning of ‘scenario’ in the previous paragraph.



The Extended Meaning Triangle

### 3.4. Mental Models

Now, in order to avoid confusion, it seems to me to be imperative that I clearly distinguish between the terms ‘conceptual’ model and ‘mental’ model. Norman distinguished between mental model and conceptual model, asserting that the latter is “invented by teachers, designers, scientists, and engineers” (Norman 1983, 7). Such a notion of conceptual model, with which I entirely agree, I term *prescriptive*. Using a notation similar to that of the *MRC* notation, he denoted the mental model that a person has of some target system  $t$  by  $M(t)$ . The conceptual model of  $t$  was denoted  $C(t)$ . Then one distinguishes between the conceptualisation of a mental model,  $C(M(t))$ , and the actual mental model that one thinks a person might have,

## CONCEPTUAL MODELS

$M(t)$ . Norman’s idea of ‘mental’ model is what I term a *descriptive* conceptual model. Effectively, all approaches that attempt to construct conceptual models on the basis of natural language processing are descriptive!

I also make a distinction between ‘conceptual’ model and ‘mental’ model. I take the view that mental models subsume conceptual models. It is predicated that mental models result from a wide variety of external stimuli, language, images, sensations, etc., and that their existence and nature is exhibited as much by behaviour as by verbal or written expression. Conceptual models, on the other hand, are predicated to be expressible in written forms—whether in language (natural or formal) or in pictures. It is further hypothesised that only conceptual models are the meat of computing and that the Turing thesis is only applicable to same. This latter hypothesis would seem to be totally contrary to the view of Johnson-Laird who adopts the principle that “mental models, and the machinery for constructing them, are computable” (Johnson-Laird 1983, 398), but one with which I might begin to agree were the word ‘mental’ changed to ‘conceptual’.

### 4. Representations

I have already asserted that conceptual models are not *directly* perceivable, that their existence may be inferred from representations. More importantly, the construction and growth of the conceptual model proceeds by way of representations. But what exactly do I mean when I speak of representation? In classical philosophical terminology it is the form that supports, or is the carrier of, the content. To introduce the notion of representation, and to distinguish form from content, consider the following recursive definition of a (homogeneous) list:

1.  $\Lambda$  is a list, called the empty list or null list.
2.  $\langle h \mid t \rangle$  is a list if  $h$  is an element from some domain and  $t$  is a list.

There are some unstated conventions (or assumptions) being employed here. A list is always bracketed by ‘ $\langle$ ’ and ‘ $\rangle$ ’. The symbol ‘ $\mid$ ’ is used to separate the list into two parts. Individual elements occur to the left of ‘ $\mid$ ’ and the rest or remainder of

the list to the right of ‘|’. If  $l = \langle h | t \rangle$  is a list, then  $h$  is said to be the head of the list  $l$  and  $t$  is said to be the tail of the list  $l$ . Examples of lists are  $\Lambda$ ,  $\langle 2 \rangle$ ,  $\langle \text{dog, cat, mouse} \rangle$ ,  $\langle \text{John, age, 3, years} \rangle$ ,  $\langle 1, \langle 2 \rangle \rangle$ . Note that in this latter case of a non-homogeneous list,  $\langle h | t \rangle = \langle 1 \langle 2 \rangle \rangle$  implies that  $t = 1$  and  $t = \langle \langle 2 \rangle \rangle$ .

The formulation of such a definition uses a mixture of English and some special symbols: ‘ $\langle$ ’, ‘ $\rangle$ ’, ‘|’. The entire text within quotation marks may be said to be a (partial) representation of the concept of list in a particular formalism. But is not stated that the chosen notation has particular historical connotations. I have not stated that the chosen notation has particular historical connotations. I have adopted the list separation symbol ‘|’ from PROLOG. Instead of the bracketing symbols of PROLOG, and the English School of the *VDM Meta-IV* and  $\mathcal{Z}$ : ‘[’, and ‘]’, I employ those of the Danish School of the *VDM Meta-IV*. Why did I make that choice, one might ask? There is no particular good reason for adopting one over the other, other than perhaps to signal my strong preference for the approach of the Danish School. The notation is rather arbitrary in a Saussurean sense. Incidentally, I often prefer to use  $\langle h | t \rangle$  rather than the conventional *Meta-IV* form  $\langle h \rangle \wedge t$ .

The very choice of the word ‘list’ triggers strong associations with programming languages such as LISP and PROLOG and for that simple reason it ought not to be used in the context of specification. The old terminology of the *VDM Meta-IV* was ‘tuple’. But it had such strong connotations in the field of data bases that the term ‘sequence’, which has a well established mathematical connotation, is now used instead. There will always be a serious problem when using natural language words to denote the entity in question. But the abstract form of, say,  $\langle h | t \rangle$  is independent of the word used.

To concretise the notion of representation let us adopt the following working definition.

**DEFINITION 1.4.** *Let  $\mathcal{M}$  denote a conceptual model. Let  $\mathcal{F}$  denote some formalism. Then define a representation of  $\mathcal{M}$  in  $\mathcal{F}$ , denoted  $\mathcal{R}(\mathcal{M}, \mathcal{F})$ , to be a textual description of  $\mathcal{M}$  in  $\mathcal{F}$ .*

A textual description is one which is composed of two distinct disjoint sets of symbols  $\Sigma_1$  and  $\Sigma_2$ . The set  $\Sigma_1$  consists of those tokens which are used to form sentences in some natural language. The set  $\Sigma_2$  consists of notational marks. Although ideographs, say of Chinese, are in  $\Sigma_1$ , and some computer generated icons may be

## CONCEPTUAL MODELS

considered to be in  $\Sigma_2$ , the definition is intended to rule out pictures or diagrams as such. Although formalisms such as mathematical notation and logical notation are of primary concern, one may consider the formalism  $\mathcal{F}$  to denote any medium, subject to the criterion of “textual description”. I hasten to add that a formal specification is not merely a textual description in the above sense. Where appropriate, diagrams and pictures, must accompany the text. Such images are also representations in the wider sense of that term. By definition, a representation  $\mathcal{R}$  is a model of the entity  $\mathcal{M}$ .

PROPOSITION 1.4. *A representation  $\mathcal{R}$  of a conceptual model  $\mathcal{M}$  is an approximation of that model, written  $\mathcal{R} \sqsubseteq \mathcal{M}$ .*

EXAMPLE 1.4. The form  $\langle x \mid y \rangle$  is a model of a non-empty homogeneous list.

PROPOSITION 1.5. *The set of representations of a conceptual model  $\mathcal{M}$  forms a lattice with  $\sqsubseteq$  as the partial order.*

To say that the above example is a representation of the concept of list may be tested in the following manner. If the human being possesses such a concept then it is reasonable to expect him to be able to give examples of lists other than those cited. On the other hand, if he does not already have the concept of recursive function, it is unreasonable to ask him/her to design a recursive algorithm to find the length of a list, say. The concept of list is enriched by adding to it the operations that may be performed on a list. Consider the following sample definition of the length of a list:

$$\begin{aligned} \text{length: List} &\longrightarrow \text{Natural} \\ \text{length}(\Lambda) &\triangleq 0 \\ \text{length}(\langle x \mid y \rangle) &\triangleq 1 + \text{length}(y) \end{aligned}$$

Annotation:

- a) The length of a list is a mapping from the domain of lists (List) to the codomain of natural numbers (Natural).
- b) It is a function of one argument.
- c) It is given by a two line definition which corresponds exactly to the definition of list.
- d) The first line of the definition is the base case.

e) The second line of the definition is the recursive case; the right hand side argument is smaller than the left hand side argument—termination is guaranteed. The given algorithm is a representation. Several remarks are in order. Although formal, the algorithm is not ‘purely’ formal. There are still many important things left unsaid. Consider the signature. A more precise characterisation of the algorithm may be given by writing:

$$\text{length: } \Sigma^* \longrightarrow \mathbf{N}$$

where  $\Sigma$  denotes the underlying set of elements out of which the list is to be constructed. With this new representation I signal unambiguously that I am speaking about finite lists, at least within a certain set of accepted notational conventions. Moreover, I take it for granted that  $\Sigma$  itself is a finite set. Even the very choice of  $\Sigma$  has a special connotation—that of an alphabet of letters. Hence, the notion of ‘word’ is an interpretation of a list over an alphabet. If I do not wish to convey such a connotation, for whatever reason, then some anodyne notation such as  $X^*$  is to be preferred. The set of natural numbers  $\mathbf{N}$  is taken to mean the set  $\{0, 1, 2, 3, 4, \dots\}$ .

Again, one may very well ask why not use the names ‘size’, or ‘cardinality’ instead of ‘length’? In the *VDM Meta-IV* the abbreviation ‘len’ is used. To avoid commitment to particular words, one might wish to employ abstract symbols, such as ‘#’ as used in CSP and  $\mathcal{Z}$ , or even  $|-|$ , which is frequently employed in Mathematics. Given this latter form, then the body of the algorithm becomes:

$$\begin{aligned} |\langle x \rangle \wedge y| &= 1 + |y| \\ |\Lambda| &= 0 \end{aligned}$$

where I am obliged to write  $\langle x \rangle \wedge y$  in place of  $\langle x | y \rangle$  to void confusion. Even with this much representation a very large conceptual complex is implied. For instance, natural invokes the theory of natural numbers, and recursive leads to the theory of algorithms.

## CONCEPTUAL MODELS

### 4.1. Algorithms

It seems to be a truism to assert that every programmer must understand the concept of algorithm. One reasonable account of same is given by Knuth at the very beginning of *The Art of Computer Programming*, the first volume of which is graced with the title *Fundamental Algorithms* (Knuth, 1973). I do not propose to present an alternative definition.

*HYPOTHESIS 1.1. A programmer must have an adequate conceptual model of an algorithm.*

Ideally, the programmer will be familiar with at least one of the classical computational theories: Turing Machines, General Recursive Functions, Lambda Calculus, Markov Algorithms, Post Symbol-Manipulation Systems, etc., otherwise the extent to which he understands the concept of algorithm will always be in doubt. A programmer who is not so familiar can only be said to have an adequate concept of algorithm if his behaviour (and that of the program he writes) is deemed adequate by some authority.

*HYPOTHESIS 1.2. A programmer must have an adequate conceptual model of a computational machine—the concrete realisation of an algorithm.*

Programmers write programs in some programming language which is then executed on a physical computing machine. To the extent that a program executes correctly, one may consider the programmer to have an adequate conceptual model of a computational machine. However, one asks what nature such a model might have. Is it that of a real computer or something else? Extending Wegner's notion that 'a programming language can be thought of as a specification of a class of computers, where each computer in the class corresponds to a program of the language' (Wegner 1971), one may speak of a language-machine. Thus, one might have a Pascal machine, a Lisp machine, an Ada machine, etc.

*HYPOTHESIS 1.3. An imperative programmer has a conceptual model of computing which is equivalent to the Turing Machine.*

The Turing machine is essentially a finite-state machine with external storage (Minsky 1967) which uses the basic notions of 'alphabet', 'storage', 'reading', 'writing', 'moving', and 'state'. An imperative programmer is viewed as one who programs

a Turing machine, i.e., develops algorithms that rely on the notions of storage and state. Programming languages such as Pascal and Ada are usually considered to be imperative programming languages in precisely this sense, though it is possible to use them in a style that is different from the imperative style. More precisely, the Turing machine is the computational model of assembly language programmers.

*HYPOTHESIS 1.4. An applicative Programmer has a conceptual model of computing which is equivalent to the Lambda Calculus or Recursive Function Theory.*

I am totally in agreement with Hermes' observation (Hermes 1969) that "the concept of Turing-computability . . . is not flexible enough for the work of the mathematician" and, consequently, I might add, is totally inappropriate for specification. Whereas all computational models are equivalent in the sense that they all compute the same class of functions (Church's Thesis), there does appear to be a distinction in expressive power. The recursive definition of an algorithm essentially says what can be computed or more precisely what can be constructed rather than how it is to be computed. Concepts of storage and state may be given precise mathematical structure that is independent of any physical machine. In this sense it can be said that recursive algorithms are more abstract than those specified by Turing machine. But there are recursive forms that are very close to the imperative model. Consider the following representation of the length of a list in tail-recursive form:

$$\begin{aligned} \text{len}: \Sigma^* &\longrightarrow \mathbf{N} \longrightarrow \mathbf{N} \\ \text{len} \llbracket \Lambda \rrbracket s &\triangleq s \\ \text{len} \llbracket \langle x \mid y \rangle \rrbracket s &\triangleq \text{len} \llbracket y \rrbracket (s + 1) \end{aligned}$$

For  $s = 0$ , this algorithm computes the length of a list. For  $s \in \mathbf{N}$ ,  $s \neq 0$ , it computes 'length of a list +  $s$ ' and is, therefore, more general, contains more information than the earlier representation. The variable  $s$  denotes 'storage'.

Here one must make a clear distinction between imperative style algorithms and iterative algorithms. An algorithm may be both iterative and recursive, many examples of which occur in the domain of numerical analysis (Henrici 1964; Johnson and Riess 1977). For example, the solution of a fixed-point equation:

$$x = f(x)$$

has the iterative formulation

$$x_{n+1} = f(x_n)$$

## CONCEPTUAL MODELS

which in turn has the natural recursive definition:

$$\text{fixed\_point}(f, x_n, \epsilon) \stackrel{\Delta}{=} \text{if } |f(x_n) - x_n| < \epsilon \text{ then } f(x_n)$$

$$\text{fixed\_point}(f, x_n, \epsilon) \stackrel{\Delta}{=} \text{if } |f(x_n) - x_n| \geq \epsilon \text{ then fixed\_point}(f, f(x_n), \epsilon)$$

provided that convergence is guaranteed for some  $x = x_0$ . It is perhaps this confusion between iterative and imperative that prohibits Arzac from declaring that recursive algorithms are more abstract than iterative ones (Arzac 1985). Indeed, as will be evident from many of the examples in this thesis, there is a closer affiliation between iterative and recursive algorithms than there is between iterative and imperative algorithms.

*HYPOTHESIS 1.5. A PROLOG Programmer has a conceptual model of computing which is in part equivalent to the Markov Algorithm.*

The concept of the Markov algorithm (Korfhage 1966) depends on the concept of order. The order in which the productions are given is essential to the concept of the algorithm and this order is top to bottom. The evaluation of clauses in PROLOG also depends entirely on the order in which they are presented. The hypothesis is confined solely to PROLOG programmers and not to the wider group of Logic programmers or Relational Programmers. However, for this latter class, the remark by Arzac (Arzac 1985) that recurrence algorithms of a particular form, given by recurrence relations, may all be expressed by Markov algorithms, suggests that the equivalence may be greater than expected.

The notion of algorithm may be captured precisely by any of the formal systems that I have mentioned above. However, they are all too formal for the purposes of specification of algorithms. I will argue in the next section that encodings of algorithms are similarly ‘flawed’ in so far as they are also too formal and too detailed. The nature of specification is such that the representations to be used are to be formal and flexible enough in the sense of mathematics. On the other hand, too much formality and abstraction would be self-defeating. For then all links between the reality to be specified and the representational forms used would be lost. It is my goal to demonstrate how a harmonious balance may be struck.

## 5. Encodings

In the original *MRC* notation, I denoted that representation which is executable on a real machine by the term encoding. By real machine I intended to mean an actual computer or a virtual machine such as Landin’s SECD machine (Wegner 1971, 216–23) or Knuth’s MIX machine (1973, 1:120–227). I considered an encoding to be an expression in a programming language, which is largely determined both by its syntax and its semantics. A programming language, *qua* language, has a strong influence on the way programmers express algorithms and how they think about their programs.

For completeness, I widen the notion of encoding to include those expressions which are written in the formal notations of the different models of computation: Turing machines, Markov algorithms, the  $\lambda$ -calculus, etc. Now I can define precisely what I mean by the term executable specification.

**DEFINITION 1.5.** *A specification is executable if it is an encoding or if it may be transformed automatically into an encoding.*

In the remainder of this section I will focus exclusively on encoding as expression in a programming language and any remarks that I make on same are applicable *ceteris paribus* to the other forms of encodings introduced above. I have carefully drawn a distinction between ‘specification as representation’ and ‘specification as encoding’ in order to delineate the precise boundaries of the subject matter of the rest of the thesis—specification as representation.

### 5.1. Function and Algorithm

A specification is a model. I would like to say that it consists of some types and operations on those types. But use of the word type has connotations in programming languages that cause me considerable conceptual problems that I feel I ought to avoid. These are discussed below. Instead of type, I prefer to use the term set. The operations in question may be considered to be functions or algorithms and I insist on retaining a certain duality of concept in my work for reasons which I shall now make clear.

Stoy drew a clear distinction between function and algorithm by remarking that there may be many algorithms for the same function and that there are functions

## CONCEPTUAL MODELS

for which there are no algorithms (1977, 48). This is undoubtedly true in a platonic sense. However, I feel that his concern in drawing such a distinction was directly related to denotational semantics—to the implementability/constructability of certain functions. Within that context he was justified in drawing the distinction. But unfortunately he promoted a minor ‘heresy’, in the technical sense of that term—focus on some aspects of truth/reality to the exclusion of others. He had taken for granted that the essence of functionality is uniquely determined without indicating in what sense such functionality was to be expressed or represented.

The concept of function, as understood today, has a curious history. Its origin is customarily attributed to Dirichlet “who, in 1837 was led by his study of Fourier series to a major generalization in freeing the idea of a function from its former dependence on a mathematical expression or law of circumscribed kind” (Church 1956, 1:22). Lakatos furiously disputed the historical record, stated categorically that there was ample evidence in Dirichlet’s works to show that he had no idea of this concept and suggested that it was Hankel who was the originator of the ‘tale’ (1976, 151). Simply stated, a function associates values with its arguments. Consequently, we say that two functions are equivalent if the association is the same (cf., (Church 1956, 1:11–2)).

In computing, it is useful to consider the concept of algorithm along similar conceptual lines, associating inputs with outputs by a constructive mechanism. Two algorithms are equivalent if the association is the same but the mechanism is different. Stoy essentially asserts that there is a strict many-one mapping from the domain of algorithms to the domain of functions. I prefer to regard the mapping as one-one: to each algorithm there corresponds a function and then to consider the issue of the equivalence of functions. Finally, to complete the argument, I assert that, from the point of view of a conceptual model of computing, there are no interesting functions which are not the image of some algorithm. The whole goal in research in the theory of computation is, therefore, to expand the class of interesting functions.

## 5.2. The Concept of Type

I have referred above to the difficulties I have in using the term type in the context of specification as representation. The literature on typing in programming languages is already vast (cf., (Gotlieb and Gotlieb 1978; Zilles 1984)). In this subsection, typing is considered from the conceptual point of view. Sowa remarks that “many people confuse types and sets” and takes pains to point out the distinction (1984). However, by virtue of the fact that such confusion exists, one may infer that such people have conceptual models in which the concepts of type and set are unified. When such people play the rôle of programmer then one expects to find a behaviour which agrees with their conceptual model. Consider the type ‘integer’ as used in a programming language such as Pascal. Elements of the type are numbers drawn from a finite subset of  $\{0, \pm 1, \pm 2, \dots\}$ . In this case there is no distinction between type and set. Such a conceptual model has been reinforced by the designers of Ada: “A type in Ada is characterised by a set of values and a set of operations” (ALRM 1983). This notion of type is seen as the “widely accepted view” after twenty years of programming language history (Ichbiah et al. 1979), and is to be interpreted as the way in which one imposes structure on data. However, consider for example the ‘type definition’ of complex number in Ada:

```
type complex is
  record
    re: Real;
    im: Real;
  end record;
```

One is lead to the conclusion that it is not a type at all under the ‘definition’ of type as given above, unless one agrees that the set of operations may be empty. But if the programmer were to provide operations on complex number then one approaches the notion of type as defined. The Ada package may be used to group a ‘type definition’ and its operations into a single textual unit. Such a package would then *almost* fulfil the requirements for type. One practical problem remains. The set of possible operations on complex number would probably not be placed within the same textual unit. A more serious conceptual problem is related to the representation of complex number in Ada. The one given above is the cartesian product form. A different representation such as the polar form, which is also a cartesian product, must be

## CONCEPTUAL MODELS

accommodated by more programming language machinery.

Moreover, Zilles also highlights this confusion with respect to the notion of type and, taking the definition that a type is a set of entities, explains that the “set of operations” is handled by an algebra (1984). Thus he overrules the “widely accepted notion” as claimed by Ichbiah et al. above. In fact, the latter notion of type is subsumed under the notion of abstract data type.

Moving away from the family of imperative programming languages, let us consider the notion of type as predicate, a concept which dates from Aristotle who used the term ‘kategoria’ (category), the original meaning of which was ‘predicate’ (Lloyd 1968). An entity is of a specific type if there is a predicate which asserts that it is so. Thus, in a programming language such as PROLOG, one may explicitly list the elements of a type and provide a clause which checks that a given entity belongs to the type, or alternatively, provide a clause which generates the elements of a type upon request. Even here I can not draw any distinction between type and set.

Finally, from a philosophical point of view, Wittgenstein(1974) considers the terms ‘concept’ and ‘object’ as used by Frege and Russell to mean ‘property’ and ‘thing’, respectively, and that these are, in effect, the same as ‘predicate’ and ‘subject’ which in turn are related by the notational forms  $x$  is  $T$ , or  $x \in T$ , where  $T$  is some type. In other words, type, concept, property, predicate are terms used to denote essentially the same notion.

I believe that I have presented ample evidence to suggest that founding a theory of conceptual models in which the notion of type appears would run into severe difficulties. The problem appears precisely because of the connotations of the term in programming languages which rely on ‘strict typing’.

### 5.3. On the Inadequacy of Encodings

In building up a theory of conceptual models in computing, I considered abstracting from the pragmatic reality of programming languages and computing environments. In other words, from a study of certain forms—encodings and environments, I had hoped to identify the essential content of computing. In itself, the approach was pedagogically sound, a movement from the concrete to the abstract. But it was not enough. Every encoding considered, whilst it enriched my model, proved to be critically flawed by its very inflexibility. To complement the research, I turned my attention to the domain of specifications as representations. The necessary quantum jump was provided by the *VDM*, which is subject of the next chapter.

## 6. Summary

The concept of model is central to this work. The thesis itself is a metamodel of conceptual models in computing. To exhibit a model, some form of expressive notation has to be employed, together with appropriate descriptive text and accompanying diagrams.

The *MRC* notation provided a useful starting point in which to represent some of the key concepts. In particular it served to distinguish the notions of conceptual model, representation, and encoding. However, its use was very limited. Specifically, there were no operations that might be applied to the entities expressed. Of the three main entities, that which proved to be the most fruitful was representation, which corresponds exactly to the notion of ‘form’ or *Gestalt* in philosophy. The relationship between form and content with respect to a conceptual model is exactly analogous to that between syntax and semantics with respect to a programming language.

To build up my conceptual model of computing, I had initially resorted to encodings, i.e., expressions in some concrete syntaxes. To complement the work, the issue of semantics had to be addressed and since the *VDM* had initially been developed precisely for the expression of the denotational semantics of programming

## CONCEPTUAL MODELS

languages, it turned out to be a natural choice for the next stage.

The specification language of the *VDM*, called *Meta-IV*, ultimately offended my æsthetic sense. There was a certain verbosity in the style of its use which seemed to clash with the goals that I set myself in developing a constructive mathematics of specification. Whereas, I was content to use *Meta-IV* as is, in the domain of denotational semantics, it proved to be unwieldy in the domain of computer graphics and computer-aided design. The elegant simplicity of the mathematics in the latter area contrasted strongly with the clutter of traditional *Meta-IV*. Moreover, there was a strong tendency emerging to turn the *Meta-IV* into a language that might easily be processed by computer. I became concerned that the *Meta-IV* might become another APL.

I have referred to the development of software as being similar to the scripting of a tragedy in the classical Greek sense. One of key notions behind the concept of tragedy is the focus on development of plot rather than on the characters themselves. I have identified specification as representation to be the key to understanding this development and am now ready to present the elements thereof.

## Chapter 2

# An Operator Calculus for the VDM

### 1. Introduction

The acronym *VDM* stands for the ‘Vienna Development Method’, a brief history of which is given in (Bjørner et al. 1987). Its origin is to be found in the concrete reality of trying to solve the problem of the systematic development of a compiler for the PL/1 programming language. Today, the method is used (almost exclusively) for the systematic development of software systems. A meta-language, called *Meta-IV*, which is part of the *VDM*, is used for formal specifications. Not surprisingly, much of the flavour of *Meta-IV* is due to the denotational semantics approach adopted by the originators of the language. To understand some of the philosophical underpinning of *Meta-IV*, the following quotation is noteworthy:

“We stress here [...] that the meta-language [*Meta-IV*] is to be used, not for solving algorithmic problems (on a computer), but for specifying, in an implementation-independent way, the architecture (or models) of software. Instead of using informal English mixed with technical jargon, we offer you a very-high-level ‘programming’ language. We do not offer an interpreter or compiler for this meta-language. And we have absolutely no intention of ever wasting our time trying to mechanize this meta-language. We wish, as we have done in the past, and as we intend to continue doing in the future, to further develop [*sic.*] the notation and to express notions in ways for which no mechanical interpreter system can ever be provided” (Bjørner and Jones 1978, 33).

I took this expressed intention seriously and considered how that goal might be achieved. One overriding concern was to bind definitively the *Meta-IV* with classical algebraic structures and to exhibit the potential for the establishment of a body of theorems that might usefully be employed in formal specifications. However, the

## AN OPERATOR CALCULUS FOR THE VDM

phrase “not for solving algorithmic problems” jarred! There is no good reason why *Meta-IV* might not be used for just such a purpose. Indeed, one of the major goals of this work was to exhibit *Meta-IV*’s potential in this domain. Certainly it is true that one must distinguish carefully between the (formal) specification of systems and the (formal) specification of algorithms; the nature of the two activities is different (conceptually) though the same *Meta-IV* notation may be employed for both. Therein lies its power. Moreover, I consider it important to establish an operator calculus for *Meta-IV* that might remove some of the burden of always having to “think about what one is doing”, a dangerous admonition against which Whitehead railed in 1911—“It is a profoundly erroneous truism [...] that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case” (Whitehead [1911] 1978, 41). Indeed, this concern of mine, to develop an operator calculus, is very similar to that of Schönfinkel with respect to his notion of a *function calculus* (Schönfinkel [1924] 1967, 359), which is further developed below under the notion of ‘currying’. There is sufficient historical data to argue that the use of (terse) notation (in the mathematical sciences) has profoundly increased the problem-solving abilities of humans by providing them with an abstract language of considerable expressive power (c.f., Pólya’s remarks on notation ([1945] 1957, 134–41)). Just as one does not always think (consciously) of what one is doing when one speaks, so it is the case with the use of abstract notation. At present there is too much thinking involved with the use of a formal notation such as that of *Meta-IV* in the development of formal specifications.

Now it is important to be clear that the *VDM* is much more than its specification language *Meta-IV*. It is a systematic (software) development method! The *VDM* has been criticised for its inability to handle the specification of concurrent systems (Prehn 1987). However, *Meta-IV* has been wedded to CSP (Bjørner and Oest 1980; Storbank Pedersen 1987) and SMoLCS (Astesiano et al., 1987) to specify the formal semantics of the Ada programming language. There is absolutely nothing in *Meta-IV* which prohibits or obstructs such marriages. The real issue at hand is not at the syntactic level but rather at that of semantics. Indeed, *Meta-IV per se* is not much more than a body of discrete mathematics. I will demonstrate in the Chapter on *Communications and Behaviour* that there is a notation and a natural method of progression by which one may move from structure to communication

between agents having state and finally to descriptions of behaviour. Said chapter is a corner stone in establishing the use of the *VDM* in specifying/describing user conceptual models, the constructive goal of the thesis.

### 1.1. The *VDM* Schools

There is essential universal agreement on what constitutes the *VDM*. However, there are basically two major Schools of the *VDM* largely distinguished by notational differences employed in the specification language *Meta-IV*—the Danish School and the English School. The Danish notation tends towards the mathematical, the symbolic, whereas the English tool tends to be more verbose, drawing, to a certain extent, inspiration from both the Pascal programming language (Jones 1986) as well as the formal specification language  $\mathcal{Z}$  of Oxford (Hayes 1987). But even this categorisation is inadequate. For indeed, certain of the notational variants of the English School (taken from  $\mathcal{Z}$ ) are exciting in their expressive power, the most notable of which are the removal operator  $\Leftarrow$  and the restriction operator  $\triangleleft$ , to be discussed in detail later.

With respect to method, the English School tends very much towards the use of pre- and post-conditions in specifications. It adopts a formalist approach, with emphasis on three-valued logic and proofs are conducted in a tedious logical deductive style, which I dislike intensely. The Danish School has preferred the explicit ‘operational’ approach. But even here organisational preferences might use a mixed approach—choosing the pre- and post-condition style while retaining the terseness of notation typical of the Danish School. However, I will demonstrate how my operator calculus applied to invariants and proofs turn out to be equivalent to pre- and post-conditions in many cases.

There is also the Polish School, which finds expression through the MetaSoft project (Blikle 1987, 1988, 1990). I will frequently need to distinguish between the style of notation and method that I use from those of the other Schools of the *VDM*. I *presume* to use the phrase ‘the Irish School of the *VDM*’ to draw that distinction.

# AN OPERATOR CALCULUS FOR THE VDM

## 1.1.1. BSI/VDM-SL

A desire for the unification of the notations of the two main Schools of the *Meta-IV* and the construction of computer-based tool support for the European software industry has led to the proposal of the international standardisation of the *Meta-IV*, currently known as the BSI/VDM-SL, where the acronyms BSI and SL stand for ‘the British Standards Institute’ and ‘Specification Language’, respectively (Sen 1987, Andrews 1988). There is as yet little material which is readily available in the public domain. Tutorial Notes and a proto-standard syntax do exist (Storbank Pedersen 1990). A standard notation does have certain advantages, the chief one of which is that it provides a sound basis for tool support. The principal disadvantage is that flexibility is inhibited and I maintain that flexibility is crucial in developing a conceptual model for those engaged in formal specification. There is room enough for a standard *and* the operator calculus of the Irish School of the *VDM* that I propose.

## 1.1.2. RAISE

From the point of view of the European software industry, the existing *VDM* was considered to be inadequate for very large scale software specification, the two principal problems identified being the handling of concurrency to which I have already briefly referred above and support for modularity. To address these issues, the successor to the *VDM* was engendered—RAISE (Rigorous Approach to Industrial Software Engineering) (Prehn 1987). My solution to concurrency, at least at the notational level, is in very broad agreement with that adopted in RAISE (Havelund 1990, 187). With respect to structuring, the RAISE solution is conceptually very similar to that of Ada, a solution which unfortunately and perhaps inevitably suffers from a diarrhoea of verbosity. I believe that structure in formal specification is more of an algebraic issue than a syntactic issue and I do not mean to imply that RAISE does not concern itself with the former. Rather, as I will demonstrate, too much reliance on the *name* of an algebraic structure can prohibit one from *seeing* the structure and the relationships (or morphisms) that it may have with other structures. This is not solely a RAISE problem. It is as much a problem with the style of use of the *VDM* of both the Danish and English Schools.

One further aspect of RAISE needs to be mentioned at this point. In formal

specifications one may identify two main streams of thought: the model-theoretic approach (*VDM* and  $\mathcal{Z}$ ) and the axiomatic approach (CLEAR and OBJ). RAISE has sought to combine the two approaches in a unified manner. I believe that the model-theoretic approach is the more natural from a conceptual point of view and that axiomatics are a consequence of experience with the properties of structures and their operations. In particular, I will argue that the axiomatic approach is non-intuitive and initial focus on it is actually harmful towards the development of the conceptual model of those engaged in formal specification.

## 1.2. Currying/Schönfinkeling

To complete this introduction and before I launch into the details of the basic domains and operations of the Irish School of the *VDM*, I wish to introduce one of the fundamental notational principles of same—currying or schönfinkeling.

The technique which replaces a function of several variables by a complex function of one variable is traditionally called ‘currying’ (Brady 1977). The concept and technique is originally due to Schönfinkel ([1924] 1967, 359; Bjørner et al. 1978, 201; Stoy 1977, 40). According to Quine “all functions, propositional and otherwise, are for Schönfinkel one-place functions, thanks to [...] ingenious device (which was anticipated by Frege (1893 § 36))”. It proved to be a useful technical device in the treatment of the  $\lambda$ -calculus. Today, it finds widespread use in the domain of the denotational semantics of programming languages. With respect to the *VDM Meta-IV*, it is an important notational form used extensively for example by Blikle (1987). In my own work in the *Meta-IV*, currying occupies a central position. Although I use it very much in the spirit of Schönfinkel, I do permit myself the liberty to employ ‘limited’ or partial currying, according as the intended expression conveys my particular conceptual viewpoint with respect to a given problem. As well as permitting the expression of tail-recursive algorithms, currying is fundamental to the development of communications and behavioural specifications. In addition it enjoys some elegant mathematical properties.

As an illustration, consider the definition of the ‘plus’ operation applied to two natural numbers, an example which I have chosen since it should be well-understood by all and was chosen by Stoy for the same purpose (1977, 40). It may be expressed in the form

## AN OPERATOR CALCULUS FOR THE VDM

$$\begin{aligned} \text{plus}: \mathbf{N} \times \mathbf{N} &\longrightarrow \mathbf{N} \\ \text{plus}(m, n) &\stackrel{\Delta}{=} m + n \end{aligned}$$

The curried version is

$$\begin{aligned} \text{plus}: \mathbf{N} &\longrightarrow (\mathbf{N} \longrightarrow \mathbf{N}) \\ \text{plus}[[n]]m &\stackrel{\Delta}{=} m + n \end{aligned}$$

It should be self-evident that the *essential meaning* of ‘plus’ is the same whichever signature is chosen. However, the form of the signature impels one to adopt a particular interpretation or view of ‘plus’, literally a *Gestalt* shift. The standard signature strongly suggests the view that ‘plus’ is a binary operation. On the other hand, the curried signature leads one to the interpretation that  $\text{plus}[[n]]$  is a function, in fact a whole family of functions indexed by  $\mathbf{N}$ ! This view may be reinforced by expressing the definition in the form

$$\text{plus}[[n]]: m \mapsto m + n$$

Other notational variants are possible. Schönfinkel used a bracketless form,  $\text{plus } n m$ , which was to be interpreted as  $(\text{plus } n)m$ . But he was careful to state that although the brackets were not visible, they were nevertheless present. He also used  $\text{plus}_n m$ . Blickle uses  $\text{plus}.n.m$ . I generally use either the denotational bracketted form given above, or occasionally the subscripted form. Now consider the effect of applying  $\text{plus}[[0]]$  to  $m$ :

$$\text{plus}[[0]]: m \mapsto m + 0 = m$$

Thus, it is clear that  $\text{plus}[[0]]$  is the identity function. As might be expected, there is a natural composition of such functions:

$$\begin{aligned} \text{plus}[[n_1 + n_2]]: m &\mapsto m + (n_1 + n_2) \\ &= (m + n_1) + n_2 \\ &= \text{plus}[[n_2]](m + n_1) \\ &= \text{plus}[[n_2]](\text{plus}[[n_1]]m) \\ &= (\text{plus}[[n_2]] \circ \text{plus}[[n_1]])m \end{aligned}$$

and thus  $\text{plus}[[n_1 + n_2]] = \text{plus}[[n_2]] \circ \text{plus}[[n_1]]$ . Finally, it should be obvious that the usual law of associativity of the composition of functions applies here.

It has just been demonstrated that the set of all functions  $\text{plus}[[n]]$ ,  $n \in \mathbf{N}$ , denoted  $\text{plus}[[\mathbf{N}]]$  is the monoid  $(\text{plus}[[\mathbf{N}]], \circ, \text{plus}[[0]])$ , where  $\circ$  denotes functional composition and  $\text{plus}[[0]]$  is the identity function. The term monoid is formally used

to summarise the structure of the set of said functions under the law of composition. Furthermore, this monoid is isomorphic to  $(\mathbf{N}, +, 0)$ . That is to say, given the mapping  $\text{plus}[\![n]\!] \mapsto n$ , there is no real difference between the structure of the monoid  $(\text{plus}[\![\mathbf{N}]\!], \circ, \text{plus}[\![0]\!])$  and the monoid of natural numbers under addition. From a conceptual model point of view, all that one needs to know about  $(\text{plus}[\![\mathbf{N}]\!], \circ, \text{plus}[\![0]\!])$  is given by  $(\mathbf{N}, +, 0)$ , the latter being the canonical interpretation of the former, a notion which plays a critical rôle in reusing specifications. As a final remark to add to my earlier comments on Stoy's concept of function in Chapter 1, I am surprised that he did not draw the conclusion that his platonic 'plus' function did not exist! The technique of currying effectively opens avenues of interpretation at the semantic level on the very meaning of function.

### 1.2.1. The Assignment Statement

It is customary to find currying explicitly used in denotational semantics, the application domain wherein it is most frequently to be found today. For example, given the syntactic domain

$$\begin{aligned} \text{Assign} &:: Id \times Exp \\ \text{for } mk\text{-Assign}(i, e) \text{ use } &:= (i, e) \end{aligned}$$

and the semantic domains

$$\begin{aligned} ENV &= Id \xrightarrow{m} LOC \\ STORE &= LOC \xrightarrow{m} VAL \end{aligned}$$

then the semantics of the assignment statement may be expressed in the form

$$\begin{aligned} \mathbf{M}: \text{Assign} &\longrightarrow (ENV \times STORE \longrightarrow ENV \times STORE) \\ \mathbf{M}[\![ := (\mathbf{i}, \mathbf{e}) ]\!](\varrho, \sigma) &\triangleq \mathbf{M}[\![ := ]\!](\mathbf{M}[\![\mathbf{i}]\!], \mathbf{M}[\![\mathbf{e}]\!])(\varrho, \sigma) \end{aligned}$$

where  $\mathbf{M}$  is a generic name for a 'meaning' function. In the traditional *VDM Meta-IV* there are conventions for naming a particular function such as  $\mathbf{M}$ . For example, in the signature, the usual name for  $\mathbf{M}$  is *Int-Assign*, the prefix *Int-* being an abbreviation for 'interpret' which suggests an interpreter. It is customary in denotational semantics to isolate the syntactic argument by the use of special parentheses, here denoted by ' $\![\!]$ ' and ' $\!]$ '. This is particularly true of the Oxford style of denotational semantics. Although currying is widely used in the traditional *VDM*

# AN OPERATOR CALCULUS FOR THE VDM

such distinguishing brackets have not been used in the literature. I suspect that this has been due to the use of traditional printing technology.

## 1.2.2. The Stack

Naturally, one may employ currying more generally than denotational semantics. Its use for the definition of ‘plus’ has already been exhibited. Here, a partial model-theoretic specification of an unbounded stack is presented

$$STACK = ELEMENT^*$$

The standard specification of a *Push* operation may be expressed as

$$\begin{aligned} Push: STACK \times ELEMENT &\longrightarrow STACK \\ Push(s, e) &\triangleq \langle e \rangle \wedge s \end{aligned}$$

the signature of which is very similar to what one might expect to find in algebraic specifications. A corresponding curried version is

$$\begin{aligned} Push: ELEMENT &\longrightarrow (STACK \longrightarrow STACK) \\ Push[e]s &\triangleq \langle e \rangle \wedge s \end{aligned}$$

Two interpretations are suggested by the curried form:

1. Considering *ELEMENT* as a syntactic domain and *STACK* as a semantic domain, then *Push* may be viewed as a ‘meaning’ function.
2. Going still further,  $Push[e]$  may be considered in its entirety to be a command (i.e., an action) which causes a *STACK* to *STACK* transition. In other words, although the semantics is functional, the interpretation is imperative (i.e., procedural).

Now there is nothing sacrosanct about the use of special double braces for the element argument  $e$ . Indeed, one may dispense with bracketting altogether and use the form,  $Push\ e\ s$ , such as may be found in functional programming languages: (a variant of) ML (Salmon and Slater 1987) or Miranda (Field and Harrison 1988). Alternatively, the CSP-like notation,  $s \triangleq Push?e \rightarrow \langle e \rangle \wedge s$ , might be employed (Hoare 1985). Indeed, the spatial layout may be altered from the above 1-dimensional to a 2-dimensional one:

$$s \xrightarrow{Push[e]} \langle e \rangle \wedge s$$

which recalls the transition notation of SMO LCS (Astesiano and Reggio 1987). It is precisely this *Gestalt* shift that led to the development of my approach to the use

of the *VDM* for the specification of communication and behaviour. There is nothing new in exploiting notational variants to reinforce a particular conceptual view of functions. The application domain of formal languages and automata provides another well-known example (Eilenberg 1974, 1).

EXAMPLE 2.1. Let  $Q = \{0, 1\}$  denote a set of states,  $\sigma \in \Sigma$  denote a letter of an input alphabet, and  $\delta: Q \times \Sigma \longrightarrow Q$  denote a state transition function. Suppose that for a particular automaton one has the transition  $\delta(0, \sigma) = 1$ . This may be expressed in a variety of other forms:  $0 \xrightarrow{\sigma} 1$ ,  $0\sigma = 1$ ,  $\sigma(0) = 1$ ,  $F_{\sigma}: 0 \mapsto 1$ ,  $F[\sigma]0 = 1$ , etc.

Now, the important point which I wish to stress, is that the alternative forms are all variants of (implicitly) curried functions. In other words, currying is standard in Automata Theory, even though the name may not be used.

Finally, one may wish to employ the conventions of the *VDM* as used in denotational semantics to give yet another semantically equivalent variant. If we consider *STACK* to be a semantic domain and define the abstract syntax

$$Push :: ELEMENT$$

then the semantics of *Push* may be specified by

$$\begin{aligned} Int\text{-}Push: Push &\longrightarrow STACK \longrightarrow STACK \\ Int\text{-}Push\llbracket mk\text{-}Push(e) \rrbracket s &\triangleq \langle e \rangle \wedge s \end{aligned}$$

This ability to switch between different syntactic forms and to recognise the similarity of semantic notions underlying each is absolutely crucial to the proper development of a conceptual model for formal specifications. Whereas the double bracketing notation is adopted as the main syntactic form for the purposes of exposition in this work, transformations to other syntactic forms will be freely given to illustrate significant conceptual points.

The remainder of the Chapter deals with the basic elements of the Irish School of the *VDM Meta-IV*: sets, sequences, maps, trees and cartesian products. Bags or multisets, and relations are introduced in Chapter 5. I have relied heavily upon a body of well-established mathematics and *VDM* to give ‘meaning’ to many of the assertions that I make since I do not wish to ‘reinvent the wheel’. Only where my own conceptual viewpoint with respect to the *VDM* needs to be emphasised do I explicitly invoke authorities. Inevitably, because of the linear structure of the text, I am obliged to make many forward references.

# AN OPERATOR CALCULUS FOR THE VDM

## 2. The Domain of Sets

Sets and their operations are usually well covered in secondary (i.e., post-primary) or high school and it may be presumed that they are well understood. For completeness, simple definitions of operators with some remarks more appropriate to *VDM* users are presented here. Among the basic sets used are

<b>N</b>	the natural numbers $\{0, 1, 2, \dots\}$
<b>N<sub>1</sub></b>	the strictly positive natural numbers $\mathbf{N} \setminus \{0\}$
<b>Z</b>	the rational integers $\{0, \pm 1, \pm 2, \dots\}$
<b>Q</b>	the rational numbers
<b>R</b>	the real numbers
<b>C</b>	the complex numbers
<b>B</b>	the set of boolean values

Consider the finite set of natural numbers  $\{0 \dots k - 1\}$ , where  $k \in \mathbf{N}_1$ . I use the notation **k** to denote this set, in order to trigger connotations with the set of digits at base  $k$ , used in the Theory of Automata (Eilenberg 1974, 1: 104). In Group Theory, the same set is denoted  $\mathbf{Z}_k$  and I have occasion to use this alternative form. The set of numbers  $1 \dots k$ , I denote by **k'**, where the prime has the connotation of successor.

In the *VDM* the set domain is a powerset! (For the origin of the term ‘power set’ see (Jacobson 1974, 5)). This is a source of conceptual difficulties for beginners of the method. To a certain extent this is due primarily to the use of syntax such as ‘**set of...**’ (English School) or ‘**...-set**’ (Danish School). In this work, the classical powerset notation  $\mathcal{P}$  is employed exclusively. Consider, for example, the domain equation

$$DICT = \mathcal{P}WORD$$

which models a spelling-checker dictionary  $\delta$ ,  $\delta \in DICT$ , as a set of words  $w$ , where  $w \in WORD$ . Now the important point is that the above domain equation covers all possible dictionaries  $\delta$ , some of which are

$$\emptyset, \{w_1\}, \{w_1, w_2\}, \dots, WORD$$

It is customary in mathematics to denote the empty set by  $\emptyset$ . In the *VDM*, the empty set is traditionally denoted by  $\{ \}$  and I occasionally use this convention for the

simple reason that it is consistent with the usual notation for empty sequence  $\langle \rangle$ , and empty map  $[]$ , used by the Danish School. But, from the point of view of aesthetics, a fundamental principle to which I try to pay great attention, I have noticed that the traditional *VDM* conventions lead to specifications which suffer from a diarrhoea of brackets which is even uglier than the ‘Lots of Irritating Spurious Parentheses’. I have found the conventions of Automata Theory more pleasing:  $\Lambda$  and  $\theta$  for the the empty sequence and the empty map (Eilenberg 1976, 2:3), respectively.

Whereas, there is no difficulty in principle in using the powerset constructor in formal specifications, there is an important computability issue that arises in considering (ultimate) implementation. For example, although the set of natural numbers  $\mathbf{N}$  is countable (cardinality  $\aleph_0$ ), the powerset  $\mathcal{P}\mathbf{N}$  (cardinality  $2^{\aleph_0}$ , equal to the ‘continuum’, the cardinality of the real number line) is not! This is a classic example considered by Cantor (Davis and Hersch 1980) who used the famous diagonal proof to establish the result. For this reason, some authors such as (Rydeheard 1986) prefer to confine such powerset domains to finite sets, effectively using a finite set constructor (written *FinSet*) and denoted here by  $\mathcal{F}$ . Thus, if a set  $A$  is finite then the cardinality of  $\mathcal{F}A$  is also finite. However, in practice, I do not care to make such a distinction at the notational level. For, even though arbitrarily infinite sets are expressible, the method that I expound is constructive, i.e., only finite sets ever arise from constructions.

Recalling the issue I raised with respect to the confusion between the notions of type and set in Chapter 1, a set may be specified either by listing its elements explicitly or by using a predicate. For example, the set  $\mathbf{3}$  may be given either in the form  $\{0, 1, 2\}$ , or  $\{j \mid j \in \mathbf{N}, j < 3\}$ . The latter form is, of course, essential when one wishes to identify an infinite set.

# AN OPERATOR CALCULUS FOR THE VDM

## 2.1. Set Operations

The set operations are union ( $\cup$ ), difference ( $\setminus$ ), intersection ( $\cap$ ), membership ( $\in$ ), cardinality (*card*), proper subset ( $\subset$ ), subset ( $\subseteq$ ), and distributed union ( $\cup/\cup$ ). Naturally, if one prefers, one may introduce the counterparts of subset: proper superset ( $\supset$ ) and superset ( $\supseteq$ ). A particularly interesting operation from both a practical as well as theoretical point of view is symmetric difference ( $\Delta$ ). I take it for granted that the meaning of these operations are well-understood. Consequently, I give definitions of them, with supplementary comments, to illustrate the sort of specifications that one might expect to find in the style of the Irish School of the VDM. Hence, it is deliberate that the specifications of set union and set intersection, which would be assumed to look very similar, actually appear to be rather different.

### 2.1.1. Membership

Test for membership of an element in a set is a fundamental operation. In this respect, the operator, here denoted  $\in$ , may be regarded as an infix operator with signature:

$$- \in -: \Sigma \times \mathcal{P}\Sigma \longrightarrow \mathbf{B}$$

A typical expression in which this might occur is

$$\text{if } x \in S \text{ then } \dots \text{ else } \dots$$

To express such use, the characteristic function is preferred:

$$\begin{aligned} \chi: \Sigma &\longrightarrow \mathcal{P}\Sigma \longrightarrow \mathbf{B} \\ \chi[x]S &\stackrel{\Delta}{=} \text{true}, & \text{if } x \in S; \\ &\text{false}, & \text{otherwise.} \end{aligned}$$

where the amended expression now reads

$$\begin{aligned} \chi[x]S &\rightarrow \dots \\ &\rightarrow \dots \end{aligned}$$

and a form similar to the McCarthy conditional is employed. An alternative that I often use in preference in written manuscripts is the subscripted form

$$\begin{aligned} \chi_S(x) &\rightarrow \dots \\ &\rightarrow \dots \end{aligned}$$

a notation which is frequently employed in mathematics (Jacobson 1974, 5). Consider once again the dictionary introduced above. A useful operation is to look up

a word. From a conceptual model point of view, this *is* the primary operation that one would want to perform on a spelling-checker dictionary. This may be formally specified by

$$\begin{aligned} Lkp: WORD &\longrightarrow DICT \longrightarrow \mathbf{B} \\ Lkp[[w]]\delta &\triangleq \chi[[w]]\delta \end{aligned}$$

Note carefully that this look up operation *is* the characteristic function itself!

Curiously however, in practice, the membership operator is more frequently used in the VDM as a choice operator, occurring in expressions such as

$$\text{let } x \in S \text{ in } \dots$$

where  $x$  is an arbitrary element to be chosen from the set  $S$ . For this use, the membership operator  $\in$  is retained.

From practical experience with specifications written in the Irish style, I have found it useful and expressive to recombine the two separate notions of ‘test for membership’ and ‘choice’. An example will illustrate the point. Let us suppose that I wish to specify the entering of a new word into the dictionary and I write, using the McCarthy conditional:

$$\begin{aligned} Ent: WORD &\longrightarrow DICT \longrightarrow DICT \\ Ent[[w]]\delta &\triangleq \chi[[w]]\delta \rightarrow \delta, \delta \cup \{w\} \end{aligned}$$

with the intention that if  $\chi[[w]]\delta$  is true then choose  $\delta$  else choose  $\delta \cup \{w\}$ . I find it more expressive to treat the characteristic function as a projection function over the ordered pair  $(\delta, \delta \cup \{w\})$ . Formally,

$$\chi[[w]]\delta(\delta, \delta \cup \{w\}) \triangleq \begin{cases} \delta, & \text{if } \chi[[w]]\delta; \\ \delta \cup \{w\}, & \text{otherwise.} \end{cases}$$

Returning to the original definition of the characteristic function, I note that there is considerable merit in replacing the definition with

$$\chi[[x]]S \triangleq \begin{cases} 1, & \text{if } x \in S, \\ 0, & \text{otherwise} \end{cases}$$

This now suggests using the characteristic function to express the idea of degree of membership of an element  $x$  in a set  $S$ , as used in fuzzy set theory, where the standard notation is  $\mu_S(x)$  (Klir and Folger 1988, 10). Specifically, the signature now becomes

$$\chi: \Sigma \longrightarrow \mathcal{P}\Sigma \longrightarrow [0, 1]$$

## AN OPERATOR CALCULUS FOR THE VDM

where  $[0, 1]$  denotes the unit real number interval. A generalisation of a different sort may be obtained by considering the characteristic function as a ‘counting’ function. Thus we may interpret  $\chi[x]S$  to be the number of occurrences of the element  $x$  in  $S$ , a notion which assumes more significance in the case of bags or multisets to be discussed later. Finally, dropping the name  $\chi$ , we obtain the notion of set as function:  $\{x\}: S \rightarrow \{0, 1\}$ ,  $\{x\}: s \mapsto 1$ , if  $s \in \{x\}$ , and 0, otherwise. This may be generalised to

$$A: S \rightarrow \mathbf{N}_1$$

where  $A(x)$  denotes the multiplicity with which  $x$  belongs to  $A$  (Eilenberg 1974, 1:126). Other generalisations will be mentioned in due course.

### 2.1.2. Subset

The subset operator  $\subset$  is essentially a generalisation of the set membership operator  $\in$ . Conventional definitions may be obtained in any good mathematical text. Here I prefer to give a constructive recursive definition:

$$\begin{aligned} - \subset - &: \mathcal{P}\Sigma \times \mathcal{P}\Sigma \rightarrow \mathbf{B} \\ \emptyset \subset S &\triangleq \text{true} \\ S_1 \subset S_2 &\triangleq \text{let } e \in S_1 \text{ in } \chi[e]S_2 \wedge ((S_1 \setminus \{e\}) \subset S_2) \end{aligned}$$

where, of course that it is understood that  $S \subset S \triangleq \text{false}$ . Set difference  $S_1 \setminus S_2$  is discussed below. To exhibit more clearly that the subset operator is indeed a generalisation of the characteristic function, consider the following simple illustration. Let  $X = \{x_1, x_2, \dots, x_k, \dots, x_n\}$ . Then the expression  $X \subset S$ ,  $X \neq S$ , may be understood to be an abbreviation for

$$\begin{aligned} X \subset S &= \chi[x_1]S \wedge \chi[x_2]S \wedge \dots \wedge \chi[x_k]S \wedge \dots \wedge \chi[x_n]S \\ &= \wedge/\{\chi[x_1]S, \chi[x_2]S, \dots, \chi[x_k]S, \dots, \chi[x_n]S\} \\ &= \wedge/\{\chi[x_1], \chi[x_2], \dots, \chi[x_k], \dots, \chi[x_n]\}S \\ &= \wedge/(\mathcal{P}\chi[-]\{x_1, x_2, \dots, x_k, \dots, x_n\})S \\ &= \wedge/(\mathcal{P}\chi[-]X)S \end{aligned}$$

where both the reduction,  $\wedge/$ , and the iteration,  $\mathcal{P}\chi[-]$ , will be defined rigorously later. The other significant point to note is that I sometimes find it more expressive to denote the set of operators,  $\{\chi[x_1], \chi[x_2], \dots, \chi[x_k], \dots, \chi[x_n]\}$ , by  $\chi[X]$ , which recalls Eilenberg’s concept of set as function mentioned above. With this notation, one has  $X \subset S = \wedge/\chi[X]S$ , subject to the constraint that  $X \neq S$ .

### 2.1.3. Union

Sets may be combined by applying the union operator. Formally, the union of two sets is given by:

$$\begin{aligned} - \cup -: \mathcal{P}\Sigma \times \mathcal{P}\Sigma &\longrightarrow \mathcal{P}\Sigma \\ S_1 \cup S_2 &\triangleq \{e \mid (e \in S_1 \vee e \in S_2)\} \end{aligned}$$

In the *VDM*, the union operator is most frequently employed for set extension, i.e., augmenting a given set by one or more elements as indicated by the frequently occurring expression of the form

$$S \cup \{x\}$$

Looking again at the specification of the operation to add a word to a dictionary

$$\begin{aligned} Ent: WORD &\longrightarrow DICT \longrightarrow DICT \\ Ent[[w]]\delta &\triangleq \delta \cup \{w\} \end{aligned}$$

one might wish to distinguish the cases whether the word  $w$  was already present in the dictionary  $\delta$  or not, in which case the body of the specification becomes

$$\begin{aligned} Ent[[w]]\delta &\triangleq \\ \chi[[w]]\delta &\rightarrow \delta \cup \{w\} \\ &\rightarrow \perp \end{aligned}$$

where, in the method that I propose in this thesis, the symbol  $\perp$  is to be taken to read ‘let’s not decide yet’ what to do about this situation now. It does *not* signify ‘error’ or ‘undefined’. In addition, note that I have chosen to use the ‘primitive’  $\chi[[w]]\delta$ , rather than  $Lkp[[w]]\delta$ . I frequently prefer to use a structural form of an operator in specifications rather than relying on the name of an operation that I have specified.

Using this notion of extending a set one element at a time leads to a simple recursive definition for set union:

$$\begin{aligned} \emptyset \cup S &\triangleq S \\ S_1 \cup S_2 &\triangleq \text{let } e \in S_1 \text{ in } (S_1 \setminus \{e\}) \cup (S_2 \uplus \{e\}) \end{aligned}$$

where I have used  $S \uplus \{e\}$  to denote the union of a set  $S$  with a singleton  $\{e\}$  with the intention that  $S_1 \uplus S_2$  is undefined if  $S_1 \cap S_2 \neq \emptyset$ . I frequently employ the  $\uplus$  notation in the recursive definition of algorithms over set domains. A generalisation of the ‘enter’ operation gives the notion of merging two dictionaries:

## AN OPERATOR CALCULUS FOR THE VDM

$$\circ/Ent: \mathcal{P}WORD \times DICT \longrightarrow DICT$$

$$\circ/Ent[[ws]]\delta \triangleq \delta \cup ws$$

Clearly  $\circ/Ent[[ws]]\delta$  is exactly equivalent to  $\delta \cup ws$  just as  $plus[[n]]m$  was equivalent to  $m + n$ . The use of the notation  $\circ/Ent[[ws]]$  may be justified in the following manner. Let  $ws = \{w_1, w_2, \dots, w_k, \dots, w_n\}$ . Then entering the set of words  $ws$  into the dictionary  $\delta$  may be expressed in the form

$$\begin{aligned} ws \cup \delta &= (Ent[[w_1]] \circ Ent[[w_2]] \circ \dots \circ Ent[[w_k]] \circ \dots \circ Ent[[w_n]])\delta \\ &= \circ/\{Ent[[w_1]], Ent[[w_2]], \dots, Ent[[w_k]], \dots, Ent[[w_n]]\}\delta \\ &= \circ/(\mathcal{P}Ent[-])\{w_1, w_2, \dots, w_k, \dots, w_n\}\delta \\ &= \circ/(\mathcal{P}Ent[-]ws)\delta \\ &= \circ/Ent[[ws]]\delta \end{aligned}$$

Note that the order of composition is not important, a misconception that might arise due to the indexing notation employed.

### 2.1.4. Intersection

The intersection operator may be used to find that subset which is common to two sets:

$$- \cap -: \mathcal{P}\Sigma \times \mathcal{P}\Sigma \longrightarrow \mathcal{P}\Sigma$$

$$S_1 \cap S_2 \triangleq \{e \mid (e \in S_1 \wedge e \in S_2)\}$$

It is not commonly used in the *VDM*. A constructive tail-recursive algorithm that may be used to compute the intersection of two sets is given by:

$$S_1 \cap S_2 \triangleq \cap[[S_1, S_2]]\emptyset$$

where

$$\cap: \mathcal{P}\Sigma \times \mathcal{P}\Sigma \longrightarrow \mathcal{P}\Sigma \longrightarrow \mathcal{P}\Sigma$$

$$\cap[[\emptyset, S]]T \triangleq T$$

$$\cap[[S_1, S_2]]T \triangleq$$

let  $e \in S_1$  in

$$\chi[[e]]S_2 \rightarrow \cap[[S_1 \setminus \{e\}, S_2]](T \uplus \{e\})$$

$$\rightarrow \cap[[S_1 \setminus \{e\}, S_2]]T$$

Tail-recursion is discussed in detail in Chapter 4. It is worth noting that in the recursive case, the set  $S_2$  is a read-only structure and the argument  $T$  accumulates

the result. To illustrate more clearly the construction process, I often prefer to regroup the arguments thus

$$\begin{aligned} \cap[\emptyset](S, T) &\triangleq (S, T) \\ \cap[S_1](S_2, T) &\triangleq \\ &\text{let } e \in S_1 \text{ in} \\ &\chi[e]S_2 \rightarrow \cap[S_1 \setminus \{e\}](S_2, T \uplus \{e\}) \\ &\rightarrow \cap[S_1 \setminus \{e\}](S_2, T) \end{aligned}$$

Rewriting the McCarthy conditional in the form

$$\chi[e]S_2 \left( \cap[S_1 \setminus \{e\}](S_2, T \uplus \{e\}), \cap[S_1 \setminus \{e\}](S_2, T) \right)$$

suggests the commutativity

$$\cap[S_1 \setminus \{e\}] \circ \chi[e]S_2 \left( (S_2, T \uplus \{e\}), (S_2, T) \right)$$

#### 2.1.5. Difference

I have already introduced the notion of set difference above. Formally, it may be defined by

$$\begin{aligned} - \setminus -: \mathcal{P}\Sigma \times \mathcal{P}\Sigma &\longrightarrow \mathcal{P}\Sigma \\ S_1 \setminus S_2 &\triangleq \{e \mid (e \in S_1 \wedge e \notin S_2)\} \end{aligned}$$

There are two variants of the notation for the difference operator in use in this work:  $S_1 - S_2$  and  $S_2 \leftarrow S_1$ . I generally prefer the latter notation where the difference operator may be interpreted as a monoid endomorphism, to be discussed later, in the respect that it is similar to the multiplication of natural numbers. Again, as is the case of the union operator, the difference operator is most frequently used for the purpose of set diminution, a typically occurring expression being one of the form:

$$S \setminus \{x\}, \quad \text{or} \quad \{x\} \leftarrow S$$

It should be clear how a constructive recursive definition may be formulated.

Returning again to the example of the dictionary, deletion of a word from the dictionary may be specified by

$$\begin{aligned} \text{Rem}: \text{WORD} &\longrightarrow \text{DICT} \longrightarrow \text{DICT} \\ \text{Rem}[[w]]\delta &\triangleq \chi[[w]]\delta(\{w\} \leftarrow \delta, \delta) \end{aligned}$$

where I have chosen to distinguish between the cases whether the word is or is not in the dictionary. A generalisation of the remove operation, in a fashion similar to

## AN OPERATOR CALCULUS FOR THE VDM

that of the enter operation, leads to a curried function that *is* the set difference of two dictionaries.

### 2.1.6. Symmetric Difference

The symmetric difference of two sets produces a set which is basically the union of the two sets less their intersection.

$$\begin{aligned} - \Delta - : \mathcal{P}\Sigma \times \mathcal{P}\Sigma &\longrightarrow \mathcal{P}\Sigma \\ S_1 \Delta S_2 &\triangleq (S_1 \cup S_2) \setminus (S_1 \cap S_2) \end{aligned}$$

or alternatively as

$$S_1 \Delta S_2 \triangleq (S_1 \setminus S_2) \cup (S_2 \setminus S_1)$$

which also may be written in the preferred form

$$S_1 \Delta S_2 \triangleq (S_2 \triangleleft S_1) \cup (S_1 \triangleleft S_2)$$

Interestingly, the symmetric difference operator permits two interpretations of the expression  $s \Delta \{e\}$ :

$$S \Delta \{e\} = \begin{cases} S \cup \{e\}, & \text{if } \chi_S(e); \\ S \setminus \{e\}, & \text{otherwise.} \end{cases}$$

If an element is not in the set then it is added to the set. If the element is already in the set then it is deleted from the set. The symmetric difference operator finds a natural rôle in the specification of a scan line polygon filling algorithm (see Salmon and Slater (1987, 429 *et seq.*)) and is discussed in full later:

$$\begin{aligned} \text{newjoinedlines} : \text{Scanline} \times \text{ActiveEtable} \times \mathcal{P}\text{PolyInd} &\longrightarrow \text{DC\_LineRec}^* \\ \text{newjoinedlines}(i, \langle e \rangle, ps) &\triangleq \Lambda \\ \text{newjoinedlines}(i, \langle e_1, e_2 \rangle \wedge t) &\triangleq \\ \text{let } e_1 = (-, x_l, -, p_i), e_2 = (-, x_r, -, -) &\text{ in} \\ \text{let } ps' = ps \Delta \{p_i\} &\text{ in} \\ \text{let } col = \text{minZcolour}(ps', (x_l, i)) &\text{ in} \\ \langle [(x_l, i), (x_r, i)], col \rangle \wedge \text{newjoinedlines}(i, \langle e_2 \rangle \wedge t, ps') & \end{aligned}$$

The algorithm is based on the fact that if a leading edge of a polygon is encountered in a sweep, then at some subsequent stage a trailing edge must also be encountered. Such edges must occur in pairs. (Edges which are parallel to the scan line are processed separately). In the specification of the algorithm, a set of polygon indices is maintained to keep track of edges during processing. Initially, the set is empty.

When a particular leading edge is encountered, that polygon index is added to the set. At some further point in the processing, it will be removed again when the trailing edge is encountered. Rather than use set union and set difference with an appropriate if-then-else construct (as is used in the text cited), the symmetric difference operator does the job more precisely and elegantly.

A tail-recursive definition for symmetric difference is given by:

$$\begin{aligned} \Delta [\emptyset](S, T) &\triangleq (S, T) \\ \Delta [S_1](S_2, T) &\triangleq \\ &\text{let } e \in S_1 \text{ in } \Delta [\{e\} \leftarrow S_1] \circ \chi[e]S_2 \left( (S_2, \{e\} \leftarrow T), (T \uplus \{e\}) \right) \end{aligned}$$

There is much more to the symmetric difference than at first might appear. Theoretically, it is probably the most interesting of all the set operators, giving us a ring. But, its significance for formal specifications in the operator calculus that I propose lies elsewhere. It generalises two very different set operators  $\leftarrow$  and  $\uplus$  to give a new one that has elegant mathematical properties. By employing the symmetric difference, a McCarthy conditional may be eliminated. There are practical cases where its use is justified, such as in the algorithm cited above. But it is a paradigm for the introduction of similar generalised operators as will be made clearer later on.

### 2.1.7. Cardinality

The cardinality of a set is the number of elements in the set. I give the definition in naïve recursive form:

$$\begin{aligned} \text{card}: \mathcal{P}\Sigma &\longrightarrow \mathbf{N} \\ \text{card}(\emptyset) &\triangleq 0 \\ \text{card}(S) &\triangleq \text{let } x \in S \text{ in } 1 + \text{card}(x \leftarrow S) \end{aligned}$$

I use the term ‘naïve’ whenever I wish to signal clearly that there is an obvious tail-recursive form for the same algorithm. In place of  $\text{card } S$ , I frequently prefer to use either  $\#S$  or  $|S|$ .

# AN OPERATOR CALCULUS FOR THE VDM

## 2.1.8. Distributed Union

The distributed union operator is a generalisation of the binary union operator  $\cup$  over a family of sets. The Danish School uses the keyword **union**, the English School prefers the conventional mathematical notation ( $\bigcup$ ), which has merit, and I prefer a reduction style operator ( $\cup/$ ) which I consider to be more expressive and of general applicability:

$$\begin{aligned} \cup/ : \mathcal{P}\mathcal{P}\Sigma &\longrightarrow \mathcal{P}\Sigma \\ \cup/\{S\} &\triangleq S \\ \cup/S_s &\triangleq \text{let } S, T \in S_s \text{ in } \cup/((S \cup T) \cup (\{S, T\} \triangleleft S_s)) \end{aligned}$$

For example, if  $S_s = \{S_1, S_2, \dots, S_j, \dots, S_n\}$  then

$$\cup/S_s = S_1 \cup S_2 \cup \dots \cup S_j \cup \dots \cup S_n$$

Note that the base case is  $\cup/\{S\} = S$ . For completeness, I may also define  $\cup/\emptyset = \emptyset$ . But from a constructive point of view it is not really necessary.

Every binary operation on sets, say  $- \otimes - : \mathcal{P}\Sigma \times \mathcal{P}\Sigma \longrightarrow \mathcal{P}\Sigma$ , may be extended to a reduction operator over a family of sets:

$$\otimes/ : \mathcal{P}\mathcal{P}\Sigma \longrightarrow \mathcal{P}\Sigma$$

Hence, both set intersection and set symmetric difference generalise to  $\cap/$  and  $\Delta/$ , respectively. Reduction is presented in more detail in Chapter 4.

The distributed union operator, which generalises set union, has introduced a shift in the domain of complexity. We have moved from sets, which are elements of some powerset domain,  $\mathcal{P}\Sigma$ , to a domain which contains families of sets,  $\mathcal{P}\mathcal{P}\Sigma$ .

2.2. Conceptual Expressiveness

What exactly is involved/implied when one chooses powerset to model some entity in the *VDM*? The construction of an abstract model, *MODEL*, using the powerset domain concept:

$$MODEL = \mathcal{P}\Sigma$$

is essentially given by the application of the powerset functor  $\mathcal{P}$  (MacLane and Birkhoff 1979, 153 *et seq.*):

$$\mathcal{P}: \Sigma \longrightarrow \mathcal{P}\Sigma$$

This categorical notion may be immediately extended to functions on sets. Therefore, consider a function  $f$  which maps a set  $\Sigma$  into  $T$

$$f: \Sigma \longrightarrow T$$

Then  $f$  may be extended to the powerset domain

$$\begin{array}{ccc} f: & \Sigma & \longrightarrow & T \\ & & & \downarrow \mathcal{P} \\ \mathcal{P}f: & \mathcal{P}\Sigma & \longrightarrow & \mathcal{P}T \end{array}$$

Here, it must be noted that  $\mathcal{P}f$  denotes a maplist type of function which applies the function  $f$  to each element of a set. The term ‘maplist’ is used trigger the connotation of the operation which is now called ‘mapcar’ in LISP. The distinct advantage of the use of  $\mathcal{P}X$  over the syntactic variants ‘set of  $X$ ’ and ‘ $X$ –set’ now becomes clear. Suppose that a dictionary  $\delta$  is partitioned as the set  $\{\delta_1, \delta_2, \dots, \delta_j, \dots, \delta_n\}$ , then the look up operation of a word  $w$  in the dictionary  $\delta$  is immediately given by the expression  $\vee / (\mathcal{P}\chi[[w]])\delta$ .

# AN OPERATOR CALCULUS FOR THE VDM

## 3. The Domain of Sequences

The sequence in *Meta-IV* is an abstraction that exhibits a duality akin to that of the electron in physics which was once treated as a particle and at other times as a wave, a duality that was confirmed in the Theory of Quantum Mechanics (Hawking 1988, 56). In the first instance, the sequence abstracts the notion of homogeneous list, a structure that is concretely realised in programming languages such as LISP and PROLOG. Operations such as head, tail and concatenation are naturally associated with the notion of sequence as list. But the sequence is also an abstraction of the notion of vector (in the mathematical sense) and 1-dimensional array, such as is found in programming languages like Pascal and Ada. Where we wish to make explicit the identification between sequence and, vector or array, then indexing (selection) may be an appropriate operator. However, there is no real dichotomy between the ‘list’ concept and the ‘vector’ concept of sequence. I claim that most algorithms on sequences, which are traditionally expressed using the index form, may be specified in an index-free form. Many examples of same occur throughout the thesis. The original *Meta-IV* term for a sequence was *tuple*, a convention that caused confusion when speaking about *Meta-IV* specifications in the context of relational data bases.

The user of the *Meta-IV* is invited to exhibit schizophrenic behaviour with respect to the sequence in its use in formal specifications, a significant issue from the point of conceptual models. Not only might one consider the sequence as both a list and a vector, but as will be made clear in the following section, the sequence is simply a particular form of a finite function (a map)! It is recommended, however, that such schizophrenic behaviour be minimised wherever possible in using the sequence for formal specification.

The sequence domain is the conceptual link by which the quantum jump from specification as encoding to specification as representation, referred to in Chapter 1, may be made. Any specification written using sequences may be transformed immediately into an encoding in LISP or PROLOG, for example, thus providing rapid feedback for the purpose of validation. With a little more effort, the same specification may be transformed into imperative encodings. Indeed, strictly speaking, every specification must ultimately be transformed into one which uses only sequences

and their operations for implementation on a computer *qua* Turing Machine. But therein lies a weakness as well as a strength. It is tempting to focus exclusively on the use of sequence for specifications precisely because there is a natural counterpart in programming languages. Whenever the *VDM* is used for reverse engineering from encoding to representation and the sequence appears to be the natural choice, then serious attention must be paid to analyse whether or not the concept of ordering plays a rôle in the problem domain. If not, then we may abstract further.

Since the concept of sequence does not appear strange to software engineers, I take this opportunity to explore the relationship of sequence with problem domains other than the customary ones traditional to software development. Consider the domain equation

$$WORD = CHAR^*$$

which specifies that a word is a (possibly empty) sequence of characters. It is important to note that *WORD* denotes the set of all possible words, some of which are:

$$\Lambda, \langle a \rangle, \langle b \rangle, \dots, \langle a, a \rangle, \langle a, b \rangle, \langle b, a \rangle, \dots$$

where *CHAR* denotes the underlying alphabet. Note that the *WORD* domain is precisely the structure one might use to model the ‘atoms’ of LISP or PROLOG. In the abstract, the alphabet will generally be denoted by  $\Sigma$  in this work. The star is taken from the classical theory of formal languages. Thus,  $\Sigma^*$  denotes all possible words over the alphabet  $\Sigma$ . To exclude the possibility of the empty word, the appropriate expression is  $\Sigma^+$ . In the theory of formal languages, it is customary to omit brackets. Thus, the above possible words may also be denoted by

$$\Lambda \text{ (or } \varepsilon), a, b, \dots, aa, ab, ba, \dots$$

Whereas domain equations of the above form may be used to capture the notion of vector, it is sometimes more appropriate to be explicit about the fact that the vectors in question are of fixed dimension  $n$ , in which case one might prefer the form:

$$VEC = ELEMENT^n$$

where it is understood that indexing of such vectors is in the range  $\{1 \dots n\}$ . In other words, the origin is fixed at 1, the usual *VDM Meta-IV* sequence indexing convention. But there are occasions when one wishes to index sequences with ori-

## AN OPERATOR CALCULUS FOR THE VDM

gin 0, as is the case in computer graphics, for example. This is a problem that frequently arises when dealing with vectors, one that was explicitly catered for in APL, permitting a user to set the origin at either 0 or 1. A more general concept of  $j$ -origin indexing in APL was introduced by Iverson (1962, 14). A similar strategy is adopted in the Irish School of the *VDM Meta-IV*.

### 3.1. Sequence Operations

The ‘basic’ operations on sequence are head ( $hd$ ), tail ( $tl$ ), concatenation ( $\wedge$ ), length ( $len$ ), selection ( $[ - ]$ ), elements of ( $elems$ ), indices of ( $inds$ ) and distributed concatenation ( $\wedge /$ ). Additional operations are membership ( $\chi$ ), reversal ( $rev$ ) and removal ( $\Leftarrow$ ). There is a real problem in distinguishing between ‘basic’ operations and ‘user-defined’ operations, a distinction which I abhor, lest I be dragged into some sort of foundational issue which I always wish to avoid. Such user-defined operations spring naturally from the use of sequences in encodings. For effective construction and computability, I am always willing to admit as ‘predefined’ some operation that has a precise recursive definition that is intuitively correct. Therefore, assuming that I or someone else has defined precisely a sort operation on sequences over some set or alphabet with an appropriate ordering relation, say quicksort, then I am prepared to admit sort as a ‘basic’ operation on sequences in subsequent specifications without any particular commitment that, indeed, the sort should be quicksort.

For the purpose of defining the operations on a sequence the following conventions are employed. The empty sequence is denoted by  $\Lambda$ . A sequence of one element  $e$  is denoted by  $\langle e \rangle$ . Indeed, there is a primitive *injection* function, of considerable theoretical significance elaborated upon in Chapter 3, denoted  $j: \Sigma \longrightarrow \Sigma^*$ , such that for all  $e \in \Sigma$ ,  $j: e \mapsto \langle e \rangle$ . I also have occasion to use a superscripted version  $j^n(e)$  that constructs a sequence of elements  $e$  of length  $n$ . Thus,  $j^4(1) = \langle 1, 1, 1, 1 \rangle$ . There is a primitive sequence constructor, denoted by  $\wedge$  which ‘adds’ a sequence of one element  $\langle e \rangle$  to an existing sequence  $\tau$  to give the new sequence  $\langle e \rangle \wedge \tau$ . In fact, the whole idea of a primitive sequence constructor will be abolished once the definition of concatenation has been given. Finally, a sequence may be defined implicitly by using the form  $\langle k \mid P(k) \rangle$ , where  $P$  is some predicate, analogously to that used for sets. Thus the sequence of the first 10 natural numbers,  $\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \rangle$ , may be denoted  $\langle j \mid 0 \leq j \leq 9 \rangle$ . Note carefully the convention that the predicate

$0 \leq j \leq 9$  is intended to denote the order in which the elements are to occur. This is similar to, say, the mathematical convention for sums:

$$\sum_{j=0}^n a_j x^j = a_0 + a_1 x + \dots + a_j x^j + \dots + a_n x^n$$

or  $\sum_{0 \leq j \leq n} a_j x^j$ , as used by Knuth (1973, 1:26). A predicate such as  $j \in \{0 \dots 9\}$  will not do. Similarly, the expression  $\langle \tau_j \mid \tau_j = \Lambda, 1 \leq j \leq n \rangle$  gives a sequence of empty sequences of length  $n$ . I do not care to be totally formal, provided the meaning is unambiguously understood.

### 3.1.1. Head and Tail

The first element in a non-empty sequence is given by the head operator, the rest of a non-empty sequence by the tail operator:

$$\begin{aligned} hd: \Sigma^+ &\longrightarrow \Sigma \\ hd(\langle e \rangle \wedge \tau) &\triangleq e \end{aligned}$$

and

$$\begin{aligned} tl: \Sigma^+ &\longrightarrow \Sigma^* \\ tl(\langle e \rangle \wedge \tau) &\triangleq \tau \end{aligned}$$

I avoid the use of  $hd$  and  $tl$  wherever possible in specifications, preferring to write either  $\langle e \rangle \wedge \tau$  or  $\langle e \mid \tau \rangle$ , depending on the context, and thus exhibiting directly the head element  $e$  and the tail  $\tau$ . It is important to draw attention to the signature of both operations. I use  $\Sigma^+$  and not  $\Sigma^*$ . The head and tail operations are not applicable to empty sequences. In other words, there is no question of having a partial function or the possibility of ‘error’ or ‘undefined’. In this respect, specification is different from programming. Indeed, in the Irish VDM the use of the structural forms referred to above eliminates any problem that might be supposed to exist. For example, a stack is adequately modelled by

$$STACK = X^*$$

Then the operations  $top$  and  $pop$  may be specified by

$$\begin{aligned} Top: STACK &\longrightarrow X \\ Top(\langle x \rangle \wedge \sigma) &\triangleq x \end{aligned}$$

and

## AN OPERATOR CALCULUS FOR THE VDM

$$\begin{aligned} Pop: STACK &\longrightarrow STACK \\ Pop(\langle x \rangle \wedge \sigma) &\triangleq \sigma \end{aligned}$$

The reliance on the *name* *STACK* used to cause problems for the algebraic axiomatic or equational treatment of *STACK* as an abstract data type in so far that the notion of ‘error’ or ‘undefined’ had to be treated formally within the theory, a problem that may be overcome, for example, by introducing the hidden sort of non-empty stack, *NeSTACK* (Goguen 1990, 18), where in the model-theoretic approach

$$NeSTACK = X^+$$

which is precisely in agreement with my definition of both the head and tail operations. The psychological (and practical) importance of refusing to admit the notions of ‘error’ or ‘undefined’ will be more fully developed later.

### 3.1.2. Concatenation

What has hitherto been used for the list constructor operation  $\wedge$  is in fact used generally for concatenation in the *Meta-IV*. However, in order to avoid confusion in defining the concatenation operator, the notation  $\oplus$  for concatenation (i.e., append) is used here only:

$$\begin{aligned} - \oplus -: \Sigma^* \times \Sigma^* &\longrightarrow \Sigma^* \\ \Lambda \oplus \sigma &\triangleq \sigma \\ (\langle e \rangle \wedge \tau) \oplus \sigma &\triangleq \langle e \rangle \wedge (\tau \oplus \sigma) \end{aligned}$$

Thus, concatenation is just an extension of the primitive list constructor operator. In formal language theory, concatenation is denoted by juxtaposition, the singleton sequence  $\langle e \rangle$  by the element  $e$  and the empty sequence by  $\varepsilon$  or  $\Lambda$ . Thus, the sequence  $\langle a, t, o, m \rangle = \langle a \rangle \wedge \langle t \rangle \wedge \langle o \rangle \wedge \langle m \rangle$  is conveniently denoted by *atom*.

### 3.1.3. Membership

A straightforward definition of the membership operation is immediately available in terms of that for set membership:

$$\chi_\tau(e) \triangleq \begin{cases} true & \text{if } e \in elems(\tau); \\ false & \text{otherwise.} \end{cases}$$

where *elems*  $\tau$  denotes the set of elements in  $\tau$ . Alternatively, a constructive definition, reminiscent of a typical Prolog predicate to determine list membership is

$$\begin{aligned} \chi[e]\Lambda &\triangleq \text{false} \\ \chi[e](\langle v \rangle \wedge \tau) &\triangleq \\ &v = e \rightarrow \text{true} \\ &\rightarrow \chi[e]\tau \end{aligned}$$

There is no notion of choice operator for a sequence as is the case for a set. However, knowing that a specific element  $x$  is in a non-empty sequence  $\sigma$ , or if one wishes to exhibit the structure of an non-empty sequence, then one may write

$$\text{let } \sigma = \sigma_l \wedge \langle x \rangle \wedge \sigma_r \text{ in } \dots$$

which provides a choice of possibly empty left,  $\sigma_l$ , and right,  $\sigma_r$ , subsequences.

### 3.1.4. Length

The length operator gives the number of elements in a sequence:

$$\begin{aligned} \text{len}: \Sigma^* &\longrightarrow \mathbf{N} \\ \text{len}(\Lambda) &\triangleq 0 \\ \text{len}(\langle e \rangle \wedge \tau) &\triangleq 1 + \text{len}(\tau) \end{aligned}$$

Again, as in the case of *card*, it is frequently preferable to employ  $\#\sigma$  or  $|\sigma|$  in lieu of  $\text{len } \sigma$ . I have given the naïve recursive definition which is equivalent to *len* as a *homomorphism of the free monoid*, an important notion developed fully in the next Chapter, there is a tail-recursive definition that exhibits interesting theoretical properties, as well as being highly efficient, to be discussed in Chapter 4.

### 3.1.5. Elements

The set of elements in a sequence is determined by:

$$\begin{aligned} \text{elems}: \Sigma^* &\longrightarrow \mathcal{P}\Sigma \\ \text{elems}(\Lambda) &\triangleq \emptyset \\ \text{elems}(\langle e \rangle \wedge \tau) &\triangleq \{e\} \cup \text{elems}(\tau) \end{aligned}$$

The operator *elems* is another homomorphism of the free monoid. Consider an alternative implicit definition, given by Mazurkiewicz in his development of Trace Theory (1987, 2:282): for each  $w \in A^*$ :

$$\text{elems}(w) \triangleq \{a \in A \mid \exists w_1, w_2 \in A^*: w = w_1 a w_2\}$$

where concatenation is denoted by juxtaposition. Mazurkiewicz actually denotes his operator by ‘Rng’ and refers to  $\text{Rng}(w)$  as the range of  $w$ . The same convention is

## AN OPERATOR CALCULUS FOR THE VDM

now used in the English School of the *VDM* (Jones 1986, 176), being justified by the fact that a sequence  $\sigma \in \Sigma$  is a map  $\mu \in \mathbf{N}_1 \xrightarrow{m} \Sigma$ . I claim that the first definition, which is based on a homomorphism is much clearer than the latter. By ‘clearer’, I mean to say that I can see at a glance what is meant by *elems*. Mazurkiewicz’s definition of the *elems* operator may be rewritten in the form

$$\begin{aligned} elems(w) &\triangleq \text{let } w = w_1 \wedge \langle a \rangle \wedge w_2 \text{ in } \{a\} \cup elems(w_1 \wedge w_2) \\ elems(\Lambda) &\triangleq \emptyset \end{aligned}$$

where use is made of the ‘choice’ operator on sequences mentioned earlier.

EXAMPLE 2.2. Consider the problem of stating, in formal notation, that a given sequence  $\sigma$  does not contain any duplicate elements. There are many ways in which this may be done. A particularly elegant solution is obtained by using a combination of both the *len* and *elems* operator. A sequence  $\sigma$  does not contain duplicates if the cardinality of the set of elements in the sequence is exactly equal to the length of the sequence:  $|elems \sigma| = |\sigma|$ .

There is another operator *items*, also a free monoid homomorphism, that is more general than *elems*. It maps a sequence into a bag or multiset and takes into account not only the occurrences of elements in a sequence but also the number of said occurrences. It will be introduced in the section on bags in Chapter 5.

### 3.1.6. Indices

Considering a sequence as an abstraction of a vector, it is useful to know the index range over which such a sequence ranges. The operator *inds* is provided for this purpose. In the following definition I assume an origin of 1:

$$\begin{aligned} inds: \Sigma^+ &\longrightarrow \mathbf{N}_1 \\ inds(\tau) &\triangleq \{1 \dots len(\tau)\} \end{aligned}$$

I rarely have occasion to use the *inds* operator. Instead, in the Irish School of the *VDM* I use the notion of a *slice* to designate an indexed range of a sequence. Such slices occur regularly in the transformation of an index-free recursive algorithm to an imperative algorithm using indexing. Briefly, a sequence  $\sigma$  of length  $n$ , may be regarded as the slice  $\sigma[1 \dots n]$ . A subsequence of length  $k - j + 1$  is denoted by  $\sigma[j \dots k]$  where the ordering is significant. In the event that  $j = k$  then  $\sigma[j \dots j]$  may be abbreviated to  $\sigma[j]$  and if  $j > k$  then we have the null sequence.

EXAMPLE 2.3. Consider the expression  $\sigma = \sigma_l \wedge \langle x \rangle \wedge \sigma_r$ , where  $|\sigma| = n$ ,  $n \geq 1$ . The corresponding expression in slice notation is  $\sigma[1 \dots n] = \sigma_l[1 \dots j - 1] \wedge \sigma[j] \wedge \sigma_r[j + 1 \dots n]$ .

The slice notation may be extended to sequences of sequences in a natural way. Let  $M \in (\Sigma^*)^*$  denote a model of an  $m \times n$  matrix. Then  $M[1 \dots m]$  is the slice expression corresponding to  $M = \langle r_1, r_2, \dots, r_i, \dots, r_m \rangle$ , where the  $i$ th row  $r_i$  may be denoted by the slice  $r_i[1 \dots n]$ .

### 3.1.7. Selection

It is possible to select the  $j$ th element of a non-empty sequence  $\tau$ , where  $j \in \text{inds}(\tau)$ . This is simply vector indexing whereby  $\tau_j$  denotes the element in question:

$$\begin{aligned} [-]: \Sigma^+ \times \mathbf{N}_1 &\longrightarrow \Sigma \\ (\langle e \rangle \wedge \tau)[1] &\triangleq e \\ (\langle e \rangle \wedge \tau)[j] &\triangleq \tau[j - 1] \end{aligned}$$

One ought *not* suppose that ‘selection by index’ is ‘implemented’ in this fashion. We are not dealing with ‘functional programming’. It *is*, of course, one way in which indexing ‘might’ be implemented, one which is rather inefficient!

The generalisation of the selection operator is precisely the slice notation that I have introduced above. In the case of the matrix example,  $M[i]$  denotes the  $i$ th row and, consequently,  $M[i][j] = M[i, j]$  denotes the element in the  $i$ th row and  $j$ th column of  $M$ ;  $M[i \dots i + 1]$  denotes the  $i$ th and  $i + 1$ st rows.

### 3.1.8. Reversal

Although not included as a basic *Meta-IV* operator, reversal seems to me to be of such importance that I include it here. The naïve recursive definition is straightforward:

$$\begin{aligned} \text{rev}: \Sigma^* &\longrightarrow \Sigma^* \\ \text{rev}(\Lambda) &\triangleq \Lambda \\ \text{rev}(\langle e \rangle \wedge \tau) &\triangleq \text{rev}(\tau) \wedge \langle e \rangle \end{aligned}$$

Following the conventions of formal language theory, I adopt the notation  $w^\ell$  to denote  $\text{rev } w$ . Given some alphabet  $\Sigma$ . Let  $\Sigma^{-1}$  denote the alphabet of formal ‘inverse’ elements where  $\sigma \in \Sigma$  has the formal inverse  $\sigma^{-1} \in \Sigma^{-1}$ . Define the annihilation

## AN OPERATOR CALCULUS FOR THE VDM

operator  $\odot$  on sequences over  $(\Sigma \cup \Sigma^{-1})^*$  to be the extension of concatenation in the following sense:

$$\begin{aligned}\langle e \rangle \odot \langle e^{-1} \rangle &= \Lambda = \langle e^{-1} \rangle \odot \langle e \rangle \\ \langle e \rangle \odot \langle f \rangle &= \langle e \rangle \wedge \langle f \rangle\end{aligned}$$

If we let  $w^{-1}$  denote the formal inverse of  $w$  where  $\forall i \in \text{inds } w, w^{-1}[i] = w[|w| - i + 1]^{-1}$ , then  $w \odot w^{-1} = \Lambda$ . Thus  $m^{-1}o^{-1}t^{-1}a^{-1}$  is the formal inverse of *atom*. A word  $w$  in  $(\Sigma \cup \Sigma^{-1})^*$  is said to be reduced if for each  $e$  in  $w$ , its adjacent element is not its formal inverse  $e^{-1}$ . Formally, if we denote by  $\mathcal{M}$  the set of reduced words over  $(\Sigma \cup \Sigma^{-1})^*$ , then  $(\mathcal{M}, \odot)$  is the free group of  $\Sigma$  (Papy 1964, 209).

This leads us to the famous ‘word problem’ for groups. The word problem for a group  $G$  is solvable if there exists a decision algorithm for the set of questions of the form: Does the word  $w$  represent the identity element in  $G$  (Rotman 1973, 278)? Rotman gives such a decision procedure for the free group (where  $w$  is not necessarily reduced), in a form that uses the ubiquitous ‘goto’, remarking that “the reader should experiment a bit with this program until he is convinced it is a decision process”. Both the admonition to the reader and the occurrence of a ‘goto’ algorithm in an otherwise elegant formal text on Group Theory has always struck me as being odd. This is undoubtedly due to the primitive conceptual model projected by ‘state of the art’ computing at the time the text was written. I give here an alternative decision algorithm, one that is more susceptible to intuition and reasoning.

$$\begin{aligned}\text{solvable}: (\Sigma \cup \Sigma^{-1})^* &\longrightarrow \{\mathbf{yes}, \mathbf{no}\} \\ \text{solvable}(w) &\stackrel{\Delta}{=} \text{let } w' = \text{reduce}(w) \text{ in } (w' = \Lambda \rightarrow \mathbf{yes}, \mathbf{no})\end{aligned}$$

where

$$\begin{aligned}\text{reduce}: (\Sigma \cup \Sigma^{-1})^* &\longrightarrow (\Sigma \cup \Sigma^{-1})^* \\ \text{reduce}(w) &\stackrel{\Delta}{=} \\ &\text{if } w = \Lambda \text{ then } \Lambda \\ &\text{else let } w' = \text{reduce}'[[w]]\Lambda \text{ in} \\ &\quad (w' = w) \rightarrow w', \text{reduce}(w')\end{aligned}$$

and  $\text{reduce}'$  is specified by

$$\begin{aligned}
 \text{reduce}' : (\Sigma \cup \Sigma^{-1})^* &\longrightarrow (\Sigma \cup \Sigma^{-1})^* \longrightarrow (\Sigma \cup \Sigma^{-1})^* \\
 \text{reduce}'[\langle x_1, x_2 \rangle \wedge \tau]w &\triangleq \\
 x_1 = x_2^{-1} &\rightarrow \text{reduce}'[\tau]w \\
 x_2 = x_1^{-1} &\rightarrow \text{reduce}'[\tau]w \\
 \top &\rightarrow \text{reduce}'[\langle x_2 \rangle \wedge \tau](w \wedge \langle x_1 \rangle) \\
 \text{reduce}'[\langle x_1 \rangle]w &\triangleq w \wedge \langle x_1 \rangle \\
 \text{reduce}'[\Lambda]w &\triangleq \Lambda
 \end{aligned}$$

To understand the preceding specification it is necessary to redo it. This method of achieving understanding is as true for formal specifications as it is for mathematics and APL programs. Indeed, ‘doing’ specifications is essentially a social activity. A final remark is worth making. To the practised *Meta-IV* developer, implementation of *reduce'* may be carried out using a stack *s* and a queue *q*:

$$\text{reduce}'[s]q \triangleq \dots$$

### 3.1.9. Removal

The definition of the removal operation presented here is one that removes all occurrences of an element from a sequence. The definition is constructive:

$$\begin{aligned}
 - \Leftarrow - : \mathcal{P}\Sigma \times \Sigma^* &\longrightarrow \Sigma^* \\
 \{e\} \Leftarrow \Lambda &\triangleq \Lambda \\
 \{e\} \Leftarrow (\langle v \rangle \wedge \tau) &\triangleq \\
 v = e &\rightarrow \{e\} \Leftarrow \tau \\
 &\rightarrow \langle v \rangle \wedge (\{e\} \Leftarrow \tau)
 \end{aligned}$$

I customarily use expressions of the form  $\{e\} \Leftarrow \tau$  to remove all occurrences of one element at a time. Now one might wonder why I insist on the removal operator deleting *all* occurrences and not just the first occurrence of an element. The reason is simple. Using the former notion permits me to identify the removal operator as an *endomorphism* of the free monoid (details in Chapter 3). The operator which deletes only the first occurrence of an element does have its uses but does not enjoy such a beautiful property. In addition, the removal operator as endomorphism is the natural counterpart to the set difference operator and there is another natural counterpart for maps.

# AN OPERATOR CALCULUS FOR THE VDM

## 3.1.10. Distributed Concatenation

Distributed concatenation of sequences of sequences, also known as ‘flattening’ in certain quarters, denoted **conc** by the Danish School, is similar to the distributed union of sets of sets. Here I denote distributed concatenation by the reduction  $\wedge /$ :

$$\begin{aligned} \wedge /: (\Sigma^*)^+ &\longrightarrow \Sigma^+ \\ \wedge / \langle \sigma \rangle &\triangleq \sigma \\ \wedge / \langle \sigma_1, \sigma_2 \mid \tau \rangle &\triangleq \wedge / \langle \sigma_1 \wedge \sigma_2 \mid \tau \rangle \end{aligned}$$

## 3.2. Conceptual Expressiveness

As in the case of the powerset domain, we have a similar categorical concept. The construction of an abstract model *MODEL* using the star operator:

$$MODEL = \Sigma^*$$

is essentially application of the star functor  $-^*$ :

$$-^*: \Sigma \longrightarrow \Sigma^*$$

Let  $f$  be a function which maps a set  $\Sigma$  into  $T$

$$f: \Sigma \longrightarrow T$$

This function may be extended to sequences of elements over  $\Sigma$ :

$$\begin{array}{ccc} f: & \Sigma & \longrightarrow & T \\ & & & \downarrow -^* \\ f^*: & \Sigma^* & \longrightarrow & T^* \end{array}$$

This *is* the classical ‘maplist’ function of LISP already referred to in the previous section on Domains of Sets. Again it is worth remarking on the expressiveness of the star functor for sequences compared to the inadequacy of ‘**seq of ...**’ used by the English School. To illustrate the use of star functor consider a simple example.

EXAMPLE 2.4. Let  $\sigma = \langle (a_1, b_1), (a_2, b_2), \dots, (a_j, b_j), \dots, (a_n, b_n) \rangle$  denote a sequence of ordered pairs. Then one may derive sequences  $\sigma_a = \langle a_1, a_2, \dots, a_j, \dots, a_n \rangle$  and  $\sigma_b = \langle b_1, b_2, \dots, b_j, \dots, b_n \rangle$  by mapping the appropriate projection functions associated with cartesian products,  $\sigma_a = \pi_1^* \sigma$  and  $\sigma_b = \pi_2^* \sigma$ , respectively, where  $\pi_1: (a, b) \mapsto a$  and  $\pi_2: (a, b) \mapsto b$ . Now suppose that we wish to ensure that in the sequence of ordered pairs  $\sigma$ , there is exactly one occurrence of each  $a_j$ . This may be specified succinctly by  $|elems \circ \pi_1^* \sigma| = |\sigma|$ .

## 4. Domain of Maps

Of all the *Meta-IV* domains used in formal specification, the map is the most useful and most frequently used. The map is basically intended to be an abstraction for a finite function or graph, the latter term being used in the mathematical sense and not to be confused with the structure of the same name used in computing. Naturally, it may also be used to represent arbitrary infinite functions. Conceptually, it may be viewed as an abstraction of the 1-dimensional array of programming languages, which in turn is a linguistic expression of a computer store. This duality of concept which is embraced by the term map finds concrete expression in, for example, (i) the syntactic similarity of the Ada notation for function application and array access, and (ii) the concept of a memo function (Field and Harrison 1988, 447).

Whereas I have already indicated that the powerset domain gives rise to conceptual difficulties for beginners, the map domain is even more of a problem. It is incumbent upon me, therefore, to discuss the concept at some length. Let  $X$  and  $Y$  denote two sets, then the set of all (total) functions on  $X$  to  $Y$ , the *function set*, is customarily denoted  $Y^X$  (Godement 1969, 56; Mac Lane and Birkhoff 1979, 154). There are  $|Y|^{|X|}$  such functions. On the other hand the set of all partial functions on  $X$  to  $Y$  may be denoted by  $PF(X, Y)$ , following the notational conventions of Eilenberg (1976). Naturally, the function set is a strict subset,  $Y^X \subset PF(X, Y)$ . The *VDM Meta-IV* notation embraces this notion of the set of all partial functions using the notation

$$M = X \xrightarrow{m} Y$$

To make this clear, I present a very simple example, adopted from Mac Lane and Birkhoff (1979).

EXAMPLE 2.5. Consider the two sets  $X = \mathbf{2}' = \{1, 2\}$  and  $Y = \{a, b\}$ . Then every function  $f: \mathbf{2}' \rightarrow Y$  is determined by the values  $f(1)$  and  $f(2)$  in  $Y$ . There are exactly four such functions:

$$\begin{array}{ll} f_1(1) = a & f_1(2) = a \\ f_2(1) = a & f_2(2) = b \\ f_3(1) = b & f_3(2) = a \\ f_4(1) = b & f_4(2) = b \end{array}$$

## AN OPERATOR CALCULUS FOR THE VDM

In the *Meta-IV* map notation these are denoted by

$$f_1 = [1 \mapsto a, 2 \mapsto a]$$

$$f_2 = [1 \mapsto a, 2 \mapsto b]$$

$$f_3 = [1 \mapsto b, 2 \mapsto a]$$

$$f_4 = [1 \mapsto b, 2 \mapsto b]$$

However, there are other elements of the domain  $X \xrightarrow{m} Y$  besides those in the function set  $Y^X$ . For example, there is the empty map  $\theta$  and the maps  $[1 \mapsto a]$ ,  $[2 \mapsto a]$ ,  $[1 \mapsto b]$ , and  $[2 \mapsto b]$ , giving a total of

$$\sum_{j=0}^{|X|} \binom{|X|}{j} |Y|^j = (|Y| + 1)^{|X|}$$

elements. Since the map  $[1 \mapsto a]$  is not defined for 2, then it is partial. The set of all possible maps is given by the distributed union of the set of all function sets  $Y^S$ , where  $S$  is an element of the power set of  $X$ . Formally,

$$\cup / \{Y^S \mid S \in \mathcal{P}X\}$$

This structure gives us a natural inclusion ordering of map, induced from that of the powerset domain, that fits in exactly with the notion of ‘approximates’ in Chapter 1. Specifically, I will write  $\mu_j \sqsubseteq \mu_k$  to indicate that the map  $\mu_j$  approximates the map  $\mu_k$ . Formally

$$\mu_j \sqsubseteq \mu_k \triangleq (dom \mu_j \subseteq dom \mu_k) \wedge (rng \mu_j \subseteq rng \mu_k)$$

It is not a coincidence that this leads naturally into the foundations of denotational semantics (Stoy 1980, 48; 1982, 80).

An alternative conceptual view of a map  $\mu$  may be obtained by considering it to be a set of ordered pairs  $\mu = \{(x, y) \mid x \in X, y \in Y\}$  subject to the restriction that if  $(x_j, y_j)$  and  $(x_k, y_k)$  are in  $\mu$ , then  $y_j = y_k \implies x_j = x_k$ . Thus, we have the strict inclusion

$$X \xrightarrow{m} Y \subset \mathcal{P}(X \times Y)$$

where  $\mathcal{P}(X \times Y)$  denotes the domain of all binary relations over  $X$  and  $Y$ . In practice, as I have mentioned, the map domain arises very frequently in specifications. Consider for example, the domain equation for a simple temperature chart:

$$CHART = CITY \xrightarrow{m} TEMPERATURE$$

Elements of this domain might be

$\theta$ ,  
 $[Dublin \mapsto 14, Cork \mapsto 16, Belfast \mapsto 11]$ ,  
 $[Galway \mapsto 21, Belfast \mapsto 18]$   
 $\dots$

The map is conceptually a table. The empty table is denoted  $\theta$  and

$$[Dublin \mapsto 14, Cork \mapsto 16, Belfast \mapsto 11]$$

might just as well be represented in the more conventional form:

<i>Dublin</i>	14
<i>Cork</i>	16
<i>Belfast</i>	11

It is frequently useful to distinguish the subset of injective maps from a set  $X$  into a set  $Y$ . Such a subset is customarily denoted  $X \xleftrightarrow{m} Y$ . The term ‘injective’ is used rather than ‘bijective’ since, in general, elements of the domain  $X \xleftrightarrow{m} Y$  are not surjective. This is due to the asymmetry in describing maps as elements of  $\cup/\{Y^S \mid S \in \mathcal{P}X\}$ . The subset of bijective maps are properly included only within the function set  $Y^X$ .

I have already shown that maps may be defined explicitly by enumerating the (domain, range) pairs. There is an implicit map constructor corresponding to that for sets and sequences. Thus, the successor function may be defined as the (infinite) map  $[n \mapsto n + 1 \mid n \in \mathbf{N}]$ .

#### 4.1. Map Operations

The map operations that I have chosen to be basic are extend ( $\cup$ ), override ( $+$ ), application ( $((-))$ ), restriction ( $\triangleleft$ ), removal ( $\triangleleft$ ), domain ( $dom$ ), range ( $rng$ ), membership ( $\chi$ ), symmetric difference ( $\Delta$ ), composition ( $\circ$ ), and distributed extend ( $\cup/$ ). Note that it is customary in the *VDM* to use the term merge for extend.

# AN OPERATOR CALCULUS FOR THE VDM

## 4.1.1. Application

Map application is simply function application:

$$-(-): (X \xrightarrow{m} Y) \times X \longrightarrow Y$$

$$\mu(x) \triangleq \chi[(x, -)]\mu \rightarrow y, \rightarrow \perp$$

where  $\perp$  has the same connotation given earlier. The characteristic function is that as defined for sets where I am treating the map  $\mu$  as a set of ordered pairs. Naturally, in the constructive approach we will normally want to ensure that  $x$  is in the domain of the map  $\mu$ , denoted  $x \in \text{dom } \mu$ . I certainly do not want to interpret  $\perp$  as ‘error’ or ‘undefined’.

## 4.1.2. Domain and Range

When speaking about a function  $f \in Y^X$ , it is customary to use the terms domain for  $X$  and range (or codomain) for  $Y$ . This terminology is also used for a *Meta-IV* map. Let  $\mu \in X \xrightarrow{m} Y$ . Then, in general  $\mu$  is a partial function with domain  $X$ . However, the *Meta-IV* operator  $\text{dom}$ , when applied to a map  $\mu$ , permits one to speak of  $\mu$  as a total function with domain  $\text{dom } \mu \subset X$ . Formally:

$$\text{dom}: (X \xrightarrow{m} Y) \longrightarrow \mathcal{P}X$$

$$\text{dom } \mu \triangleq \{x \mid (x, y) \in \mu\}$$

A constructive definition of the  $\text{dom}$  operator may be given recursively:

$$\text{dom}: (X \xrightarrow{m} Y) \longrightarrow \mathcal{P}X$$

$$\text{dom } \theta \triangleq \emptyset$$

$$\text{dom } \mu \triangleq \text{let } (x, y) \in \mu \text{ in } \{x\} \cup \text{dom}(\mu \setminus \{(x, y)\})$$

where I have chosen to consider the map as a set of ordered pairs.

The image of  $\text{dom } \mu$  under  $\mu$ , denoted  $\mu(\text{dom } \mu)$ , is the set of all  $y \in Y$  such that the image of  $x \in \text{dom } \mu$  is equal to  $y$ :

$$\mu(\text{dom } \mu) = \{y \mid y \in Y, x \in \text{dom } \mu, \mu(x) = y\}$$

In the *VDM Meta-IV*, the range operator  $\text{rng}$  constructs this subset of the range:

$$\text{rng } \mu \triangleq \mu(\text{dom } \mu)$$

The range operator  $\text{rng}$  may be defined analogously to that of the  $\text{dom}$  operator:

$$\text{rng}: (X \xrightarrow[m]{} Y) \longrightarrow \mathcal{P}Y$$

$$\text{rng } \mu \triangleq \{y \mid (x, y) \in \mu\}$$

Now I choose to give a constructive tail-recursive definition:

$$\text{rng}: (X \xrightarrow[m]{} Y) \longrightarrow \mathcal{P}Y \longrightarrow \mathcal{P}Y$$

$$\text{rng } [\theta]s \triangleq s$$

$$\text{rng } [\mu]s \triangleq \text{let } (x, y) \in \mu \text{ in } \text{rng } [\mu \setminus \{(x, y)\}](s \cup \{y\})$$

with  $\text{rng}(\mu) = \text{rng } [\mu]\emptyset$ . For all maps  $\mu \in X \xrightarrow[m]{} Y$ , it is always the case that

$$|\text{rng } \mu| \leq |\text{dom } \mu|$$

with strict equality if and only if  $\mu$  is 1–1, or injective.

#### 4.1.3. Inverse Image

Let  $\mu \in (X \xrightarrow[m]{} Y)$  be a given map, then, even though the inverse map may not exist, it is possible to consider the inverse image of an element  $y$  in  $Y$ , denoted  $\mu^{-1}(y)$ . Formally,

$$\mu^{-1}(y) \triangleq \{x \mid \mu(x) = y\}$$

This may be naturally extended to subsets of  $Y$ . Let  $T \in \mathcal{P}Y$ , then one might define  $\mu^{-1}(T)$  to be the set of all  $x \in X$  such that  $\mu(x) \in T$ . However, I find it more useful to define the extension in a slightly different manner. The inverse image of a set  $T$  is the distributed union of the set of inverse images of each element  $y \in T$ . Formally:

$$\mu^{-1}(T) \triangleq \cup / \{S \mid S = \mu^{-1}(y), y \in T\}$$

An equivalent operator form is

$$\mu^{-1}(T) \triangleq \cup / \circ \mathcal{P}\mu^{-1}(T)$$

Then, given a map  $\mu$ ,  $\mathcal{P}\mu^{-1} \circ \text{rng } \mu$  is a partition of  $\text{dom } \mu$ . In fact, I have almost sufficient technical material to say exactly what I mean by partition using the operator calculus notation. Let  $j$  denote the function  $j: \mathcal{P}X \longrightarrow (X \xrightarrow[m]{} \mathbf{N})$ , which injects a set  $S$  into a map,  $j: S \mapsto [S \mapsto |S|]$ . Then  $\mathcal{P}\mu^{-1} \circ \text{rng } \mu$  is a partition of  $\text{dom } \mu$ , if and only if

$$\Delta / \circ \mathcal{P}\mu^{-1} \circ \text{rng } \mu = \text{dom } \mu$$

$$|\oplus / \circ \mathcal{P}(j \circ \mu^{-1}) \circ \text{rng } \mu| = |\text{dom } \mu|$$

## AN OPERATOR CALCULUS FOR THE VDM

where the  $\oplus$  operator is an extended bag addition operator and  $|-|$  on the left hand side is a bag size operator. Incidentally, one may show that  $\mathcal{P}(j \circ \mu^{-1}) = \mathcal{P}j \circ \mathcal{P}\mu^{-1}$ .

### 4.1.4. Extend

The idea behind map extension is that an existing map could be augmented incrementally. A beautiful example of the usefulness of the concept is to be found in the determination of the least fixed point of recursive specifications where the notion of function as graph is employed (Schmidt 1986, 104 *et seq.*). A natural use for it, within the original ambit of the VDM, was in the construction of a symbol table (environment) in defining the semantics of declarative statements of programming languages. Thus, given a map  $\mu$  with domain  $dom \mu = S$ , then  $\mu$  may be extended by  $[x \mapsto y]$ , denoted  $\mu \cup [x \mapsto y]$ . But this only made conceptual sense if  $x \notin S$ . In general, a map  $\mu_1$  could be extended by another map  $\mu_2$  if and only if  $\mu_1$  and  $\mu_2$  had no domain elements in common, i.e.,  $dom \mu_1 \cap dom \mu_2 = \emptyset$ . Under this assumption, the extension of a map  $\mu_1$  by a map  $\mu_2$  can be defined by

$$\begin{aligned} - \cup -: (X_1 \xrightarrow{m} Y) \times (X_2 \xrightarrow{m} Y) &\longrightarrow ((X_1 \uplus X_2) \xrightarrow{m} Y) \\ \mu_1 \cup \mu_2 &\triangleq \{(x, y) \mid x \in (dom \mu_1 \Delta dom \mu_2) \wedge ((x, y) \in \mu_1 \vee (x, y) \in \mu_2)\} \end{aligned}$$

where, in the signature,  $X_1 \cap X_2 = \emptyset$ , and  $X_1 \uplus X_2$  denotes the disjoint union of  $X_1$  and  $X_2$ . Clearly the extend operator is commutative.

This particular definition of the extend operator causes conceptual problems for those who see it as set union. Indeed it *may* be interpreted as set union, but only in those situations where the intersection of the domains of the map arguments is empty. Thus, viewing  $\mu$  as a set, then  $\mu \cup \mu = \mu$ ; viewing  $\mu$  as a map  $\mu \cup \mu$  is undefined! Here, the *Meta-IV* makes a clear distinction between the conceptual model of *a priori* existence and constructability. To emphasise this distinction, the extend operator may only be used when it is defined. In other words, in formal specifications using the *Meta-IV* it is not admissible to obtain an undefined result. So great is the conceptual confusion with respect to the extend operator symbol  $\cup$ , that I have a strong urge to replace it with  $\uplus$ .

I have found the extend operator to be particularly useful in denoting a non-empty map. Considering the recursive definition of  $dom$  given earlier, it may be rewritten in the form:

$$\begin{aligned} \text{dom } \theta &\triangleq \emptyset \\ \text{dom}([x \mapsto y] \cup \mu) &\triangleq \{x\} \cup \text{dom } \mu \end{aligned}$$

This use of the extend operator is exactly analogous to the form I use for non-empty sequences.

#### 4.1.5. Removal

The inverse of extension is removal. Let  $\mu \cup [x \mapsto y]$  be a map. By definition  $x \notin \text{dom } \mu$ . Then  $\{x\} \Leftarrow (\mu \cup [x \mapsto y]) = \mu$ . It may be defined by

$$\begin{aligned} - \Leftarrow -: \mathcal{P}X \times (X \xrightarrow{m} Y) &\longrightarrow (X \xrightarrow{m} Y) \\ S \Leftarrow \mu &\triangleq \{(x, y) \mid x \in (\text{dom } \mu \setminus S) \wedge (x, y) \in \mu\} \end{aligned}$$

The notation used here is that of the English School of the *Meta-IV* and of  $\mathcal{Z}$ . The more conventional mathematical notation (adopted by the Danish School) is  $\mu \setminus S$ . As might be anticipated, the removal operator is the generalization spoken of earlier and is an endomorphism of maps to be dealt with in detail in the next Chapter.

#### 4.1.6. Restriction

Map restriction is complementary to map removal:

$$\begin{aligned} - \triangleleft -: \mathcal{P}X \times (X \xrightarrow{m} Y) &\longrightarrow (X \xrightarrow{m} Y) \\ S \triangleleft \mu &\triangleq \{(x, y) \mid x \in S \wedge (x, y) \in \mu\} \end{aligned}$$

The conventional mathematical notation for restriction is  $\mu \upharpoonright S$ . The complementary nature of the two operations is exhibited by the equation:

$$(S \Leftarrow \mu) \cup (S \triangleleft \mu) = \mu$$

which, of course, may also be expressed in the form:

$$(S \Leftarrow \mu) \Delta (S \triangleleft \mu) = \mu$$

where  $\Delta$  is a symmetric difference operator for maps defined below. Writing the restrict and remove operators in curried form gives us the pair of map partitioning functionals  $(\Leftarrow \llbracket S \rrbracket, \triangleleft \llbracket S \rrbracket)$  with respect to a fixed set  $S$ . In manuscripts I prefer an indexed form  $(\triangleleft_S, \Leftarrow_S)$ .

## AN OPERATOR CALCULUS FOR THE VDM

### 4.1.7. Override

Map override (also known as overwrite) is clearly a natural consequence of specifying the formal semantics of the assignment statement in programming languages. For, if  $\mu$  denotes the store, then the meaning of  $i := v$  was to be given by overriding  $\mu(i)$  by  $v$ . Strictly, it was to be understood that, of course, override was valid if  $i \in \text{dom } \mu$ . However, for generality, it is customary to embrace the extend operator within the semantics of the override operator. However, I advocate that it not be used in this manner as conceptual confusion frequently results. The override operator may be defined by

$$\begin{aligned}
 - + -: (X_1 \xrightarrow{m} Y) \times (X_2 \xrightarrow{m} Y) &\longrightarrow ((X_1 \cup X_2) \xrightarrow{m} Y) \\
 \mu_1 + \mu_2 &\stackrel{\Delta}{=} (\text{dom } \mu_2 \triangleleft \mu_1) \cup \mu_2
 \end{aligned}$$

Quite clearly, from the definition, the operator is non-commutative. The form of the definition suggests a new operator that is commutative—the symmetric difference of maps.

### 4.1.8. Symmetric Difference

Just as in the case of sets, it is useful to define a symmetric difference  $\mu_1 \Delta \mu_2$  of maps  $\mu_1, \mu_2$ . The definition given here is a conceptual generalisation of symmetric difference for sets and provides a nice symmetry to the definition of the extend operation:

$$\begin{aligned}
 - \Delta -: (X_1 \xrightarrow{m} Y) \times (X_2 \xrightarrow{m} Y) &\longrightarrow ((X_1 \Delta X_2) \xrightarrow{m} Y) \\
 \mu_1 \Delta \mu_2 &\stackrel{\Delta}{=} (\text{dom } \mu_2 \triangleleft \mu_1) \cup (\text{dom } \mu_1 \triangleleft \mu_2)
 \end{aligned}$$

In addition, wherever the extend operator could be validly employed in a map expression, then it can be replaced by the symmetric difference operator. But the symmetric difference operator is always defined. Thus, for example,  $\mu \Delta \mu = \theta$ .

#### 4.1.9. Membership

A characteristic function may be used to test for map membership:

$$\begin{aligned} \chi: (X \times Y) &\longrightarrow (X \xrightarrow{m} Y) \longrightarrow \mathbf{B} \\ \chi_\mu(x, y) &\stackrel{\Delta}{=} \\ &x \in \text{dom } \mu \rightarrow \text{true} \\ &\qquad \qquad \qquad \rightarrow \text{false} \end{aligned}$$

Note that the argument might just as well be written  $(x, -)$ . It is only the domain element  $x$  that is significant. Hence, I frequently write  $\chi_\mu(x)$  in place of that given.

As was the case for sets, a choice operator for maps is given by the usual choice operator for sets. This is a direct consequence that a map is a set of ordered pairs. Thus, to choose the pair  $(x, y)$  from the map  $\mu$ , one may write

$$\text{let } (x, y) \in \mu \text{ in } \dots$$

But as noted above, I generally avoid such expressions in favour of the operational form  $[x \mapsto y] \cup \mu$ .

#### 4.1.10. Composition

Let  $f \in Y^X$  and  $g \in Z^Y$ , be two functions, then the composition of  $f$  and  $g$ , denoted  $g \circ f$  and read as ‘ $g$  after  $f$ ’, is an element of  $Z^X$ . *Meta-IV* maps may be composed in a similar fashion.

DEFINITION 2.1. *Given two maps  $\mu_1 \in (X \xrightarrow{m} Y)$ , and  $\mu_2 \in (Y \xrightarrow{m} Z)$ , such that  $\text{rng } \mu_1 = \text{dom } \mu_2$ , then  $\mu_2 \circ \mu_1$  is the resulting map in  $(X \xrightarrow{m} Z)$ .*

Formally:

$$\begin{aligned} - \circ -: (X \xrightarrow{m} Y) \times (Y \xrightarrow{m} Z) &\longrightarrow (X \xrightarrow{m} Z) \\ \mu_2 \circ \mu_1 &\stackrel{\Delta}{=} \{(x, z) \mid x \in \text{dom } \mu_1 \wedge \mu_2(\mu_1(x)) = z\} \end{aligned}$$

The distinction between function and *Meta-IV* map must always be kept in mind. It is easy to make subtle errors. Consider an enthusiast who, given the following domain equations

$$\begin{aligned} \mathcal{F} &= X \xleftrightarrow{m} Y \\ \mathcal{G} &= Y \xleftrightarrow{m} \mathcal{P}Z \end{aligned}$$

## AN OPERATOR CALCULUS FOR THE VDM

intends to construct the composite map  $g \circ f \in X \xrightarrow[m]{\leftrightarrow} \mathcal{P}Z$ , where  $f \in \mathcal{F}$ , and  $g \in \mathcal{G}$ , always being careful to ensure that  $\text{rng } f = \text{dom } g$ . Now suppose that the same enthusiast specifies an operation to extend a map  $f$  in the following way, taking care to ensure that compositionality is conserved:

$$\begin{aligned} \text{Ent}: X &\longrightarrow (\mathcal{F} \times \mathcal{G}) \longrightarrow (\mathcal{F} \times \mathcal{G}) \\ \text{Ent}[[x]](f, g) &\triangleq \\ &\text{let } y \in (Y \setminus \text{rng } f) \text{ in } (f \cup [x \mapsto y], g \cup [y \mapsto \emptyset]) \end{aligned}$$

subject to the precondition that  $x \notin \text{dom } f$ . Unfortunately, after two successive *Ent* operations, one would have constructed a pair of maps

$$(f \cup [x_1 \mapsto y_1] \cup [y_1 \mapsto \emptyset], g \cup [x_2 \mapsto y_2] \cup [y_2 \mapsto \emptyset])$$

which gives the composition  $(g \circ f) \cup [x_1 \mapsto \emptyset] \cup [x_2 \mapsto \emptyset]$ , which is no longer a 1–1 map. The problem has arisen due the ubiquitous default  $\emptyset$  assignment. Exactly this problem arises in stage 1 of the file system case study (Bjørner and Jones 1982, 362), discussed in Appendix C.

### 4.1.11. Distributed Extend

The distributed extend operator is a generalisation of the binary extend operator and is the counterpart to distributed union for sets of sets and distributed concatenation for sequences of sequences. However, it is only applicable to sets of mutually pairwise disjoint maps.

$$\begin{aligned} \cup /: \mathcal{P}(X \xrightarrow[m]{\rightarrow} Y) &\longrightarrow (X \xrightarrow[m]{\rightarrow} Y) \\ \cup / \{\mu\} &\triangleq \mu \\ \cup / M &\triangleq \text{let } \mu_1, \mu_2 \in M \text{ in} \\ &\cup / \left( (\mu_1 \cup \mu_2) \cup \{\mu_1, \mu_2\} \leftarrow M \right) \end{aligned}$$

Note that the operator  $\cup$  denotes map extend and set union in that order from left to right.

## 4.2. Conceptual Expressiveness

Analogous to both the powerset functor,  $\mathcal{P}-$ , and the star functor,  $-^*$ , there is a corresponding map functor. The construction of an abstract model *MODEL* using the map operator:

$$MODEL = X \xrightarrow{m} Y$$

is essentially application of a functor  $(- \xrightarrow{m} -)$ :

$$(- \xrightarrow{m} -): X \times Y \longrightarrow (X \xrightarrow{m} Y)$$

Given a pair of functions  $f \in S^X$ , and  $g \in T^Y$  which map sets  $X$  and  $Y$  into  $S$  and  $T$ , respectively. Then, these functions may be extended to maps:

$$\begin{array}{ccc} f \times g: & X \times Y & \longrightarrow & S \times T \\ & & & \downarrow - \xrightarrow{m} - \\ f \xrightarrow{m} g: & X \xrightarrow{m} Y & \longrightarrow & S \xrightarrow{m} T \end{array}$$

In practice to avoid confusion, the function pair  $f \xrightarrow{m} g$  may be written  $(f, g)$ . Where a function  $f$  is to be applied to the domain of a map  $\mu$ , then one may simply write  $(f, -)\mu$ . Similarly, the application of  $g$  to the range of  $\mu$  is denoted  $(-, g)\mu$ .

# AN OPERATOR CALCULUS FOR THE VDM

## 5. Domain of Trees

In the introduction to the Chapter I mentioned that the *VDM Meta-IV* was originally designed to cope with the problem of specifying the denotational semantics of PL/1, with a view to constructing a compiler for same. Such origins have coloured the concepts embodied in the notation and method. In particular, the notion of abstract syntax of a programming language led directly to the domain of (parse) trees. Consider, once again, the assignment statement discussed earlier in the chapter. A particular concrete syntax in BNF form might look like:

$$\langle \text{assignment statement} \rangle ::= \langle \text{identifier} \rangle \text{ ':=' } \langle \text{expression} \rangle$$

where ‘:=’ is a terminal symbol in the grammar. Other choices of terminal symbol are always possible: ‘←’, or ‘=’. Even the production rule might look slightly different:

$$\langle \text{assignment statement} \rangle ::= \langle \text{expression} \rangle \text{ '}' \langle \text{identifier} \rangle$$

In all such cases it is possible to find a single homomorphic image in abstract syntax:

$$\textit{Assign} ::= \textit{Id} \times \textit{Exp}$$

Then typical assignment statements are constructed as  $mk\text{-Assign}(i, e)$  where ‘mk-’ denotes ‘make’ or ‘construct’. Conceptually, this expression is just a parse tree, a concept reinforced by comparing it with the standard way of expressing parse trees in the Definite Clause Grammar form of PROLOG:  $mk\_assign(I, E)$ , where  $mk\_assign$  is the name of the functor, a PROLOG concept not to be confused with the categorical concept of functor used earlier, and  $I, E$ , are variables denoting the yet to be instantiated parse trees for identifier  $I$  and expression  $E$ .

The tree domain is essentially a ‘tagged’ cartesian product, the convention being that, whenever the symbol  $::$  is employed, a symbol that probably owes its origins to the  $::=$  of the BNF, one may use the name of the domain prefixed with *mk-* as a tag. Dare I say it that the tag is a sort of ‘typing mechanism’? In practice, it was essential to use it to distinguish between two isomorphic pieces of abstract syntax. For example, consider the specification of abstract syntax for the creation and deletion of files in a simple file system

$$FS = Fn \xrightarrow{m} FILE$$

where the create and delete commands are given by

$$Crea :: Fn$$

$$Del :: Fn$$

Then without using tagging, it is not immediately clear whether the form  $(fn)$  is intended to be  $mk-Crea[fn]$  or  $mk-Del[fn]$ . To include a specific selector function in the definition of each piece of abstract syntax would have been ‘overkill’. To obtain an untagged cartesian product, one may use  $=$  in place of  $::$ . However, it seems to me that the distinction can appear a little too pedantic.

Although it was most frequently employed in specifying abstract syntax, the tree domain is an essential building block of constructive specifications, essential in the sense that the associated notation distinguishes the object. Consider for example, the following specification of the RGB colour model used in computer graphics:

$$RGB :: RED\_VAL \times GREEN\_VAL \times BLUE\_VAL$$

$$RED\_VAL = UNIT$$

$$GREEN\_VAL = UNIT$$

$$BLUE\_VAL = UNIT$$

$$UNIT = [0, 1] \quad \text{-- the closed interval of the real number line}$$

An object of type  $RGB$  can be constructed with the  $mk$ -operator. Thus, the colour ‘red’ is  $mk-RGB(1, 0, 0)$ . Now, given a cartesian product, one also needs to be able, on occasion, to select its components. There is the convention in the *Meta-IV* to specify such selector operations explicitly in the domain definition:

$$RGB :: s-red: RED\_VAL \quad s-green: GREEN\_VAL \quad s-blue: BLUE\_VAL$$

Note that, in this case the ‘ $\times$ ’ is omitted. Given an  $RGB$  colour  $c$ , then one can select the red component by writing  $s-red(c)$ .

This latter use of the tree domain of the *VDM* corresponds exactly to the record construct of imperative programming languages like Pascal and Ada. With respect to a conceptual model, both modes of use: for abstract syntax and for structured entities, are essentially the same. There is one other mode of use which must be singled out—that which binds together a collection of domains into a ‘state’. In very large complex specifications, subsystems are usually treated separately. Each subsystem may then be identified by a ‘local’ state. Construction of the entire

## AN OPERATOR CALCULUS FOR THE VDM

system specification is simply a matter of assembling the subsystems into a global state. Details of this particular mode of use are presented at length in Appendix C on the File System case study.

### 5.1. Projection

In practice, where the tree domain is used extensively in application areas such as computer graphics and constructive mathematics, it is notationally tedious to have to adhere to the convention of tagging cartesian products, where for precision, one wishes to retain the  $::$  symbol. For precisely this purpose I have introduced the ‘for *tagged tree* use *symbolic expression*’ construct, one which I have borrowed from Ada. Thus, for instance, in defining the abstract syntax of the assignment statement, I might wish to employ the suggestive form,  $:= (i, e)$ , in place of  $mk\text{-}Assign(i, e)$ . This is accomplished by writing:

for  $mk\text{-}Assign(i, e)$  use  $:= (i, e)$

More interestingly, I frequently prefer to ignore the tag completely. Thus, in the case of the *RGB* colour, I would write

for  $mk\text{-}RGB(r, g, b)$  use  $(r, g, b)$

In such cases, it is convenient to use standard mathematical projection functions  $\pi_j$  to select the  $j$ th component of an  $n$ -tuple, where  $1 \leq j \leq n$ . Recall that the term tuple was normally used for a sequence in the VDM. It is traditional in mathematics to employ it for an  $n$ -fold cartesian product,  $X^n$ , of a set  $X$ . Tuples in  $X^n$  are isomorphic to sequences (or vectors) of length  $n$ . In formal specifications the  $n$ -fold cartesian product is more likely to be of the form

$$X_1 \times X_2 \times \dots \times X_j \times \dots \times X_n$$

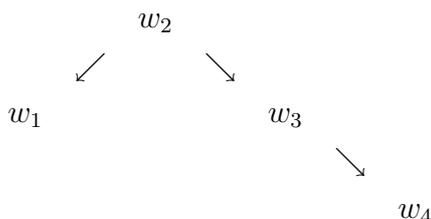
where the domains  $X_j$  are distinct.

5.2. Recursive Trees

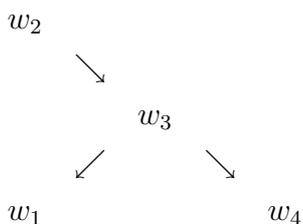
Whereas I have suggested that the notions of tree and cartesian product are interchangeable, it is not quite the full truth. In the *Meta-IV* the tree domain is the natural choice for recursive structures. Consider, for example, the formal specification of a binary tree which is intended to model a dictionary of words using the underlying notion of lexicographical ordering:

$$\begin{aligned} \text{BIN\_TREE} &:: [\text{NODE}] \\ \text{NODE} &= \text{BIN\_TREE} \times \text{WORD} \times \text{BIN\_TREE} \end{aligned}$$

where the expression  $[\text{NODE}]$  is used to abbreviate  $(\{\text{nil}\} \mid \text{NODE})$ . Then an empty binary tree is given by  $\text{mk-BIN\_TREE}(\text{nil})$ . Let  $w_1, w_2, w_3, w_4$  denote four words whose lexicographical order is given by the subscripts. Then entering the words in the sequences  $\langle w_2, w_1, w_3, w_4 \rangle, \langle w_2, w_3, w_1, w_4 \rangle$ , gives the trees



and



respectively. In formal tagged notation, these may be written as

$$\begin{aligned} &\text{mk-BIN\_TREE}(\text{mk-BIN\_TREE}(\text{mk-BIN\_TREE}(\text{nil}), \\ &\qquad\qquad\qquad w_1, \\ &\qquad\qquad\qquad \text{mk-BIN\_TREE}(\text{nil})), \\ &\qquad\qquad\qquad w_2, \\ &\qquad\qquad\qquad \text{mk-BIN\_TREE}(\text{mk-BIN\_TREE}(\text{nil}), \\ &\qquad\qquad\qquad\qquad\qquad\qquad w_3, \\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{mk-BIN\_TREE}(\text{mk-BIN\_TREE}(\text{nil}), \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad w_4, \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{mk-BIN\_TREE}(\text{nil}))) \end{aligned}$$

## AN OPERATOR CALCULUS FOR THE VDM

and

$$\begin{aligned}
 &mk\text{-}BIN\_TREE(mk\text{-}BIN\_TREE(nil), \\
 &\quad w_2, \\
 &\quad mk\text{-}BIN\_TREE( mk\text{-}BIN\_TREE(nil), \\
 &\quad\quad w_1, \\
 &\quad\quad mk\text{-}BIN\_TREE(nil)), \\
 &\quad\quad w_3, \\
 &\quad\quad mk\text{-}BIN\_TREE(nil), \\
 &\quad\quad w_4, \\
 &\quad\quad mk\text{-}BIN\_TREE(nil))
 \end{aligned}$$

In untagged form, the expressions are much like LISP:

$$((nil, w_1, nil), w_2, (nil, w_3, (nil, w_4, nil)))$$

and

$$(nil, w_2, ((nil, w_1, nil), w_3, (nil, w_4, nil)))$$

### 5.3. Operations

Operations on recursive trees are naturally given in recursive form. Let  $\beta$  denote a binary tree and  $w$  a word. Then inserting  $w$  into  $\beta$  may be specified by:

$$\begin{aligned}
 &Ins: WORD \longrightarrow BIN\_TREE \longrightarrow BIN\_TREE \\
 &Ins[[w]]\beta \triangleq \\
 &\quad \beta = mk\text{-}BIN\_TREE(nil) \\
 &\quad \rightarrow mk\text{-}BIN\_TREE(mk\text{-}BIN\_TREE(nil), w, mk\text{-}BIN\_TREE(nil)) \\
 &\quad \rightarrow \text{let } mk\text{-}BIN\_TREE(\beta_l, w', \beta_r) = \beta \text{ in} \\
 &\quad\quad \text{cases} \\
 &\quad\quad w < w' \rightarrow mk\text{-}BIN\_TREE(Ins[[w]]\beta_l, w', \beta_r) \\
 &\quad\quad w = w' \rightarrow \beta \\
 &\quad\quad w > w' \rightarrow mk\text{-}BIN\_TREE(\beta_l, w', Ins[[w]]\beta_r)
 \end{aligned}$$

Comparing this kind of specification with similar sorts of recursive algorithms in the literature, one notes that the only real difference is a matter of notation. Indeed, said algorithms are customarily given in some form of pseudo-code or actual programming language. The practitioner of the *VDM* must acquire the skills necessary to recognise the applicability of such algorithms to the problem in hand and translate them into the *Meta-IV*. This is a simple reusability issue—to avoid ‘reinventing the wheel’. Consider a simple instance. In 1980, Jean Vuillemin published

an important paper on *A Unifying Look at Data Structures* in the Communications of the ACM (Vuillemin 1980). The spirit of the work reported, which deals primarily with ‘cartesian trees’, is exactly analogous to that which I have attempted in this thesis. To illustrate what I mean by translation into the *Meta-IV*, I reproduce one of his algorithms, that which ‘cuts’ a cartesian tree. A node of the tree is a point  $p = (x, y)$ . The left subtree, node, and right subtree of a tree  $CT$  are selected using the projection functions  $l$ ,  $v$ , and  $r$ , respectively. An empty tree is denoted  $\emptyset$ , and a non-empty tree by  $\langle L, p, R \rangle$ . The algorithm is given in a form of ALGOL 60, a convention used at the time:

```

proc  $(L, R) \leftarrow \text{CUT}(CT, c)$ 
  Cartesian tree  $L, R, CT$ ; real  $c$ ; point  $p$ ;
  if  $CT = \emptyset$  then  $(L, R) \leftarrow (\emptyset, \emptyset)$ ;
    else  $p \leftarrow v(CT)$ ;
      if  $c < x(p)$  then  $(L, R) \leftarrow \text{CUT}(l(CT), c)$ ;
         $R \leftarrow \langle R, p, r(T) \rangle$ 
      else  $(L, R) \leftarrow \text{CUT}(r(CT), c)$ ;
         $L \leftarrow \langle l(T), p, L \rangle$ 
    fi
  fi
fproc CUT.

```

In translating this into *Meta-IV*, the principal difficulty is to understand Vuillemin’s notation. The notation that he uses for empty tree is, of course, the traditional empty set symbol. His notation for a non-empty tree uses the angle brackets, reserved in *Meta-IV* for sequences. The only other major problem is the occurrence of the symbol  $T$  which seems to denote a tree but is not declared. With the aid of some examples, and after hand-tracing the algorithm, the *Meta-IV* specification becomes apparent:

## AN OPERATOR CALCULUS FOR THE VDM

$$\begin{aligned}
 \text{CUT: } \mathbf{R} &\longrightarrow CT \longrightarrow CT^2 \\
 \text{CUT}[c]nil &\triangleq (nil, nil) \\
 \text{CUT}[c]mk-CT(L, p, R) &\triangleq \\
 &c < \pi_1 p \\
 &\rightarrow \text{let } (L', R') = \text{CUT}[c]L \text{ in} \\
 &\quad (L', mk-CT(R', p, R)) \\
 &\rightarrow \text{let } (L', R') = \text{CUT}[c]R \text{ in} \\
 &\quad (mk-CT(L, p, L'), R')
 \end{aligned}$$

where the domains in question are

$$\begin{aligned}
 CT &:: [NODE] \\
 NODE &:: CT \times POINT \times CT \\
 POINT &= \mathbf{R} \times \mathbf{R}
 \end{aligned}$$

This is a typical example of reverse-engineering important well-developed algorithms into the *Meta-IV*. The major distinction between the two forms of specification presented above is obvious—the notation of the *Meta-IV* specification does not depend on the whim of an individual author (*mea culpa?*) and every symbol is unambiguously defined. In addition there no ‘unexplained’ symbols in the formal text.

## 6. Summary

Anyone acquainted with the *VDM* will have no difficulty in grasping most of the material of this chapter. It deals with the basic *Meta-IV* domains: set, sequence, map, and tree. The re-presentation of this fundamental material has been necessary to exhibit something of the style of use of the Irish School of the *VDM*—in particular, the operator calculus which is at the heart of the method. One major facet is, of course, missing—the *method* itself, although cursory remarks on same have been made in the text.

Method is not just a style of use of a notation. It embraces a particular philosophy as much as a body of formal techniques. Crucial to the Irish method is the emphasis on fundamental algebraic structures: semigroups, monoids, semirings, etc., and their morphisms. Not only does the *VDM* lead naturally into a study of such algebra, but the very act of investigation of said algebra is essential for an understanding, both of the development method itself and the style of proof to be used in the method. The usual *VDM* techniques of the use of invariants and reification are part of the Irish *VDM* School. The concept of the *evolution* of a specification in contrast to reification is, I believe, unique to the Irish School. Traditionally, the invariant is interpreted as a sort of well-formedness condition or context condition for *Meta-IV* domains. In the Irish School, it plays a central rôle in determining what is meant by the very concept of proof with respect to formal specification.

A necessary corollary follows—the sort of mathematics that is essential for real computing is identified. Emphasis is needed on monoids rather than groups, for example, and hence on semirings rather than rings. This may appear to be a trivial observation. But it is not. Much of Group Theory is of no use whatsoever in computing. A group is a specialisation of a monoid—every group is *ipso facto* a monoid—and much of Group Theory depends on the notion of inverse element which, but for a few exceptions, has no rôle to play in computing. Monoids lies at the very centre of the Theory of Formal Languages and Finite Automata, the branch of computing which is probably the most mature. It should come as no surprise, therefore, to discover that a great wealth of mathematics already exists which could be of enormous benefit to those engaged in formal specification. To give but one simple example, the notion of *covering* in Automata Theory predates the use of

## AN OPERATOR CALCULUS FOR THE VDM

retrieve functions in the *VDM*.

Those using formal methods in the software industry have argued that some form of modularity is needed to control the complexity and size of the specifications. Emphasis is being placed on tool support. Much of the complexity arises from verbosity, arising from the recommendation to ‘use meaningful names’. Some emphasis ought also to be placed on the recommendation to ‘use meaningful mathematical structures’. Again, I feel that there is an over-emphasis on using formal logic in proofs and, consequently, there exists pressure to build more support tools. This is an unfortunate inheritance of this century. I will demonstrate that the style of proof actually used by mathematicians is adequate and sufficient for most formal specifications. For this I need to lay the foundation.

The next chapter deals with the essentials of the Theory of Monoids that I use. It is followed immediately by a full discussion of the Theory of Tail-Recursive forms and, with this introduction details of the method proper are given in Chapter 5.

# Chapter 3

## Monoids and Homomorphisms

### 1. Introduction

What is that one must know and what skills must one possess in order to be able to develop ‘good’ formal specifications that exhibit on the one hand mathematical rigour, and on the other reflect the real world of named entities? To what degree ought one to abstract from the concerns of implementation on a computer without losing touch with such concerns? I will argue that ‘to do’ formal specifications is ‘to do’ mathematics. But what sort of mathematics do I mean? I abhor the purely *formal* approach which seems to be prevalent in computing—an emphasis on predicate logic and theorem-proving. To prove theorems does not seem to me to be the essence of doing mathematics, an intuition which is happily reflected in the works of Pólya ([1962] 1981) and Lakatos (1976).

Now I do not wish to claim that proving theorems is unnecessary! This would be an incorrect reading of my position. In fact, one of the most important facets of doing formal specifications *is* theorem-proving; but here the terms theorem and proof and the process of theorem-proving have very particular meanings in the Irish School of the *VDM*, meanings which will be elaborated in due course. I insist that the real meat of formal specifications is in the discovery of said theorems, and theorems in formal specifications are statements about structure and algorithms over structure. Thus the primary goal is to exhibit structure. For the most part I mean algebraic structure. I would dearly like to exhibit a ‘geometry’ of formal specifications; this has eluded me so far. Proofs are often merely constructions: constructions of the structures and operations on the structures. Finding relationships between such structures is also interpreted as finding proofs. Finally, there is the ‘traditional’ notion of doing proofs; in the Irish School, this is to be interpreted as application of the operator calculus to discrete mathematical expressions. Having narrowed down

## MONOIDS AND HOMOMORPHISMS

the domain of discourse to algebraic structure, other distinctions must now be made.

There are algebraic theories that focus on abstract data types. Now it ought not to be supposed that the algebraic theory of abstract data types ought to be axiomatic or equational, a common misconception and one pointed out to me by Andrezej Blikle, I believe. In their desire to obtain the essential abstraction, researchers focus on properties which are then stated in terms of axioms or equations. Unfortunately, by so doing they are obliged to rely on names and may not resort to any one particular model. Indeed, they are then obliged to ensure that there is ‘no junk, no confusion’. Such reliance swings them away from ordinary mathematical structure into a world of term algebras. One might say that such algebraic theories have been built from the conceptual model of programming—from subroutines to procedures and functions, thence to modules and finally to abstract data types. I do not advocate disregard for the axiomatic approach. Instead, I assert that properties and axioms arise naturally through consideration of the algebraic structures to be built. My emphasis is always with due respect to the notion of a conceptual model.

There are process algebras for concurrency and parallelism. Researchers in this field focus on capturing behaviour. The algebra abstracts from the world of programming—the behaviour of processes. One gets the impression from the literature that such algebras are proposed as alternatives to the algebras of ‘static’ structures: Milner, for example, states quite categorically that “the ‘functional’ theory is no longer pertinent” for interaction and communication (1989, 2). It would have been better had he made remarks on the complementarity of the approaches. There is no denying the elegance of his work. But the very starting point of his theory—semantics of programs—has led him to make such observations.

There is the very abstract Category Theory. At least it has the merit of being essentially independent of programming concerns. It *is* employed to give meaning to computable objects and there is even the notion of categorical programming (Burstall and Rydeheard 1986). I do not believe that *starting* with category theory is the right conceptual approach to doing real formal specifications. It is simply too abstract. But that does not mean that one should not use category theory—some elementary essential aspects have already appeared in Chapter 2. The VDM domains of set, sequence and map were all tagged with the notion of functor precisely in order to give meaning to the notations for functions that ‘iterate’ over such

domains:  $\mathcal{P}f$ ,  $f^*$ , and  $f \xrightarrow[m]{}$   $g$ .

I propose a middle way—that which is based on the Theory of Monoids and Semigroups. The very notion of category is a generalisation of that of a monoid. The algebras that I consider have inherently nothing whatsoever to do with programming *per se*; and yet they are particularly apt for precisely that purpose. I will argue that they are ideally suited for the conceptual model of formal specifications and permit significant advances in computing compared with other approaches. For example, the natural numbers have been in existence for a long time before there was ever an axiomatic theory of the natural numbers, courtesy of Peano in his *Arithmetices principia nova methodo exposita*, 1889 (Boyer 1968, 645). Lack of such knowledge did not seem to inhibit mathematicians in Number Theory. Nor should it inhibit developers of formal specifications. It is nice to know that such an axiomatic theory exists. But it is quite disconcerting that researchers in computing find it necessary to incorporate it into the domain of formal specifications as has been done for example by the promoters of COLD-K, a formal design kernel language (Jonkers 1989, 147). Note that I do not draw a distinct line of separation between specification and design. Having provided the axiomatic theory, they then had to ensure that only the standard model  $\mathbf{N}$  was intended. To take for granted all the usual properties of the semiring  $(\mathbf{N}, +, \times, 0, 1)$ , a pair of monoids over  $\mathbf{N}$ , with the additional property that multiplication distributes over addition, seems a better conceptual approach.

In this Chapter, the material of which I view as a ‘two-edged sword’, I shall take great pains to delineate the sort of algebra which is necessary for a sound conceptual model of *VDM* and its applications. From *VDM* specifications one may ‘cut’ into algebra, bringing fresh insight and ‘problems to solve’ to a very old established body of mathematics. As a corollary, the very way one views the *VDM* is radically changed by the classical algebraic approach. I do not need to elaborate on the pedagogical conclusions that must inevitably be drawn.

# MONOIDS AND HOMOMORPHISMS

## 2. Monoids

The study of algebra may be said to be the study of structures and morphisms between structures. In mathematics, the theory of Groups plays a central rôle. In computing, on the other hand, the weaker theory of Monoids (and Semigroups) is more important. There is not much material on monoids in the literature on a par and level with that for groups. That which does exist is generally esoteric. The most useful work on the subject that I have found is Jacobson's *Basic Algebra I* (1974). In presenting the basic mathematical theory of monoids, I will draw freely on assumed knowledge of the basic sets of numbers.

Consider the set of natural numbers under addition. It is obvious that for all  $m, n$  in  $\mathbf{N}$ , then  $m + n$  is a natural number. Formally, we say that  $\mathbf{N}$  is closed under addition. We also take for granted that the order of bracketing in an expression of the form  $n_1 + n_2 + n_3$  is irrelevant. In other words,  $(n_1 + n_2) + n_3 = n_1 + (n_2 + n_3)$ . This is the associative law. Finally, there is a unique number in  $\mathbf{N}$ , the zero, such that  $n + 0 = n = 0 + n$ . Such an element is called an identity element. It just so happens that there exist many sets of elements, furnished with a binary operation, that satisfy these three properties of closure, associativity, and existence of unique identity element. Such structures are called monoids. Formally, we have

DEFINITION 3.1. *A monoid  $(M, \oplus, u)$  is a set  $M$ , together with a binary operator  $\oplus$ , and a distinguished element  $u$ , the identity element under  $\oplus$ , such that the following laws hold:*

1.  $M$  is closed under  $\oplus$ :

$$\forall m_1, m_2 \in M, m_1 \oplus m_2 \in M$$

2.  $\oplus$  is associative:

$$\forall m_1, m_2, m_3 \in M, (m_1 \oplus m_2) \oplus m_3 = m_1 \oplus (m_2 \oplus m_3)$$

3.  $u$  is the identity element for  $\oplus$ :

$$\forall m \in M, \exists! u \in M, m \oplus u = m = u \oplus m$$

Note that I have used the traditional *Meta-IV* expression,  $\exists! \dots$ , to represent 'there exists a unique ...'. Should it be the case that no such identity element can be found, then the structure in question is said to be a *semigroup*. In addition to the

three properties cited, the monoid  $(\mathbf{N}, +, 0)$  also enjoys another property, that of commutativity. In other words, for all natural numbers  $m, n$ , it is the case that  $m + n = n + m$ . We say that  $(\mathbf{N}, +, 0)$  is a commutative monoid.

The set of natural numbers is very large. One interesting question that naturally arises is whether there are subsets of  $\mathbf{N}$  that are closed under addition. Consider for example the set of even natural numbers:  $E = \{2n \mid n \in \mathbf{N}\}$ . Clearly the sum of two even numbers is even. Thus  $E$  is closed under addition. On the other hand, the set of odd natural numbers:  $O = \{2n + 1 \mid n \in \mathbf{N}\}$ , is not closed. A simple counter example is  $1 + 1 = 2 \notin O$ . This notion of closed subset of a monoid may be defined formally:

DEFINITION 3.2. *Let  $S$  be a subset of the monoid  $(M, \oplus, u)$ , i.e.,  $S \subset M$ . Then  $S$  is closed under  $\oplus$ , if  $\forall s_1, s_2 \in S, s_1 \oplus s_2 \in S$ .*

It is to be expected that a subset of a monoid, closed under the monoidal binary operation would itself be a monoid. Thus, for instance, the set of even natural numbers is indeed a monoid, denoted  $(E, +, 0)$ . However, closure under the monoidal binary operation is not sufficient. It is essential that the unique identity element also be contained in the set in question. An artificial counter-example is readily available. Consider the set  $E \setminus \{0\}$ , i.e., the set of even numbers excluding zero. Closure is immediately obvious. Equally obvious is the fact that  $(E \setminus \{0\}, +, 0)$  is not a monoid. Subsets of monoids which are closed under the monoidal binary operation and enjoy all the properties of monoids are called submonoids. Formally

DEFINITION 3.3. *A subset  $S$  of the monoid  $(M, \oplus, u)$  is a submonoid of  $M$  if  $S$  is closed under  $\oplus$  and is a monoid  $(S, \oplus, u)$ .*

Trivially, the monoid which consists only of the unique identity element,  $(\{u\}, \oplus, u)$ , and the entire monoid itself,  $(M, \oplus, u)$ , are submonoids of  $(M, \oplus, u)$ .

The same set may give rise to different monoids under different binary operations. The set of positive natural numbers under multiplication, denoted  $(\mathbf{N}_1, \times, 1)$  is a monoid. It may readily be verified that the set of odd numbers is a submonoid of  $(\mathbf{N}_1, \times, 1)$ .

Since  $\mathbf{Z}$  and  $\mathbf{Q}$  are groups under addition, then they are *a priori* additive monoids. A similar remark applies to the multiplicative group  $\mathbf{Q}_+ \setminus \{0\}$  of strictly positive rational numbers. We might continue in this vein exhibiting monoids on

## MONOIDS AND HOMOMORPHISMS

reals and complex numbers. It is now time to consider some of those monoids and semigroups which are natural to the *VDM Meta-IV*.

### 2.1. The Domain of Sets

The set data type of the VDM gives rise to the commutative monoids  $(\mathcal{P}S, \cup, \emptyset)$ ,  $(\mathcal{P}S, \cap, S)$  and  $(\mathcal{P}S, \Delta, \emptyset)$ . Indeed,  $(\mathcal{P}S, \Delta)$  is a group. The spelling-checker dictionary introduced in Chapter 2:

$$DICT_0 = \mathcal{P}WORD$$

is a monoid  $(DICT_0, \circ/Ent_0[-], \circ/Ent_0[\emptyset])$ , where the curried function

$$\circ/Ent_0[ws]\delta \triangleq \delta \cup ws$$

is the set union operator, a generalisation of the simple enter operation.

Similarly, the generalisation of the operation to remove a word from a dictionary gives the curried function  $\circ/Rem_0[ws]\delta \triangleq ws \triangleleft \delta$ , which is set difference. Thus, under the assumption that there are the appropriate preconditions for enter and remove, then the generalisations of both may be replaced by the symmetric difference operator and, consequently, we are operating in the monoid  $(DICT_0, \Delta, \emptyset)$ . Another interpretation of the remove operation—that of a monoid endomorphism—will be presented below.

### 2.2. The Domain of Sequences

The sequence data type of the VDM is the monoid  $(\Sigma^*, \wedge, \Lambda)$ , called the free monoid with base  $\Sigma$ . Given the non-empty set  $\Sigma$ , then every subset  $R$  of  $\Sigma$  gives rise to the monoid  $(R^*, \wedge, \Lambda)$ . Naturally,  $(R^*, \wedge, \Lambda)$  is a submonoid of  $(\Sigma^*, \wedge, \Lambda)$ . This is the structure underlying the theory of formal languages.

There are other operations one may define on sequences that give rise to monoids. Consider the domain of  $n$ -dimensional vectors over the real numbers:

$$\mathcal{V}_n = \mathbf{R}^n$$

Elements of  $\mathcal{V}_n$  are fixed length sequences. Let  $\vec{v} \in \mathcal{V}_n$  denote the sequence of elements  $\langle v_1, v_2, \dots, v_j, \dots, v_n \rangle$ . Define the addition of two such sequences  $\vec{u} + \vec{v}$  to be the sequence  $\langle u_1 + v_1, \dots, u_j + v_j, \dots, u_n + v_n \rangle$ . There is, of course, a unique element in  $\mathcal{V}_n$  which plays the rôle of the zero:  $\vec{0} = \langle v_j \mid v_j = 0 \wedge 1 \leq j \leq n \rangle$ . The structure  $(\mathcal{V}_n, +, \vec{0})$  is a commutative monoid of vectors. We may continue, if we so

wish, to develop the richer structure of vector space. But that is not the real focus here. Similarly, we may define a monoid of matrices:

$$\mathcal{M}_{mn} = (\mathcal{V}_n)^m$$

where  $\mathcal{V}_n$  is the domain of  $n$ -dimensional vectors defined above. A typical  $3 \times 2$  matrix is denoted  $\langle \langle a_{11}, a_{12} \rangle, \langle a_{21}, a_{22} \rangle, \langle a_{31}, a_{32} \rangle \rangle$ . The structure  $(\mathcal{M}_{mn}, +, \mathbf{0})$ , where addition is defined in the obvious way, is also a commutative monoid. Both of these models underpin computing with arrays. In this thesis, they are particularly relevant for formal specification in computer graphics and computer-aided design.

It might also be supposed that square matrices, over real numbers say, could be given monoidal structure under multiplication. This seems reasonable, at least conceptually. Let us define  $\mathcal{M}_2 \setminus \{\mathbf{0}\}$  to be the set of all  $2 \times 2$  matrices excluding the zero matrix,  $\mathbf{0}$ . I reason analogously to the way in which the natural numbers gave the multiplicative monoid  $(\mathbf{N}_1, \times, 1)$ . Assuming the usual matrix multiplication, we find that, unfortunately, it is not true. Consider the counter-example of the square matrices  $\langle \langle 1, 0 \rangle, \langle 0, 0 \rangle \rangle$  and  $\langle \langle 0, 1 \rangle, \langle 0, 0 \rangle \rangle$ . The product is the zero matrix and, thus, the multiplication is not closed. One may obtain such a multiplicative monoid by restricting the domain to non-singular matrices.

There is great significance in this result for the *VDM*. Firstly, using the terminology of the *VDM*, we may overcome our difficulty in constructing a multiplicative monoid of square matrices by specifying an *invariant* or well-formedness condition—that of non-singularity! Secondly, if such problems do arise with ‘simple’ algebraic structures such as matrices, then we must expect similar and more complex sorts of problems to arise in *VDM* specifications. Familiarity with such algebraic structures has had a profound influence on the development of the method of the Irish School.

The free monoid with base  $\Sigma$  is a model for concepts other than words and matrices. Consider, for example, the world of unbounded stacks and queues. The relevant operations that I wish to consider are shown in the following table:

<i>STACK</i>	<i>QUEUE</i>
$New \triangleq \Lambda$	$Crea \triangleq \Lambda$
$Push[e]\sigma \triangleq \langle e \rangle \wedge \sigma$	$Enq[e]\kappa \triangleq \kappa \wedge \langle e \rangle$
$Pop(\langle e \rangle \wedge s) \triangleq \sigma$	$Deq(\langle e \rangle \wedge \kappa) \triangleq \kappa$

Clearly, from the point of view of a monoid object, a stack and a queue are essentially the same. There is that one small difference as indicated by the push operation

## MONOIDS AND HOMOMORPHISMS

on stack and the enqueue operation on queue. Given the sequence of elements  $\langle e_1, e_2, e_3 \rangle$  as arguments to push and starting with a new stack produces the object  $s = \langle e_3, e_2, e_1 \rangle$ . In the case of a queue, one obtains  $q = \langle e_1, e_2, e_3 \rangle$ . Clearly, the queue  $q$  is the reverse of the stack  $s$ :  $q = s^\ell$ . Indeed, one is the mirror image of the other. We can construct such a mirror image in the following manner. Define anti-concatenation, denoted  $\wedge^\ell$ , on the set  $\Sigma^*$  as follows:

$$\sigma \wedge^\ell \tau \triangleq \tau \wedge \sigma$$

Then, it is trivial to show that  $(\Sigma^*, \wedge^\ell, \Lambda)$  is a non-commutative free monoid, the mirror image of  $(\Sigma^*, \wedge, \Lambda)$ . The conceptual significance of this result is that a stack in one space *is* a queue in the other, and vice versa! But one ought not to conclude that the concept of stack may be replaced by that of queue. For, given  $s = \langle e_2, e_1 \rangle$  and  $q = s^\ell$ , then  $\text{deq}(q) = \langle e_2 \rangle \neq \langle e_1 \rangle = (\text{pop}(s))^\ell$ . There is an algebra of stacks that is different from an algebra of queues. Needless to remark, it is just this sort of analysis that leads into term algebras and lays a foundation for the study of the axiomatic approach to abstract data types. I conclude this elementary introduction on the sequence monoids by giving two particularly distinguished examples:

EXAMPLE 3.1. Let  $\Sigma_!^*$  denote the set of all unique sequences, i.e., those which do not contain duplicate elements. Define the unique concatenation operator  $\diamond$  as follows:

$$\langle e \rangle \wedge \sigma \diamond \tau = \begin{cases} \sigma \diamond \tau, & \text{if } e \in \tau; \\ \langle e \rangle \wedge (\sigma \diamond \tau), & \text{otherwise.} \end{cases}$$

$$\Lambda \diamond \sigma = \sigma = \sigma \diamond \Lambda$$

Then  $(\Sigma_!^*, \diamond, \Lambda)$  is a non-commutative monoid which plays an important rôle in the implementation of sets as sequences, especially in languages such as PROLOG and LISP. We have ‘proved’ that unique sequences may be combined to give unique sequences. Set operations such as union and intersection are implementable in such a structure. The fact that it is non-commutative, signals the problem that implementations of union and intersection of sets in this structure do not commute.

EXAMPLE 3.2. Let  $\Sigma_{\leq}^*$  denote the set of all sequences whose elements are sorted in ascending order ( $\leq$ ). Define the merge operator  $\nabla$  as follows:

$$\langle e \rangle \wedge \sigma \nabla \langle f \rangle \wedge \tau = \begin{cases} \langle e \rangle \wedge (\sigma \nabla (\langle f \rangle \wedge \tau)), & \text{if } e \leq f; \\ \langle f \rangle \wedge ((\langle e \rangle \wedge \sigma) \nabla \tau), & \text{otherwise} \end{cases}$$

$$\Lambda \nabla \sigma = \sigma = \sigma \nabla \Lambda$$

Then  $(\Sigma_{\geq}^*, \nabla, \Lambda)$  is a commutative monoid. We have basically ‘proved’ that a merge sort of sorted sequences produces a sorted sequence.

### 2.3. The Domain of Maps

The map domain of the *VDM* is a rich source of monoids of a great variety. Let  $X \xrightarrow{m} Y$  denote a map over arbitrary domains  $X$  and  $Y$ . Then the structure  $(X \xrightarrow{m} Y, +, \theta)$  is a non-commutative monoid, where  $+$  denotes the override (i.e., overwrite) operator. It is interesting to note that this structure models the semantics of assignment, where  $X \xrightarrow{m} Y$  is given the interpretation  $STORE = Id \xrightarrow{m} VAL$ . Submonoids are readily obtained by restricting the domain. For example, let  $S \subset X$ . Then  $(S \xrightarrow{m} Y, +, \theta)$  is a submonoid of  $(X \xrightarrow{m} Y, +, \theta)$ . A similar result may be obtained by considering the complement of a set  $S$  with respect to  $X$ . These submonoids may also be obtained in a related manner by applying the map restriction and removal operators respectively, with respect to a *fixed set*. In this interpretation, the operators figure as monoid endomorphisms, discussed later.

Now, considering the map extend operator  $\cup$ , since  $\forall m \in (X \xrightarrow{m} Y)$ ,  $m \cup m$  is undefined, then it is not possible to use ‘extend’ to impose a monoidal structure on maps. But, looking ahead to Chapter 5, there are suitable generalisations that do indeed give structures that are monoids:

**EXAMPLE 3.3.** Let  $(\Sigma \xrightarrow{m} \mathbf{N}_1)$  denote a domain of bags or multisets. There is a generalisation of both ‘override’ and ‘extend’, denoted  $\oplus$ , which inherits the properties of addition of the natural numbers, for which  $(\Sigma \xrightarrow{m} \mathbf{N}_1, \oplus, u)$  is a monoid. But what exactly is the unique identity element  $u$ ? It certainly can not be of the form  $[e \mapsto 0]$ , since  $0 \notin \mathbf{N}_1$ . By convention, the appropriate candidate is the null bag  $\theta$ .

**EXAMPLE 3.4.** Similarly, when we come to consider the domain  $(\Sigma \xrightarrow{m} \mathcal{P}X)$  which models, among other things, a bill of materials, a dictionary, etc., then we can define an operation  $\oplus$  which is a generalisation of both extend and override that inherits the properties of set union. Such structures are also commutative monoids  $(\Sigma \xrightarrow{m} \mathcal{P}X, \oplus, u)$ , where  $u = \theta$ .

Finally, domains of the form  $(\Sigma \xrightarrow{m} X^*)$  give non-commutative monoids where again the  $\oplus$  operation inherits the properties of sequence concatenation. In practice, such

## MONOIDS AND HOMOMORPHISMS

monoids frequently appear in the guise of hash table specifications where collisions are handled by overflow chaining. To complete this introduction, it behoves me to refer to one other significant aspect of the *VDM* map domain which has a natural correspondence in algebra—composition of functions.

EXAMPLE 3.5. Since  $M = X \xrightarrow{m} Y = PF(X, Y)$  denotes the set of all partial functions from  $X$  to  $Y$ , and the function set  $Y^X$  is a subset of  $PF(X, Y)$ , then it follows immediately that  $(Y^X, \circ, \mathcal{I})$ , where  $\mathcal{I}$  denotes the identity function, is a monoid of functions under composition.

However, in formal specifications, it is rare to use the map to denote total functions. Instead, due to the constructive nature of specifications, the map most often appears as a partial function and in this rôle, the issue of compositionality, defined in Chapter 2, needs careful consideration. In the case that both the domain and the range are the same, say  $Q$ , then the resulting monoid, which is well known in Automata Theory, is called a transformation monoid (Eilenberg 1976). (It was through consideration of transformation *groups*, that the concept of abstract group was born (Jacobson 1974, 69)).

### 2.4. Other Algebraic Structures

I have briefly indicated that underlying the *VDM Meta-IV* domains, there is a world of monoids. In the next section I will demonstrate why it is important that those who develop formal specifications in *Meta-IV* ought to be familiar with such concepts. I have also intimated that there is more to formal specifications than the theory of monoids. We also need the weaker theory of semigroups. For example, in the domain of sequences, there are many operations for which the notion of unique identity is irrelevant and unnatural. It does not make sense to take the head or tail of an empty sequence—this is equivalent to asking for the top element of an empty stack and trying to pop an empty stack. One should not try to compute the minimum or maximum element of an empty sequence, etc. Such operations ‘live’ in the domain of non-empty sequences, which form semigroups under suitable binary operations. I again hasten to emphasise the important conceptual distinction between my approach and that customarily found in the computing literature with respect to the theory of abstract data types, say. In the theory of stacks, for example,

one is dealing with term algebras. It then becomes problematic when one has to confront an expression such as ‘top(new)’. A solution to the problem is to introduce the concept of ‘undefined’ or ‘error’ (stack). I take an entirely different view in my algebraic approach. For me such errors can not possibly exist at the level of formal specifications. The top operation is a semigroup operator, not a monoid operator. It is only in the evolutionary path towards ‘implementation’ that such concerns arise. I also have a more serious concern. Data structures and their operations which are mature in computing readily lend themselves to the abstract data type treatment. But in the real world of formal specifications, where the domains are complex and the necessarily incomplete set of operations is determined by end-user requirements, then the abstract data type approach, i.e., term algebra approach, seems totally irrelevant. It is for this reason that I have chosen to call on the assistance of classical algebraic structures in building the method of the Irish School.

As well as semigroups, we also need stronger theories. I have already alluded to vector spaces above. Other mathematical structures will be introduced as and when needed. The whole point of such a mathematical framework is to provide a sound operator calculus for formal specifications that obviates the need to ‘think all the time about what one is doing’ and the need to ‘prove’ everything formally.

# MONOIDS AND HOMOMORPHISMS

## 3. Monoid Homomorphisms

In the software industry there is an enormous amount of effort being expended in the construction of reusable components. Perhaps the greatest effort is being expended with respect to the Ada programming language and rather successfully. I believe that at least as much effort must also be applied to the more promising concept of reusable specifications. Where emphasis is being placed on modularity and tool support for specifications, as in the case of the BSI/*VDM* and RAISE, then one can not expect much more than is already being achieved for reusable software components. The concept of reusability, as applied to the domain of specifications, is very well understood in mathematics, where it has been fruitfully employed for over a century. The study of reusability is nothing more than the study of relationships between structures and the key relationship is the homomorphism. In the *VDM*, a key concept of the method is reification of abstract models to concrete models. The relationship of the latter to the former is also based on the homomorphism.

A homomorphism is essentially a mapping from one structure to another of the same kind. It conserves structure. The etymology of the term reflects this meaning: from the Greek *ὅμοιος*—similar, alike, same, and *μορφή*—form or shape. Thus one may speak about semigroup homomorphisms, monoid homomorphisms, etc. Perhaps the most famous homomorphism of all is that of logarithms that exhibited the structural similarity between two groups of real numbers, one a multiplicative group of strictly positive real numbers  $\mathbf{R}_+ \setminus \{0\}$ , the other an additive group of reals  $\mathbf{R}$ . We may express this homomorphism by the following commuting diagram:

$$\begin{array}{ccc} \mathbf{R}^+ \setminus \{0\} \times \mathbf{R}^+ \setminus \{0\} & \xrightarrow{\times} & \mathbf{R}^+ \setminus \{0\} \\ \downarrow \log \times \log & & \downarrow \log \\ \mathbf{R} \times \mathbf{R} & \xrightarrow{+} & \mathbf{R} \end{array}$$

Essentially, one is able to perform multiplication on the real numbers  $x$ ,  $y$ , by first mapping them to their equivalents  $\log(x)$ ,  $\log(y)$ , performing the addition  $\log(x) + \log(y)$ , and then applying an inverse mapping (the exponential function, or antilogarithm) to give the required result  $xy$ . Formally,

$$\log xy = \log x + \log y$$

Also, in addition,  $\log$  maps the identity element of  $(\mathbf{R}^+ \setminus \{0\}, \times)$  to the identity element of  $(\mathbf{R}, +)$

$$\log 1 = 0$$

The  $\log$  function, of which the slide rule is a mechanical manifestation, essentially maps one mathematical structure (a group) into another (a group). Put in other words, the  $\log$  function lays bare the relationship between structures. This behaviour of the logarithm function was an exciting discovery. However, not all functions share this property. For example, there is no operation  $\oplus$  such that

$$\sin xy = \sin x \oplus \sin y$$

As will now be demonstrated, the basic structures of the *VDM Meta-IV* are related via certain homomorphisms given by basic operations. First, the technical definition of a monoid homomorphism is required.

DEFINITION 3.4. *A monoid homomorphism  $h$  from the monoid  $(M, \oplus, u)$  to some other monoid  $(P, \otimes, v)$  is a map  $h: M \longrightarrow P$  such that*

$$h(m_1 \oplus m_2) = h(m_1) \otimes h(m_2)$$

$$h(u) = v$$

Again, it is customary to exhibit such a structural relationship via a commuting diagram:

$$\begin{array}{ccc} M \times M & \xrightarrow{\oplus} & M \\ \downarrow h \times h & & \downarrow h \\ P \times P & \xrightarrow{\otimes} & P \end{array}$$

There are different kinds of homomorphisms, and correspondingly special names. The logarithm homomorphism is especially remarkable in so far that it sets up an exact one-to-one correspondence between products and sums. The corresponding function is invertible. Such a homomorphism is called an *isomorphism*. Similarly, the function  $f: \mathbf{N} \longrightarrow E$ , defined by  $f: n \mapsto 2n$ , which maps the monoid of natural numbers to the monoid of even numbers is an isomorphism. However, if we consider the same function as a homomorphism from  $\mathbf{N}$  to itself, then it is called a *monomorphism*. It is one-to-one. But there are elements in  $\mathbf{N}$  which are not in the range of the map—all of the odd numbers in fact. When we consider modulo arithmetic, important for hash tables in computing for example, we meet an *epimorphism*. Let  $p$  be some prime. Define the function  $f: \mathbf{N} \longrightarrow \mathbf{Z}_p$ , by  $f: n \mapsto n \bmod p$ , where  $\mathbf{Z}_p$  is

## MONOIDS AND HOMOMORPHISMS

the additive group of natural numbers  $\{0 \dots p - 1\}$ . Many elements in  $\mathbf{N}$  get ‘glued together’ under  $f$ —a happy phrase often used by Andrej Blickle. It is not possible to tell from  $j \in \mathbf{Z}_p$  which of the infinite number of elements in  $\mathbf{N}$  maps to  $j$ . We may restate this more formally. First a few auxillary definitions are given.

DEFINITION 3.5. *An injective homomorphism is called a monomorphism. A surjective homomorphism is called an epimorphism. A homomorphism which is both injective and surjective (i.e., which is bijective) is called an isomorphism.*

To illustrate these notions in the context of VDM specifications, let us look at the spelling-checker dictionary considered at length in Appendix A. The most abstract model may be viewed as the monoid  $(\text{DICT}_0 = \text{PWORD}, \cup, \emptyset)$ . A simple reification is the monoid  $(\text{DICT}_2 = \text{WORD}^*, \wedge, \Lambda)$ , i.e., where we implement sets using sequences. Then the retrieve function is the monoid epimorphism *elems*, which is a basic sequence operator.

In the File System case study, details of which are to be found in Chapter 5 and in Appendix C, I had great difficulty in applying my operator calculus to the first level of reification  $FS_1$ . Specifically, I could not find an elegant retrieve function. The problem was resolved by identifying omissions in the original abstract specification  $FS_0$ , constructing a new reification  $FS'_1$ , which was inserted between  $FS_0$  and  $FS_1$ , showing that the latter was isomorphic to  $FS'_1$  and then constructing an epimorphism from  $FS'_1$  to  $FS_0$ .

There is even more nomenclature. Where there is only the one monoid in question that is both the domain and range of the map then the homomorphism acquires other names.

DEFINITION 3.6. *Let  $h$  be a monoid homomorphism from  $(M, \oplus, u)$  onto itself. Then  $h$  is called an endomorphism. If  $h$  is an isomorphism, then it is called an automorphism.*

Remove or delete operations are usually monoid endomorphisms. The notions of image and inverse image of maps may be extended to homomorphisms in a natural way.

## Monoid Homomorphisms

DEFINITION 3.7. Let  $h: (M, \oplus, u) \longrightarrow (P, \otimes, v)$  be a homomorphism of monoids. The image of  $M$  under  $h$ , denoted  $h(M)$  or  $\text{Im } h$  is defined to be the set of all  $y$  in  $P$  such that  $h(x) = y$ ,  $x \in M$ .

Formally

$$hM = \{h(x) \mid x \in M\}$$

DEFINITION 3.8. Let  $h: (M, \oplus, u) \longrightarrow (P, \otimes, v)$  be a homomorphism of monoids. The inverse image of  $P$  under  $h$ , denoted  $h^{-1}P$  is defined to be the set of all elements  $x$  in  $M$  such that  $h(x) \in P$ .

Formally

$$h^{-1}P = \{x \mid x \in M \wedge h(x) \in P\}$$

DEFINITION 3.9. Let  $h: (M, \oplus, u) \longrightarrow (P, \otimes, v)$  be a homomorphism of monoids. The set of elements of  $M$  which has the same image as  $x \in M$  under the homomorphism  $h$  is denoted by  $h^{-1}hM$ . Then,  $\{h^{-1}hx \mid x \in M\}$  is a partition of  $M$  called the quotient of  $M$  by  $h$  denoted  $M/h$ .

This concept of taking the inverse image of an image provided me with the key to the solution of the File System Case Study problem which I address in Chapter 5.

All of the above definitions are equally applicable to groups. There is one further concept which plays an important rôle in group homomorphisms and none whatsoever in monoid homomorphisms—the kernel. The kernel is the set of all elements in  $M$  that maps to the identity element  $v \in P$ . In the strict Theory of Monoids, i.e., excluding groups, the kernel of every monoid homomorphism consists of a singleton—the identity element  $u \in M$ . Armed with these definitions let us now examine the homomorphisms that underly much of the *VDM Meta-IV*.

# MONOIDS AND HOMOMORPHISMS

## 3.1. The Domain of Sequences

Essentially, all that one needs to know about the homomorphisms on the free monoid of sequences in *Meta-IV* is given by the following theorem adopted from (Arbib and Manes 1975).

**THEOREM 3.1. Unique Homomorphism Theorem:** *Let  $(\Sigma^*, \wedge, \Lambda)$  denote the free monoid with base  $\Sigma$ . Given an arbitrary monoid  $(M, \oplus, u)$  and any map  $F$  from  $\Sigma$  to  $M$  viewed as a set,  $F: \Sigma \longrightarrow M$ , there is a unique homomorphism  $\psi$  from  $(\Sigma^*, \wedge, \Lambda)$  to  $(M, \oplus, u)$  which extends  $F$ .*

Such homomorphisms will be called  $\Sigma^*$ -homomorphisms or simply sequence homomorphisms defined by

$$\begin{aligned} \psi: (\Sigma^*, \wedge, \Lambda) &\longrightarrow (M, \oplus, u) \\ \psi(\sigma \wedge \tau) &= \psi(\sigma) \oplus \psi(\tau) \\ \psi(\Lambda) &= u \end{aligned}$$

They are represented by the commuting diagrams shown below.

$$\begin{array}{ccccc} \Sigma & \xrightarrow{j} & \Sigma^* & & \Sigma^* \times \Sigma^* & \xrightarrow{\wedge} & \Sigma^* \\ & & \downarrow \psi & & \downarrow \psi \times \psi & & \downarrow \psi \\ & & M & & M \times M & \xrightarrow{\oplus} & M \end{array}$$

The important relation is given by

$$\psi(j(e)) = \psi(\langle e \rangle) = F(e)$$

where  $j$  is the unit function that injects an element of a set into a sequence of one element. It is of great importance in computing in the sense that it provides the mechanism by which a complex data structure can be built out of a simpler one. Indeed, without it, it is not possible to derive a naïve recursive algorithm from the homomorphism. Consider  $\psi(\langle e \rangle \wedge \tau) = \psi(\langle e \rangle) \oplus \psi(\tau)$ . Without knowing the image of  $\psi(\langle e \rangle)$ , we are unable to proceed any further. The derived naïve recursive algorithm is, therefore:

$$\begin{aligned} \psi: (\Sigma^*, \wedge, \Lambda) &\longrightarrow (M, \oplus, u) \\ \psi(\langle e \rangle \wedge \tau) &= F(e) \oplus \psi(\tau) \\ \psi(\Lambda) &= u \end{aligned}$$

In the next Chapter we will see that there are other ‘more efficient’ constructive recursive methods for denoting the  $\psi$  homomorphism. To elaborate on the significance

of the unique homomorphism theorem cited above, some simple  $\Sigma^*$ -homomorphisms are presented as basic lemmas.

LEMMA 3.1. *The Meta-IV sequence operator  $len$  is an endomorphism from the free monoid to the additive monoid of natural numbers,  $(\mathbf{N}, +, 0)$ .*

Formally

$$\begin{aligned} len: (\Sigma^*, \wedge, \Lambda) &\longrightarrow (\mathbf{N}, +, 0) \\ len(\sigma \wedge \tau) &= len \sigma + len \tau \\ len \Lambda &= 0 \end{aligned}$$

In particular, note that  $len(j(e)) = len \langle e \rangle = 1$ . The image of  $len$  is the set of natural numbers,  $\text{Im } len = len(\Sigma^*) = \mathbf{N}$ .

LEMMA 3.2. *The Meta-IV sequence operator  $elems$  is an epimorphism from the free monoid to the commutative monoid of sets,  $(\mathcal{P}\Sigma, \cup, \emptyset)$ .*

Formally

$$\begin{aligned} elems: (\Sigma^*, \wedge, \Lambda) &\longrightarrow (\mathcal{P}\Sigma, \cup, \emptyset) \\ elems(\sigma \wedge \tau) &= elems \sigma \cup elems \tau \\ elems \Lambda &= \emptyset \end{aligned}$$

Here,  $elems(j(e)) = elems(\langle e \rangle) = \{e\}$ . The image of  $elems$  is  $\text{Im } elems = elems(\Sigma^*) = \mathcal{P}\Sigma$ .

LEMMA 3.3. *The items operator is an epimorphism from the free monoid to the monoid of additive bags,  $(\Sigma \xrightarrow{m} \mathbf{N}, \oplus, \theta)$ .*

Formally

$$\begin{aligned} items: (\Sigma^*, \wedge, \Lambda) &\longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1, \oplus, \theta) \\ items(\sigma \wedge \tau) &= items(\sigma) \oplus items(\tau) \\ items(\Lambda) &= \theta \end{aligned}$$

Here,  $items(j(e)) = items(\langle e \rangle) = [e \mapsto 1]$ . The image of  $items$  is  $\text{Im } items = items(\Sigma^*) = \mathbf{N}_1$ .

Let us now consider the operation of reversing the elements of a sequence. We have already remarked on how it characterises the relationship between stacks and queues. In Automata Theory, it is used to distinguish between deterministic and nondeterministic pushdown automata. We have noted that it plays a fundamental

## MONOIDS AND HOMOMORPHISMS

rôle in the construction of the free group and it is, of course, in its naïve recursive form, a benchmark for PROLOG interpreters and compilers. In short, it is a *very* interesting operation on sequences. It comes as no surprise, therefore, to learn that *rev* is not a  $\Sigma^*$ -homomorphism since

$$\begin{aligned} \text{rev}(\sigma \wedge \tau) &= \text{rev}(\tau) \wedge \text{rev}(\sigma) \\ &\neq \text{rev}(\sigma) \wedge \text{rev}(\tau) \end{aligned}$$

However, *rev* is an isomorphism from  $(\Sigma^*, \wedge, \Lambda)$  to  $(\Sigma^*, \wedge^e, \Lambda)$ :

$$\begin{aligned} \text{rev}: (\Sigma^*, \wedge, \Lambda) &\longrightarrow (\Sigma^*, \wedge^e, \Lambda) \\ \text{rev}(\sigma \wedge \tau) &= \text{rev}(\tau) \wedge \text{rev}(\sigma) \\ &= \text{rev}(\sigma) \wedge^e \text{rev}(\tau) \\ \text{rev}(\Lambda) &= \Lambda \end{aligned}$$

Because  $(\Sigma^*, \wedge^e, \Lambda)$  is the mirror image of  $(\Sigma^*, \wedge, \Lambda)$ , then *rev* is said to be an anti-homomorphism of the free monoid  $(\Sigma^*, \wedge, \Lambda)$ . Indeed, it is an anti-isomorphism. Further details on anti-homomorphisms may be found in (Jacobson 1974, 108–10).

Palindromes are words that read the same backwards and forwards. For example,  $\langle m, a, d, a, m \rangle$  is a palindrome. A predicate to test if a word is a palindrome is simply given in terms of *rev*:

$$\text{palindrome}(\tau) \triangleq \tau = \text{rev}(\tau)$$

In other words, for any  $(\Sigma^*, \wedge, \Lambda)$ , the set of palindromes is stable under *rev*.

### 3.1.1. Filter Algorithms on Sequences

Were it simply the case that only the basic *Meta-IV* operators were homomorphisms and other (recursive) algorithms on sequences did not share such properties, then indeed, computing would be a drudgery. Happily that is not the case. I now consider some simple, but important algorithms that just happen to be homomorphisms. I also take this opportunity to present some of the flavour of the Irish School of the *VDM*. Specifically, in the search for a particular sequence homomorphism—the filter, we will not stop at the result but continue to explore other possibilities, even at the last, introducing some elementary notions from complexity theory.

First, there is an algorithm that deletes all occurrences of an element from a sequence:

$$\begin{aligned} delall: \Sigma &\longrightarrow (\Sigma^*, \wedge, \Lambda) \longrightarrow (\Sigma^*, \wedge, \Lambda) \\ delall[e](\sigma \wedge \tau) &= delall[e](\sigma) \wedge delall[e](\tau) \\ delall[e]\Lambda &= \Lambda \end{aligned}$$

with the constructive step given by  $delall[e](a) = (e = a \rightarrow \Lambda, \langle a \rangle)$ , expressed in McCarthy conditional form. For all  $e \in \Sigma$ ,  $delall[e]$  is an endomorphism of the free monoid  $(\Sigma^*, \wedge, \Lambda)$ . I have used the operator symbol,  $\Leftarrow$ , to denote the *delall* endomorphism. Using this operator, the endomorphism has the form

$$\begin{aligned} e \Leftarrow (\sigma \wedge \tau) &= (e \Leftarrow \sigma) \wedge (e \Leftarrow \tau) \\ e \Leftarrow \Lambda &= \Lambda \end{aligned}$$

Now a filter algorithm which removes duplicates from a sequence may be expressed in terms of the *delall* algorithm:

$$\begin{aligned} (\Sigma^*, \wedge, \Lambda) &\longrightarrow (\Sigma_!^*, \diamond, \Lambda) \\ filter(\langle e \rangle \wedge \tau) &\stackrel{\Delta}{=} \langle e \rangle \wedge filter(e \Leftarrow \tau) \\ filter(\Lambda) &\stackrel{\Delta}{=} \Lambda \end{aligned}$$

It would be nice to suppose that filter is the monoid homomorphism:

$$\begin{aligned} filter: (\Sigma^*, \wedge, \Lambda) &\longrightarrow (\Sigma_!^*, \diamond, \Lambda) \\ filter(\sigma \wedge \tau) &= filter(\sigma) \diamond filter(\tau) \\ filter(\Lambda) &= \Lambda \end{aligned}$$

Unfortunately, that is not the case. Consider the counter example  $filter(\langle e, a, e \rangle)$ :

$$\begin{aligned} filter(\langle e, a, e \rangle) &= \langle e \rangle \wedge filter(e \Leftarrow \langle a, e \rangle) \\ &= \langle e \rangle \wedge \langle a \rangle \\ &= \langle e, a \rangle \end{aligned}$$

On the other hand

$$\begin{aligned} filter(\langle e \rangle \wedge \langle a, e \rangle) &= filter(\langle e \rangle) \diamond filter(\langle a, e \rangle) \\ &= \langle e \rangle \diamond \langle a, e \rangle \\ &= \Lambda \diamond \langle a, e \rangle \\ &= \langle a, e \rangle \end{aligned}$$

In each case we obtain unique sequences. Unfortunately, the ordering property of the sequence prohibits this definition of filter from being a homomorphism. This is a common problem with algorithms on sequences due to the inherent ordering of

## MONOIDS AND HOMOMORPHISMS

the elements. A filter algorithm that is indeed a homomorphism may be obtained by mimicking the definition of the  $\diamond$  operator. Let us write the definition of the  $\diamond$  operator in the form:

$$(\langle e \rangle \wedge \sigma) \diamond \tau = \chi[[e]]\tau(\sigma \diamond \tau, \langle e \rangle \wedge (\sigma \diamond \tau))$$

Then, we define

$$\text{filter}(\langle e \rangle \wedge \tau) \triangleq \chi[[e]]\tau(\text{filter}(\tau), \langle e \rangle \wedge \text{filter}(\tau))$$

Denoting this filter algorithm by the symbol,  $\wr$ , we then have

$$\wr(\sigma \wedge \tau) = (\wr\sigma) \diamond (\wr\tau)$$

$$\wr\Lambda = \Lambda$$

Now, it is clear that the problem with respect to the original filter algorithm arose simply because the definition of the  $\diamond$  operator depended on ordering. This leads us to the conjecture that we can find another monoid of unique sequences under the operation  $\diamond'$  such that the original filter algorithm is a homomorphism:

$$(\langle e \rangle \wedge \sigma) \diamond' \tau = \langle e \rangle \wedge ((e \triangleleft \sigma) \diamond' (e \triangleleft \tau))$$

$$\Lambda \diamond' \sigma = \sigma = \sigma \diamond' \Lambda$$

where I have used the *delall* in the definition of the new operator. Note, in particular, that I have taken advantage of the fact that  $e \triangleleft \sigma = \sigma$  in order to eliminate the characteristic function  $\chi[[e]]\sigma$ .

There is still another filter algorithm that determines the  $\wr$  homomorphism. If we look at the original filter algorithm from the point of view of complexity, it is clear that it is of the order  $n^2$  where  $n$  is the length of the sequence. To obtain a linear algorithm of the order  $n$ , we may introduce some ancillary structure that ‘counts’ occurrences of elements as we encounter them. This idea comes from an example of the use of bags in Trace Theory to be discussed in Chapter 5. We do not actually need to count the *number* of occurrences, just simply to mark the occurrences. Thus instead of using a bag, a set will do. These ideas lead to the following algorithm:

$$\text{filter}: \Sigma^* \longrightarrow \Sigma_!^*$$

$$\text{filter}(\sigma) \triangleq \pi_2 \circ \text{filter}[[\sigma]](\emptyset, \Lambda)$$

where the tail-recursive form is defined by

$$\begin{aligned}
 \text{filter}: \Sigma^* &\longrightarrow (\mathcal{P}\Sigma \times \Sigma_!^*) \longrightarrow (\mathcal{P}\Sigma \times \Sigma_!^*) \\
 \text{filter}[\Lambda](s, \tau) &\stackrel{\Delta}{=} (s, \tau) \\
 \text{filter}[\langle e \rangle \wedge \sigma](s, \tau) &\stackrel{\Delta}{=} \\
 &(\text{filter}[\sigma])\chi[e]s((s, \tau), (s \uplus \{e\}, \tau \wedge \langle e \rangle))
 \end{aligned}$$

where I have used the inversion

$$\begin{aligned}
 &(\text{filter}[\sigma])\chi[e]s((s, \tau), (s \uplus \{e\}, \tau \wedge \langle e \rangle)) \\
 &= \chi[e]s(\text{filter}[\sigma](s, \tau), \text{filter}[\sigma](s \uplus \{e\}, \tau \wedge \langle e \rangle))
 \end{aligned}$$

As a ‘by-product’ of this algorithm the set of elements in the sequence  $\sigma$  is given by  $\pi_1 \circ \text{filter}[\sigma](\emptyset, \Lambda)$ . Finally, the expression  $\tau \wedge \langle e \rangle$  in the tail-recursive form is significant in so far that it preserves the ordering of the original sequence. Were I to write  $\langle e \rangle \wedge \tau$  instead, and thus produce the result in reverse order, I would obtain yet another filter algorithm that was neither a homomorphism of  $(\Sigma_!^*, \diamond, \Lambda)$ , nor  $(\Sigma_!^*, \diamond', \Lambda)$ .

### 3.1.2. The Hashing Algorithm on Sequences

A hashing function is a monoid homomorphism from the freemonoid  $(\Sigma^*, \wedge, \Lambda)$  to the additive cyclic group of integers modulo  $p$ , where  $p$  is prime, considered as a monoid,  $(\mathbf{Z}_p, +, 0)$ .

$$\begin{aligned}
 \text{hash}(\sigma \wedge \tau) &= (\text{hash}(\sigma) + \text{hash}(\tau)) \bmod p \\
 \text{hash}(\Lambda) &= 0
 \end{aligned}$$

where  $\text{hash}(j(e)) = \text{hash}(\langle e \rangle) = \dots$ , the value in  $\mathbf{Z}_p$  depending on the particular hashing function chosen. In other words,  $\text{hash}$  denotes a whole family of homomorphisms from  $(\Sigma^*, \wedge, \Lambda)$  to  $(\mathbf{Z}_p, +, 0)$ . Further, the fact that  $\text{hash}$  is obviously an epimorphism, i.e.,  $\text{Im } \text{hash} = \mathbf{Z}_p$ , immediately implies the need for handling so-called ‘collisions’ in a computer implementation. One curious member of the family is the constant hash function  $\text{hash}: \sigma \mapsto 0$  which has been found to be convenient when debugging a program (Knuth 1973, 3:513).

## MONOIDS AND HOMOMORPHISMS

### 3.1.3. The Mergesort Algorithm on Sequences

Mergesort is a monoid homomorphism from  $(\Sigma^*, \wedge, \Lambda)$  to the monoid  $(\Sigma_{\leq}^*, \nabla, \Lambda)$ .

$$\begin{aligned} \text{mergesort}(\sigma \wedge \tau) &= \text{mergesort}(\sigma) \nabla \text{mergesort}(\tau) \\ \text{mergesort}(\Lambda) &= \Lambda \end{aligned}$$

where  $\nabla$  is the merge operator on sorted sequences and

$$\text{mergesort}(j(e)) = \text{mergesort}(\langle e \rangle) = \langle e \rangle$$

But what exactly is mergesort and what relation does it bear to an arbitrary sort algorithm, say insertion sort or quicksort? A moment's reflection will indicate that the essential nature of mergesort depends on the monoid  $(\Sigma_{\leq}^*, \nabla, \Lambda)$ , and in particular on the definition of  $\nabla$ . Let  $\text{sort}_1$  and  $\text{sort}_2$  denote two distinct sorting algorithms. Then  $\text{sort}_1(\sigma) \nabla \text{sort}_2(\tau)$  is an element of  $(\Sigma_{\leq}^*, \nabla, \Lambda)$ . In other words, we can always split a sequence into  $n$  parts, retaining the same ordering among the elements, of course, use a different sorting algorithm on each part and obtain the required result using the operator  $\nabla$ . Sorting *is* the mergesort homomorphism.

### 3.1.4. Finite State Machines

The following homomorphism is fundamental in the Theory of Finite State Machines. Let  $M = (Q, \Sigma, F)$  denote a finite state machine, where  $Q$  is the set of states,  $\Sigma$  is the (input) alphabet, and  $F: Q \times \Sigma \rightarrow Q$  is the state transition function. Define  $PF(Q)$  to be the set of all partial functions  $F_\sigma: Q \rightarrow Q, \forall \sigma \in \Sigma$ . Then the function  $G: \Sigma \rightarrow PF(Q)$ , which maps an element  $e \in \Sigma$  to the function  $F_e$ , may be extended to the monoid homomorphism  $\psi$ :

$$\begin{aligned} \psi(\sigma \wedge \tau) &= \psi(\sigma) \circ \psi(\tau) \\ &= F_\sigma \circ F_\tau \\ \psi(\Lambda) &= F_\emptyset = \theta \end{aligned}$$

where  $\psi(j(e)) = \psi(\langle e \rangle) = F_e$  and from here we may proceed into a study of transformation semigroups and transformation monoids (Eilenberg 1976, 2:6).

It is worth recalling that all of the material presented in this subsection derives from a single theorem—the unique homomorphism theorem. Within the framework of a single concept, the  $\Sigma^*$ -homomorphism, we have been able to explore a wide variety of material, all of which is relevant to the use of the sequence domain in the VDM. Before introducing homomorphisms for the other basic domains, I would like

to return to the distributed concatenation of sequences  $\hat{\ } /$  and the star functor  $-^*$ . It should not come as a surprise to learn that both are homomorphisms. Strictly speaking, the distributed concatenation is better described as a semigroup homomorphism:

$$\begin{aligned} \hat{\ } / : ((\Sigma^*)^+, \hat{\ }) &\longrightarrow (\Sigma^*, \hat{\ }) \\ \hat{\ } / (\sigma \hat{\ } \tau) &\triangleq \hat{\ } / (\sigma) \hat{\ } \hat{\ } / (\tau) \end{aligned}$$

where  $\hat{\ } / \langle \sigma \rangle = \sigma$ . In particular, for  $\sigma = \Lambda$ , we have  $\hat{\ } / \langle \Lambda \rangle = \Lambda$ . It is clear that we may extend the definition to a monoid homomorphism by  $\hat{\ } / \Lambda = \Lambda$ .

The star functor is the mechanism by which we may obtain a family of monoid isomorphisms. To every function  $f: \Sigma \longrightarrow T$ , there is the corresponding isomorphism  $f^*: (\Sigma^*, \hat{\ }, \Lambda) \longrightarrow (T^*, \hat{\ }, \Lambda)$  given by:

$$\begin{aligned} f^*(\sigma \hat{\ } \tau) &= f^*(\sigma) \hat{\ } f^*(\tau) \\ f^*(\Lambda) &= \Lambda \end{aligned}$$

Finally, it is important to give an indication of the kind of thinking that ought to be developed by those who adopt the Irish School of the *VDM*. We will see later how this sort of algebra is used in practice in formal specifications. But, there is also the important aspect of achieving some insight into both the theoretical and the practical aspects of computing. For this purpose I now present some observations on the notions of parsing and of the difference list in PROLOG.

### 3.1.5. Applications

In essence the *len* homomorphism states that the monoid of natural numbers under addition,  $(\mathbf{N}, +, 0)$  is a simpler structure than the free monoid  $(\Sigma^*, \hat{\ }, \Lambda)$ ; but that the structure of both is basically the same. This implies that certain properties of the free monoid are carried over into the natural numbers. Hence, by focusing on the latter and considering the sorts of properties that it might have, we can obtain some insight into the properties of the former—which is at the foundation of computing. I might explore recursive function theory, which depends on  $\mathbf{N}$ , and then try to relate this to the free monoid. However, in this thesis, I have chosen some elementary material from number theory and the construction of the integers from the naturals.

One area of number theory is concerned with the study of partitions. In how many ways can one partition a number  $n$ ,  $n \geq 1$ ? For example, 5 has the partitions

## MONOIDS AND HOMOMORPHISMS

1 + 4, 1 + 1 + 3, 1 + 1 + 1 + 2, 1 + 1 + 1 + 1 + 1, 1 + 2 + 2, 2 + 3. Now the interesting point is not so much to find a solution to this problem as to ask what the corresponding problem is in the free monoid: In how many ways can one partition a sequence? This leads us to consider the meaning of the partition of a sequence—to find a problem in computing that is naturally a partition problem. After a little reflection, we identify one interpretation—the parsing problem, and now we may ask why it is that parsing is tractable. A program is simply a sequence of characters over some alphabet. The first step in parsing is to map the sequence of characters into a sequence of tokens—this is the lexical analysis stage. It itself is a partitioning problem. We might wonder whether an algorithm for lexical analysis is a free monoid homomorphism. The answer is no! Assuming the usual conventions for a Pascal-like language, then I offer the counter-example:

$$\begin{aligned} \text{lex}(\langle s, u, m \rangle \wedge \langle 1, 2 \rangle) &= \langle \text{sum}12 \rangle \\ &\neq \langle \text{sum}, 12 \rangle \\ &= \text{lex}(\langle s, u, m \rangle \wedge \text{lex}(\langle 1, 2 \rangle)) \end{aligned}$$

Lexical analysis works due to the meaning given to ‘white space’ and the set of separator symbols. One might argue that we can extend the alphabet to include white space and operator symbols, and then attempt to demonstrate that lex was a homomorphism. A counter-example to this proposal is left as a challenge.

Parsing takes a sequence of tokens and produces a parse tree. How does a parser partition the sequence? The secret lies in the concept of token classes. But since one class predominates in a program—the class of identifiers, it has been found pragmatic to consider a partitioning of that class. Consequently, the idea of a keyword was introduced. The classic example of a keyword which appears strange in practice, i.e., from the conceptual point of view of the programmer, but which is used to facilitate partitioning, is the **with** keyword used to distinguish ‘formal parameters of type function or procedure to a generic unit’, from ‘generic units of type function or procedure’. Let us now look at a different train of thought which also happens to lead to some insight into the problem of partitioning/parsing.

Perhaps one of the most unusual structures employed in computing is the difference list (used for efficient computation in PROLOG (Hogger 1984)). However, the treatment of the difference list seems to be very much an *ad hoc* affair. Hogger intimates that the use of the difference list is a ‘somewhat unintuitive device’, which

is understandable in the absence of a proper formal theory of difference lists. Such a formal basis is now presented.

The difference list may be derived from a study of the inverse image of the  $len$  homomorphism,  $len^{-1} \mathbf{N}$ . First let us consider the construction of the integers. The set  $\mathbf{Z}$  of rational integers may be defined as the set of difference numbers (Godement 1969) as follows. Consider pairs of elements  $(m, n)$  in the cartesian product domain  $\mathbf{N} \times \mathbf{N}$ . Such a pair is said to be in a reduced canonical form if either  $m$  or  $n$  (or both are 0). A rule for constructing such canonical forms is simply

$$(m, n) \equiv \begin{cases} (m - n, 0), & \text{if } m \geq n; \\ (0, n - m), & \text{otherwise.} \end{cases}$$

The canonical form  $(m, 0)$  is identified with the positive rational integer  $m$ , and the form  $(0, n)$  with the strictly negative rational integer  $-n$ . The laws of addition and multiplication may be shown to be

$$(m_1, n_1) + (m_2, n_2) = (m_1 + m_2, n_1 + n_2)$$

$$(m_1, n_1)(m_2, n_2) = (m_1m_2 + n_1n_2, m_1n_2 + m_2n_1)$$

the latter being reminiscent of the law of multiplication of rational numbers and complex numbers. Defining an equivalence relation  $\mathcal{R}$  on the set  $\mathbf{N} \times \mathbf{N}$  by

$$(m, n)\mathcal{R}(m', n') \iff m + n' = m' + n$$

gives the quotient set  $(\mathbf{N} \times \mathbf{N}) \setminus \mathcal{R}$  which is isomorphic to  $\mathbf{Z}$ . With the laws of addition and multiplication given above, one obviously has the additive monoid  $((\mathbf{N} \times \mathbf{N}) \setminus \mathcal{R}, +, (0, 0))$  and multiplicative monoid  $((\mathbf{N} \times \mathbf{N}) \setminus \mathcal{R}, \times, (1, 0))$ . Indeed, since  $(m, n)$  has the additive inverse  $(n, m)$  then  $((\mathbf{N} \times \mathbf{N}) \setminus \mathcal{R}, +)$  is an additive group.

It is now pertinent to ask what sort of structure maps onto the additive monoid  $((\mathbf{N} \times \mathbf{N}) \setminus \mathcal{R}, +, (0, 0))$  under the  $len$  homomorphism and it is reasonable to suppose that it has the form of a quotient of  $(\Sigma^* \times \Sigma^*)$  with an appropriate definition of concatenation.

$$\begin{array}{ccc} \Sigma^* & \xrightarrow{\times} & (\Sigma^* \times \Sigma^*)/R \\ \downarrow len & & \downarrow len \times len \\ \mathbf{N} & \xrightarrow{\times} & (\mathbf{N} \times \mathbf{N})/R \end{array}$$

Such a pair is a difference sequence (i.e., difference list) and it may be shown that  $(\Sigma^* \times \Sigma^*) \setminus len$  is a quotient of  $\Sigma^*$ . Let us now seek to define a concatenation operator

## MONOIDS AND HOMOMORPHISMS

such that  $((\Sigma^* \times \Sigma^*) \setminus \text{len}, \wedge, (\Lambda, \Lambda))$  is isomorphic to  $(\Sigma^*, \wedge, \Lambda)$ .

As a first step we will consider combinations of canonical forms. The goal is to ensure that  $\text{len} \times \text{len}$  maps  $(\sigma, \Lambda)$  onto the positive rational integers:

$$\begin{aligned} (\sigma_1, \Lambda) \wedge (\sigma_2, \Lambda) &\stackrel{\Delta}{=} (\sigma_1 \wedge \sigma_2, \Lambda \wedge \Lambda) \\ &= (\sigma_1 \wedge \sigma_2, \Lambda) \\ (\Lambda, \tau_1) \wedge (\Lambda, \tau_2) &\stackrel{\Delta}{=} (\Lambda \wedge \Lambda, \tau_1 \wedge \tau_2) \\ &= (\Lambda, \tau_1 \wedge \tau_2) \end{aligned}$$

Justification should be obvious:

$$\begin{aligned} \text{len} \times \text{len}: (\sigma_1, \Lambda) \wedge (\sigma_2, \Lambda) &\mapsto (\text{len} \times \text{len})(\sigma_1, \Lambda) + (\text{len} \times \text{len})(\sigma_2, \Lambda) \\ &= (\text{len } \sigma_1, \text{len } \Lambda) + (\text{len } \sigma_2, \text{len } \Lambda) \\ &= (m_1, 0) + (m_2, 0) \\ &= (m_1 + m_2, 0) \end{aligned}$$

and

$$\begin{aligned} \text{len} \times \text{len}: (\sigma_1 \wedge \sigma_2, \Lambda) &\mapsto (\text{len}(\sigma_1 \wedge \sigma_2), \text{len } \Lambda) \\ &= (\text{len } \sigma_1 + \text{len } \sigma_2, \text{len } \Lambda) \\ &= (m_1 + m_2, 0) \end{aligned}$$

Similarly,  $\text{len} \times \text{len}$  maps  $(\Lambda, \tau)$  onto strictly negative rational integers.

Now, let us address the issue of combinations of the form  $(\sigma, \Lambda) \wedge (\Lambda, \tau)$  and  $(\Lambda, \tau) \wedge (\sigma, \Lambda)$ . Under the  $\text{len} \times \text{len}$  homomorphism, we will want the effect of ‘subtraction’. Hence, we must establish the well-formedness constraints for pairs  $(\sigma, \tau)$ . As a first attempt, I considered the following definition:

$$(\sigma, \Lambda) \wedge (\Lambda, \tau) \stackrel{\Delta}{=} \begin{cases} (\text{maxprefix}(\sigma, \tau), \Lambda), & \text{if } \sigma = \sigma' \wedge \tau; \\ (\Lambda, \text{maxsuffix}(\sigma, \tau)), & \text{if } \tau = \sigma \wedge \tau'. \end{cases}$$

But this did not cater for concatenation of pairs of the form  $(\langle a \rangle, \Lambda)$  and  $(\Lambda, \langle b \rangle)$ . I failed completely to give a suitable definition for concatenation and, hence, to produce the required monoid. Instead of abandoning the approach, I followed the usual sound advice of Pólya and considered a subproblem which is directly relevant to computing. Specifically, I focused on the set of all pairs  $(\sigma, \tau)$  such that  $\sigma = \sigma' \wedge \tau$ . Such a pair maps to the ordinary sequence  $\text{maxprefix}(\sigma, \tau)$ . For example, the sequence pairs  $(\text{endfor}, \text{for})$ ,  $(\text{endwhile}, \text{while})$ , and  $(\text{endif}, \text{if})$  all map to  $\text{end}$ . The appropriate, though admittedly strange-looking, definition of concatenation of such

pairs is

$$(\sigma_1, \tau_1) \wedge (\sigma_2, \tau_2) \triangleq (\sigma_1 \uparrow \tau_1 \wedge \sigma_2 \uparrow \tau_2 \wedge \sigma_1 \downarrow \tau_1 \wedge \sigma_2 \downarrow \tau_2, \sigma_1 \downarrow \tau_1 \wedge \sigma_2 \downarrow \tau_2)$$

where  $\uparrow$  and  $\downarrow$  denote *maxprefix* and *maxsuffix*, respectively. With this definition and the obvious equivalence relation, we now have a monoid which is isomorphic to the free monoid.

To illustrate the applicability of such (restricted) difference sequences in computing, consider the following tail-recursive algorithm, denoted symbolically by  $\mathbf{r}$ , where I have explicitly exhibited the maxprefix property:

$$\begin{aligned} \mathbf{r}[\langle e \rangle \wedge \sigma \wedge \sigma', \sigma'](\tau_1, \tau_2) &= \mathbf{r}[\sigma \wedge \sigma', \sigma'](\langle e \rangle \wedge \tau_1, \tau_2) \\ \mathbf{r}[(\sigma, \sigma)](\tau_1, \tau_2) &= (\tau_1, \tau_2) \end{aligned}$$

then reversal of a sequence  $\sigma$  is given by  $\pi_1 \circ \mathbf{r}[(\sigma, \Lambda)](\Lambda, \Lambda)$ . It is precisely in this manner that the difference list is exploited in PROLOG.

### 3.2. The Domain of Sets

In the case of the sequence domain, I have concentrated on presenting sequence homomorphisms as monoid homomorphisms. For the powerset domain I introduce the notion of semiring. Consider the pair of monoids  $(\mathbf{N}, +, 0)$  and  $(\mathbf{N}_1, \times, 1)$ . In addition to the ‘inner laws’ or operations  $+$  and  $\times$ , the set of natural numbers also enjoys the property that multiplication distributes over addition:

$$\forall n_1, n_2 \in \mathbf{N}, \forall m \in \mathbf{N}_1: m \times (n_1 + n_2) = (m \times n_1) + (m \times n_2)$$

Consequently, it is customary to call the set of natural numbers a semiring, denoted  $(\mathbf{N}, +, \times, 0, 1)$ . It is also clear that the distributive law has the form of a homomorphism, i.e., multiplication with respect to a number  $m \in \mathbf{N}_1$  is a  $(\mathbf{N}, +, 0)$  homomorphism.

Now considering the powerset domain of *Meta-IV*, we have the pair of monoids  $(\mathcal{P}X, \cup, \emptyset)$  and  $(\mathcal{P}X, \cap, X)$ . Since intersection distributes over set union, then the powerset domain is also a semiring,  $(\mathcal{P}X, \cup, \cap, \emptyset, X)$ . Similarly, the set of boolean values, denoted here by  $\{0, 1\}$ , may also be viewed as a semiring. Then the membership operation of the powerset domain is a semiring homomorphism.

LEMMA 3.4. *The membership function  $\chi[\sigma]$ , indexed with respect to some element  $\sigma \in \Sigma$  is an epimorphism from the semiring of sets,  $(\mathcal{P}\Sigma, \cup, \cap, \emptyset, \Sigma)$ , to the Boolean semiring,  $(\mathbf{B}, \vee, \wedge, 0, 1)$ .*

## MONOIDS AND HOMOMORPHISMS

Formally, the membership function is a pair of monoid epimorphisms

$$\begin{aligned}\chi[\sigma]: (\mathcal{P}\Sigma, \cup, \emptyset) &\longrightarrow (\mathbf{B}, \vee, 0) \\ \chi[\sigma](s_1 \cup s_2) &= \chi[\sigma]s_1 \vee \chi[\sigma]s_2 \\ \chi[\sigma]\emptyset &= 0\end{aligned}$$

and also

$$\begin{aligned}\chi[\sigma]: (\mathcal{P}\Sigma, \cap, \Sigma) &\longrightarrow (\mathbf{B}, \wedge, 1) \\ \chi[\sigma](s_1 \cap s_2) &= \chi[\sigma]s_1 \wedge \chi[\sigma]s_2 \\ \chi[\sigma]\Sigma &= 1\end{aligned}$$

The image is  $\text{Im } \chi[\sigma] = \mathbf{B}$ . The former endomorphism, which is the lookup operation,  $Lkp$ , plays an important rôle in our simple model of a spelling-checker dictionary as a set of words.

There is not very much that can be said about the homomorphisms of the monoid  $(\mathcal{P}\Sigma, \cup, \emptyset)$  that is particularly relevant to formal specifications. The reason is simple. This monoid is usually at ‘the receiving end’ of the homomorphisms of other monoids where the underlying domain is either the sequence or the map. However, it is useful to note that set complementation, with respect to  $\mathcal{P}\Sigma$ , denoted  $\mathbf{C}$ , is an isorphism from  $(\mathcal{P}\Sigma, \cup, \emptyset)$  to  $(\mathcal{P}\Sigma, \cap, \Sigma)$ :

$$\begin{aligned}\mathbf{C}(S_1 \cup S_2) &= \mathbf{C}(S_1) \cap \mathbf{C}(S_2) \\ \mathbf{C}(\emptyset) &= \Sigma\end{aligned}$$

As an aside, let me remark that we have just completed all the essential groundwork that permits one to enter the field of topology.

Let us return to the remark that the monoid  $(\mathcal{P}\Sigma, \cup, \emptyset)$  is usually at the receiving end of some homomorphism. This is certainly the case of the *elems* homomorphism of the free monoid. Now we might be tempted to wonder whether there is a special sequence operator, which is a  $\Sigma^*$ -homomorphism, that takes the free monoid into  $(\mathcal{P}\Sigma, \cap, \Sigma)$  and that we have inadvertently omitted. That such a homomorphism exists is given by the unique homomorphism theorem and we have just constructed it:  $\mathbf{C} \circ \text{elems}$ . The corresponding sequence operator, which I will denote by  $\text{elems}^{\mathbf{C}}$ , is that which returns the set of elements **not** in a sequence. In Chapter 2 I said that one does not often have use for set intersection in formal specifications. The reason may now be made clear. The monoids  $(\mathcal{P}\Sigma, \cup, \emptyset)$  and  $(\mathcal{P}\Sigma, \cap, \Sigma)$  are isomorphic, also known as dual structures. Anything that can be done with one can also be done with the other, by the principle of duality.

Recall that the distributed concatenation of sequences,  $\wedge/$ , and the functor,  $-^*$ , also gave us homomorphisms of the free monoid. We would expect that the distributed union of sets,  $\cup/$ , and the functor,  $\mathcal{P}-$ , would also give us homomorphisms. Let  $S = \{S_i \mid S_i \in \mathcal{P}\Sigma, i \in \mathbf{N}_1\}$  be an indexed family of subsets of some set  $\Sigma$ . Consider the distributed union operator  $\cup/$  applied to  $S$ :

$$\begin{aligned} \cup/S &= \cup/\{S_1, S_2, \dots, S_i, \dots\} \\ &= S_1 \cup S_2 \cup \dots \cup S_i \cup \dots \cup S_n \\ &= (S_1 \cup \dots \cup S_i) \cup (S_{i+1} \cup \dots \cup S_n) \\ &= (\cup/\{S_1 \dots S_i\}) \cup (\cup/\{S_{i+1} \dots S_n\}) \end{aligned}$$

Clearly this is a semigroup epimorphism which I give as the lemma

LEMMA 3.5. *The distributed union operator  $\cup/$  is a semigroup epimorphism from  $(\mathcal{P}\mathcal{P}\Sigma, \cup)$  to  $(\mathcal{P}\Sigma, \cup)$ .*

Formally

$$\cup/(S \cup T) = \cup/S \cup \cup/T$$

with image  $\text{Im } \cup/ = \mathcal{P}\Sigma$ . There are, clearly, equivalent semigroup homomorphisms  $\cap/$  and  $\Delta/$ . The powerset functor gives a whole family of homomorphisms

LEMMA 3.6. *Let  $f: X \longrightarrow Y$  be an element of the function set  $Y^X$ . Then  $\mathcal{P}f$  is an epimorphism from the monoid  $(\mathcal{P}X, \cup, \emptyset)$  to the monoid  $(\mathcal{P}Y, \cup, \emptyset)$ .*

Formally

$$\begin{aligned} \mathcal{P}f(S \cup T) &= (\mathcal{P}fS) \cup (\mathcal{P}fT) \\ \mathcal{P}f(\emptyset) &= \emptyset \end{aligned}$$

Even in the case that the function  $f$  is bijective, we usually end up with an epimorphism. This is due to the ‘peculiarity’ of the concept of set which does not permit duplicate elements. I will illustrate this peculiarity in the material below on applications.

# MONOIDS AND HOMOMORPHISMS

## 3.2.1. Applications

In the Irish School of the *VDM*, the domain of relations, which is dealt with at length in Chapter 5, plays a fundamental rôle. Here, I would like to illustrate how their definition may be expressed using the operator calculus of the Irish School. The definition is adopted from (Eilenberg 1976, 2:2). A relation  $f$  from a set  $X$  to a set  $Y$ , written  $f: X \longrightarrow Y$ , is defined to be a function

$$f: \mathcal{P}X \longrightarrow \mathcal{P}Y$$

which is completely additive, i.e., which satisfies

$$(f \circ \cup/)S = (\cup/ \circ \mathcal{P}f)S$$

where  $S = \{S_i \mid i \in I\}$  is a family of subsets of  $X$  indexed by some set  $I$ . This may be illustrated by the commutative diagram

$$\begin{array}{ccc} \mathcal{P}X & \xrightarrow{f} & \mathcal{P}Y \\ \uparrow \cup/ & & \uparrow \cup/ \\ \mathcal{P}\mathcal{P}X & \xrightarrow{\mathcal{P}f} & \mathcal{P}\mathcal{P}Y \end{array}$$

Now we might ask what the term ‘completely additive’ means. We recognise the form  $\cup/ \circ \mathcal{P}f$  as the composition of a pair of  $(\mathcal{P}X, \cup, \emptyset)$  homomorphisms. Expanding the definition of ‘completely additive’ gives us

$$f(S_1 \cup S_2 \cup \dots \cup S_n) = f(S_1) \cup f(S_2) \cup \dots \cup f(S_n)$$

Thus  $f$  is essentially the extension of a monoid homomorphism  $f: (\mathcal{P}X, \cup, \emptyset) \longrightarrow (\mathcal{P}Y, \cup, \emptyset)$ . In view of this additivity,  $f$  is completely determined by its values on single element subsets, so that  $f$  may be viewed as a function

$$f: X \longrightarrow \mathcal{P}Y$$

and, thus, we are led directly and rigorously to the important *VDM* map domain of the form

$$MODEL = X \xrightarrow{m} \mathcal{P}Y$$

which models relations. To complete this section on the powerset domain I now consider a simple practical application.

Complexity is often managed by partitioning a problem into subproblems. In the case of a dictionary, we may wish to explore the possibility of partitioning it:

$$DICT = \mathcal{P}PART$$

$$PART = \mathcal{P}WORD$$

A typical dictionary  $D$  is then really a set of dictionaries,  $\{\delta_1, \delta_2, \dots, \delta_n\}$ , such that each pair  $(\delta_j, \delta_k)$  is distinct, i.e.,  $\delta_j \cap \delta_k = \emptyset$ . We might suppose this to be a model of a distributed dictionary. The model given has the form  $\mathcal{P}\mathcal{P}X$ . In Chapter 2, we noted that  $card(S_1 \cup S_2) \leq card(S_1) + card(S_2)$  with equality only in the case that  $S_1$  and  $S_2$  are disjoint. This was equally true for the symmetric difference operator. In the case of the partitioned dictionary model, it is obvious that the sum of the cardinalities of the partitions should equal the cardinality of a non-partitioned dictionary. If we let  $\delta$  denote the nonpartitioned dictionary which corresponds to  $D$ , then this essential property is simply

$$|\delta| = |\delta_1| + |\delta_2| + \dots + |\delta_n|$$

which resembles the definition of ‘completely additive’ given above:

$$(card \circ^{\cup})D = (+ / \circ \mathcal{P} card)D$$

But, whereas, there was no problem if  $f$  mapped two distinct sets  $S_j$  and  $S_k$  to the same set in the former definition, we do have a loss of information should two sets have the same cardinality. One way by which we may circumvent this difficulty is to inject the sets into a map:  $j: S \mapsto [S \mapsto |S|]$  and then to sum the range elements. It was this solution that I presented in Chapter 2 to illustrate the inverse image operator for map domains.

### 3.3. The Domain of Maps

The VDM map domain is probably the most important domain used in formal specifications. From the point of view of the theory of monoids, it gives rise to a wide variety of structures and corresponding homomorphisms. Only a certain amount of material will be touched upon here. In its most general form

$$MODEL = X \xrightarrow[m]{} Y$$

the map domain gives monoids with respect to the override operator,  $+$ , the symmetric difference operator,  $\Delta$ , which generalises the extend operator, and the composition operator,  $\circ$ .

## MONOIDS AND HOMOMORPHISMS

LEMMA 3.7. *The Meta-IV map operator  $\text{dom}$  is a monoid homomorphism from the domain of maps  $(X \xrightarrow{m} Y, +, \theta)$  to the domain of sets  $(\mathcal{P}X, \cup, \emptyset)$ .*

Formally

$$\begin{aligned}\text{dom}(\mu_1 + \mu_2) &= \text{dom } \mu_1 \cup \text{dom } \mu_2 \\ \text{dom } \theta &= \emptyset\end{aligned}$$

The  $\text{rng}$  operator is *not* a monoid homomorphism from  $(X \xrightarrow{m} Y, +, \theta)$  to  $(\mathcal{P}X, \cup, \emptyset)$ , as demonstrated by the counter-example:

$$\begin{aligned}\text{rng}([a \mapsto x, b \mapsto y] + [b \mapsto x]) &= \{x\} \\ \text{rng}([a \mapsto x, b \mapsto y]) \cup \text{rng}([b \mapsto x]) &= \{x, y\} \cup \{x\} = \{x, y\}\end{aligned}$$

However, it does have the property that

$$\begin{aligned}\text{rng}(\mu_1 + \mu_2) &\subseteq \text{rng } \mu_1 \cup \text{rng } \mu_2 \\ \text{rng } \theta &= \emptyset\end{aligned}$$

A few simple examples will show that any map ‘deletion’ operator, such as override,  $+$ , and restriction,  $\triangleleft$ , in combination with the  $\text{rng}$  operator will, in general, give a result that is not an equality relation, but a strict inclusion relation.

Although we can not use the extend operator to give us a monoid structure of maps, it is clear that the following equations hold for it:

$$\begin{aligned}\text{dom}(\mu_1 \cup \mu_2) &= \text{dom } \mu_1 \cup \text{dom } \mu_2 \\ \text{rng}(\mu_1 \cup \mu_2) &= \text{rng } \mu_1 \cup \text{rng } \mu_2\end{aligned}$$

Similarly,  $\text{dom}$  is a  $(X \xrightarrow{m} Y, \Delta, \theta)$  homomorphism.

Consider the monoid of bags  $(\Sigma \xrightarrow{m} \mathbf{N}_1, \oplus, \theta)$ . The size of a bag is a monoid homomorphism  $\text{card}: (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow \mathbf{N}$ :

$$\begin{aligned}\text{card}: (\Sigma \xrightarrow{m} \mathbf{N}_1, \oplus, \theta) &\longrightarrow (\mathbf{N}, +, 0) \\ \text{card}(\beta_1 \oplus \beta_2) &= \text{card}(\beta_1) + \text{card}(\beta_2) \\ \text{card}(\theta) &= 0\end{aligned}$$

where  $\text{card}([e \mapsto n]) = n$ . Using the operator symbol  $|-|$ , the  $\text{card}$  homomorphism may be expressed simply as

$$\begin{aligned}|\beta_1 \oplus \beta_2| &= |\beta_1| + |\beta_2| \\ |\theta| &= 0\end{aligned}$$

If we omit the null set from the partitions of a set, then the mechanism we used to sum the cardinalities of the partitions of a set may be precisely stated in the form

$$|\oplus/(\mathcal{P}J)S|$$

I would now like to introduce some further concepts from algebra in order to give an interpretation to map restriction and map removal. Consider the monoid of natural numbers  $(\mathbf{N}, +, 0)$ . We saw that every number  $m \in \mathbf{N}_1$  could be given the interpretation as a monoid endomorphism, which I express here in curried form:

$$mul\llbracket m \rrbracket(n_1 + n_2) = (mul\llbracket m \rrbracket n_1) + (mul\llbracket m \rrbracket n_2)$$

$$mul\llbracket m \rrbracket 0 = 0$$

There is no reason why we should not identify  $mul\llbracket m \rrbracket$  with  $m$  itself. In general, let  $\Omega$  be a set of elements such that for every  $\omega \in \Omega$ ,  $\omega$  is an endomorphism of the monoid  $(M, \oplus, u)$ . We will write this is the form

$$\omega(m_1 \oplus m_2) = \omega m_1 \oplus \omega m_2$$

$$\omega u = u$$

Then  $\Omega$  is called a set of operators for  $(M, \oplus, u)$  and the corresponding monoid with operators is denoted  $(\Omega, (M, \oplus, u))$ . Formally, in analogy to the definition of a group with operators (Papy 1964, 152), I give the following:

DEFINITION 3.10. *Every monoid  $(M, \oplus, u)$  provided with an outer law:*

$$\Omega \times M \longrightarrow M : (\omega, m) \mapsto \omega m$$

*which is distributive with respect to  $\oplus$ , is called a monoid with operators and denoted  $(\Omega, (M, \oplus, u))$ .*

The relationship between the monoid,  $(M, \oplus, u)$ , and the corresponding monoid with operators,  $(\Omega, (M, \oplus, u))$ , may be summed up by the following commuting diagram

$$\begin{array}{ccc} (m, n) & \xrightarrow{\oplus} & m \oplus n \\ \downarrow \omega \times \omega & & \downarrow \omega \\ (\omega m, \omega n) & \xrightarrow{\oplus} & \omega m \oplus \omega n \end{array}$$

Examples of such structures abound, both in mathematics and in formal specifications. Moreover, they are very familiar structures. We have already seen that the set of natural numbers is essentially a monoid with itself as the set of operators, multiplication being the operation in question. All that is being said is simply that multiplication distributes over addition. It may be denoted by  $(\mathbf{N}_1, (\mathbf{N}, +, 0))$ .

## MONOIDS AND HOMOMORPHISMS

Let us now consider the restriction and removal operators on maps.

LEMMA 3.8. *The restriction of a map  $\mu$  with respect to a set  $S$ , denoted  $\mu \upharpoonright S$  (Danish style) or  $S \triangleleft \mu$  (English style) is a monoid endomorphism. Formally,  $(\mathcal{P}X, (X \xrightarrow{m} Y, +, \theta))$  is a monoid with operators where the outer law is restriction.*

$$\begin{aligned} S \triangleleft (\mu_1 + \mu_2) &= (S \triangleleft \mu_1) + (S \triangleleft \mu_2) \\ S \triangleleft \theta &= \theta \end{aligned}$$

We also have  $S \triangleleft (\mu_1 \cup \mu_2) = (S \triangleleft \mu_1) \cup (S \triangleleft \mu_2)$  and  $S \triangleleft (\mu_1 \triangle \mu_2) = (S \triangleleft \mu_1) \triangle (S \triangleleft \mu_2)$ . I find it convenient to denote the operator  $S$  by  $\triangleleft \llbracket S \rrbracket$  or  $\triangleleft_S$ . Since it is an endomorphism, then the operator  $dom$  may subsequently be applied to give

$$dom(S \triangleleft \mu) = S \triangleleft dom \mu$$

LEMMA 3.9. *The removal of a map  $\mu$  with respect to a set  $S$ , denoted  $\mu \setminus S$  or  $\mu - S$  (Danish style), or  $S \triangleleft \mu$  (English style) is a monoid endomorphism. Formally,  $(\mathcal{P}X, (X \xrightarrow{m} Y, +, \theta))$  is a monoid with operators where the outer law is removal.*

$$\begin{aligned} S \triangleleft (\mu_1 + \mu_2) &= (S \triangleleft \mu_1) + (S \triangleleft \mu_2) \\ S \triangleleft \theta &= \theta \end{aligned}$$

Similar remarks apply to the removal operator. Again, we have

$$dom(S \triangleleft \mu) = S \triangleleft dom \mu$$

Note that in these two lemmas one has the same structure but with a different outer law. The implication of this will be made clear below. Now we may consider the meaning of the override operator,  $\mu_1 + \mu_2 \triangleq (dom \mu_2 \triangleleft \mu_1) \cup \mu_2$ , in a new light, especially when written in the form

$$\mu_1 + \mu_2 \triangleq (dom \mu_2 \triangleleft \mu_1) \cup (dom \mu_2 \triangleleft \mu_2)$$

Looking back on the other VDM domains of sequence and powerset, we note that there are more examples of monoids with operators.

EXAMPLE 3.6. The structure  $(\mathcal{P}X, (\mathcal{P}X, \cup, \emptyset))$  is a monoid with operators where set difference is the outer law.

Let  $\omega, S_1, S_2$  be sets in  $\mathcal{P}X$ . Consider the set difference of  $S_1 \cup S_2$  with respect to  $\omega$ . This has the property that  $(S_1 \cup S_2) \setminus \omega = (S_1 \setminus \omega) \cup (S_2 \setminus \omega)$ . If we rearrange

the order of the terms and denote set difference by the operator ‘ $\cdot$ ’, then we have

$$\omega \cdot (S_1 \cup S_2) = (\omega \cdot S_1) \cup (\omega \cdot S_2)$$

It is for precisely this reason that I have chosen to use  $\cdot$  as the set difference operator in the Irish *Meta-IV*. For example, the dictionary  $DICT_0$  is a monoid with operators  $(DICT_0, (DICT_0, \circ/Ent_0, New_0))$ , where the outer law is  $\circ/Rem_0$ . For similar reasons I have chosen  $\cdot$  and  $\cdot$  in place of the more conventional mathematical symbols  $|$  and  $\setminus$ , respectively.

EXAMPLE 3.7. The structure  $(\mathcal{P}X, (\mathcal{P}X, \Delta, \emptyset))$  is a monoid with operators where the outer law is set intersection.

EXAMPLE 3.8. The free monoid is a monoid with operators  $(\mathcal{P}\Sigma, (\Sigma^*, \wedge, \Lambda))$  where the outer law is deletion.

Rather than include a specific section related to maps to conclude this material, I have chosen to say a little more about the algebra of monoids with operators. There is sufficient material on maps in the remainder of the thesis.

### 3.4. Monoids of Endomorphisms

Given a monoid with operators (i.e., endomorphisms), and an appropriate outer law, it is natural to ask whether the set of operators has itself got some structure. In other words, do such operators combine and, if so, in what way? In general, as will be studied in this section, the answer is yes.

Consider the monoid with operators  $(\mathbf{N}_1, (\mathbf{N}, +, 0))$ . The set of operators  $\mathbf{N}_1$  is itself a monoid of endomorphisms, denoted  $(\mathbf{N}_1, \times, 1)$ . The unit element 1 is the identity endomorphism

$$1.m = m$$

The composition of two endomorphisms  $n_1$  and  $n_2$  is given by

$$(n_1 \times n_2).m = n_1.(n_2.m)$$

Such a monoid with operators is called a semiring, which in the above case is denoted by  $(\mathbf{N}, +, \times, 0, 1)$ . Formally, a semiring is a monoid with operators such that the set of operators, i.e., endomorphisms, forms a monoid under the law of composition of morphisms. We now have other examples of semirings.

## MONOIDS AND HOMOMORPHISMS

EXAMPLE 3.9. Let us look at the monoid with operators  $(\mathcal{P}X, (X \xrightarrow{m} Y, +, \theta))$  where the outer law is restriction. Then, the set of operators  $\mathcal{P}X$  is a monoid of endomorphisms  $(\mathcal{P}X, \cap, X)$ . The set  $X$  is the identity endomorphism

$$X \triangleleft \mu = \mu$$

The composition of two endomorphisms  $S_1$  and  $S_2$  is given by

$$(S_1 \cap S_2) \triangleleft \mu = S_1 \triangleleft (S_2 \triangleleft \mu)$$

EXAMPLE 3.10. Consider the monoid with operators  $(\mathcal{P}X, (X \xrightarrow{m} Y, +, \theta))$  where the outer law is removal. Then the set of operators  $\mathcal{P}X$  is a monoid of endomorphisms denoted  $(\mathcal{P}X, \cup, \emptyset)$ . The identity endomorphism is the empty set  $\emptyset$ .

$$\emptyset \triangleleft \mu = \mu$$

The composition of two endomorphisms  $S_1$  and  $S_2$  is defined by

$$(S_1 \cup S_2) \triangleleft \mu = S_1 \triangleleft (S_2 \triangleleft \mu)$$

We have already seen that the above two monoids of endomorphisms are isomorphic.

EXAMPLE 3.11. The structure  $(\mathcal{P}X, (\mathcal{P}X, \Delta, \emptyset))$  is a monoid with operators where the outer law is set intersection. The set of operators  $\mathcal{P}X$  is a monoid of endomorphisms  $(\mathcal{P}X, \cap, X)$ . The identity endomorphism is  $X$ .

$$X \Delta S = S$$

The composition of two endomorphisms  $S_1$  and  $S_2$  is defined by

$$(S_1 \cap S_2) \Delta S = S_1 \Delta (S_2 \Delta S)$$

Since  $(\mathcal{P}X, \Delta)$  is a group, the structure is in fact a ring of sets,  $(\mathcal{P}X, \Delta, \cap, \emptyset, X)$ .

Consider the monoid with operators  $(\Sigma, (\Sigma^*, \wedge, \Lambda))$  which was obtained as a result of considering the filter algorithm. The set of operators,  $\Sigma$ , in the monoid with operators,  $(\Sigma, (\Sigma^*, \wedge, \Lambda))$ , does not itself have an inner law. By generalising the outer law of deletion with respect to an element, given constructively by the ‘delall’ algorithm, to deletion with respect to a set of elements, one obtains the monoid with operators  $(\mathcal{P}\Sigma, (\Sigma^*, \wedge, \Lambda))$ . Thus,  $\forall S \in \mathcal{P}\Sigma$ , one has

$$S \triangleleft (\sigma \wedge \tau) = (S \triangleleft \sigma) \wedge (S \triangleleft \tau)$$

$$S \triangleleft \Lambda = \Lambda$$

The set of operators (endomorphisms),  $\mathcal{P}\Sigma$ , forms a monoid where the inner law is set union  $\cup$ , denoted  $(\mathcal{P}\Sigma, \cup, \emptyset)$ . The empty set  $\emptyset$  is the identity endomorphism

$$\emptyset \triangleleft \sigma = \sigma$$

The composition of two endomorphisms  $S_1$  and  $S_2$  is given by

$$(S_1 \cup S_2) \triangleleft \sigma = S_1 \triangleleft (S_2 \triangleleft \sigma)$$

This monoid of endomorphisms has already appeared with respect to map removal. In general, restriction and removal operators are applicable to all *VDM* domains. In the case of removal, I have made this obvious by using the map removal operator symbol  $\triangleleft$  for set difference, and deletion of all occurrences of an element from a sequence. The restriction operator is the dual of the removal operator. Specifically, any specification written in terms of the removal operator may be transformed into one which uses the restriction operator, and vice-versa. For sets, the corresponding operator is the intersection operator. We have not identified a corresponding operator for sequences, for which an algorithmic definition is now provided

$$S \triangleleft (\langle e \rangle \wedge \tau) = \begin{cases} \langle e \rangle \wedge (S \triangleleft \tau), & \text{if } \chi[[e]]S; \\ S \triangleleft \tau, & \text{otherwise.} \end{cases}$$

$$\emptyset \triangleleft \sigma = \Lambda$$

Since a sequence is a particular form of a map, then the notation  $S \triangleleft \sigma$  is justified. However, the above definition is more explicit.

We have now clearly separated the *VDM* operators into well-defined classes. Operators such as set intersection, sequence concatenation, and map override are essentially construction operators which give us corresponding monoids. Then there are the classes of the dual monoid endomorphisms: the removal and restriction operators. Many of the other operators are homomorphisms of one sort or another. Among those which are not homomorphisms, I note the cardinality of a set, the head and tail operators of sequences, and the range operator of maps. To put this into the practical context of formal specifications, consider once more the simple abstract model of the spelling-checker dictionary as a set of words. The enter operation corresponds to set union which, together with the create operation, gives us the (abstract) monoid  $(\mathcal{P}X, \cup, \emptyset)$ . The remove operation is a monoid endomorphism. We did not need the intersection operator (i.e., the restriction operator) which is the dual of the remove operation. The lookup operation is a monoid epimorphism and

## MONOIDS AND HOMOMORPHISMS

the size operation is, of course, the ‘odd one out’. When we come to deal with the issue of proofs, we will see that much of the hard work is primarily concerned with enter and remove operations. Since enter operations give us the actual algebraic structure of the models in question, I would like to conclude with a little more material on the nature of remove operations, i.e., on the nature of removal and restriction endomorphisms.

### 3.5. Admissible Submonoids

The following material I have adapted from (Papy 1964, 157) and applied it to the theory of monoids. Let  $(\Omega, (M, \oplus, u))$  be a monoid with operators. Every submonoid  $S$  of  $M$  which is itself a monoid with operators  $(\Omega, (S, \oplus, u))$  will be called admissible. Note that the set of operators  $\Omega$  is identical in both structures.

**DEFINITION 3.11.** *The submonoid  $S$  of  $M$  is admissible iff  $\forall \omega \in \Omega$ , and  $\forall s \in S$  then  $\omega s \in S$ .*

**EXAMPLE 3.12.** Recall that the set of even numbers,  $E = \{2n \mid n \in \mathbf{N}\}$ , is a submonoid of  $(\mathbf{N}, +, 0)$ . Since,  $\forall m \in \mathbf{N}_1, \forall e \in E$  then  $m \times e \in E$ . Therefore,  $(\mathbf{N}_1, (E, +, 0))$  is admissible.

**EXAMPLE 3.13.** Let  $T$  be a subset of an alphabet  $\Sigma$ . Then the freemonoid  $(T^*, \wedge, \Lambda)$  is a submonoid of  $(\Sigma^*, \wedge, \Lambda)$ . Clearly, both  $(\Sigma, (T^*, \wedge, \Lambda))$  and  $(\mathcal{P}\Sigma, (T^*, \wedge, \Lambda))$  are admissible, where the outer law is the *delall* algorithm in both cases.

Formally

$$\begin{aligned} \forall e \in \Sigma, \quad \forall \tau \in T, \quad e \leftarrow \tau \in T \\ \forall s \in \mathcal{P}\Sigma, \quad \forall \tau \in T, \quad s \leftarrow \tau \in T \end{aligned}$$

Let  $(\Omega, (M, \oplus, u))$  and  $(\Omega, (M', \oplus', u'))$  be two monoids with the same set of operators,  $\Omega$ . Then a monoid homomorphism  $h: (M, \oplus, u) \longrightarrow (M', \oplus', u')$  which commutes with the operators will be called admissible, i.e., such that:

$$\forall \omega \in \Omega, \quad \forall m \in M, \quad h(\omega m) = \omega h(m)$$

Such homomorphisms are also called  $\Omega$ -homomorphisms. We are now ready to consider the nature of restriction and removal operators.

EXAMPLE 3.14. The  $\Sigma^*$ -homomorphism  $elems$  is an admissible homomorphism:

$$elems(S \triangleleft \tau) = S \triangleleft (elems \tau)$$

with signature

$$elems: (\mathcal{P}\Sigma, (\Sigma^*, \wedge, \Lambda)) \longrightarrow (\mathcal{P}\Sigma, (\mathcal{P}\Sigma, \cup, \emptyset))$$

where the outer law is deletion! Note that it is only because the *delall* algorithm applied to sequences is the exact equivalent of deletion for sets that  $elems$  is admissible.

Were one to implement sets using sequences, there is a need for some algorithm which would remove duplicates. I called this algorithm the filter algorithm. Recall that the *filter* algorithm led to two different  $\Sigma^*$ -homomorphisms,  $filter_1: (\Sigma^*, \wedge, \Lambda) \longrightarrow (\Sigma_!^*, \diamond, \Lambda)$ , where

$$filter_1(\langle e \rangle \wedge \tau) = \chi[e]\tau(filter_1(\tau), \langle e \rangle \wedge filter_1(\tau))$$

and,  $filter_2: (\Sigma^*, \wedge, \Lambda) \longrightarrow (\Sigma^*, \diamond', \Lambda)$ , where

$$filter_1(\langle e \rangle \wedge \tau) = \langle e \rangle \wedge filter_1(\langle e \rangle \triangleleft \tau)$$

In order to obtain this latter result, I had to define the  $\diamond'$  operator:

$$(\langle e \rangle \wedge \sigma) \diamond' \tau = \langle e \rangle \wedge ((e \triangleleft \sigma) \diamond' (e \triangleleft \tau))$$

Then, since  $filter_2(e \triangleleft \tau) = e \triangleleft filter_2(\tau)$ ,  $filter_2$  is admissible with signature

$$filter_2: (\mathcal{P}\Sigma, (\Sigma^*, \wedge, \Lambda)) \longrightarrow (\mathcal{P}\Sigma, (\Sigma_!^*, \diamond, \Lambda))$$

where again the outer law is the *delall* algorithm. From these two examples it is clear that  $filter_2$  is an exact algorithmic equivalent of  $elems$ .

In the case of maps, we have the equalities

$$dom(S \triangleleft \mu) = S \triangleleft dom \mu$$

$$dom(S \triangleleft \mu) = S \triangleleft dom \mu$$

which are likely to turn up in proofs. Note that I have used the restriction operator  $\triangleleft$  on the right hand side of the latter equation instead of the more usual intersection operator. Since  $(\mathcal{P}X, (X \xrightarrow{m} Y, +, \theta))$  is a monoid with operators and  $dom$  is a homomorphism of  $(X \xrightarrow{m} Y, +, \theta)$ , then clearly  $dom$  is an admissible homomorphism.

# MONOIDS AND HOMOMORPHISMS

## 4. Summary

To have been able to find an æsthetic algebraic framework in which to set many apparently disparate mundane facts concerning *VDM* specifications and to exhibit clearly the underlying unifying connections in terms of monoids and their homomorphisms has been an intensely satisfying experience. That said framework also brings order and structure to the complexity of specifications remains to be demonstrated in the remainder of the thesis.

The phrase ‘monoids with operators’ points to my goal of building an operator calculus for the Irish School of the *VDM*. I eschewed the currently conventional focus on term algebras and returned to the more basic concept of the monoid. I suppose that the use of the term operator to describe operations on the basic *VDM* domains triggered my research direction. Gradually, the realisation dawned that *operations* on the models that occur in specifications might actually be thought of as operators. Thus, in the case of the spelling-checker dictionary, the enter operation is conceptually the set union operator, the inner law of a monoid—the dictionary. The deletion operation becomes a monoid endomorphism, an operator. An immediate consequence of this *Gestalt* shift was the heavy emphasis on the particular form of the notation to be used in specifications. For example, the expression  $\delta \cup \{w\}$  took on more significance than  $Ent[[w]]\delta$ . Ultimately I arrived at the conclusion that when one speaks of structuring specifications, one ought to mean identifying algebraic structure and *not* some sort of syntactic sugar that might facilitate the use of tool support. The real difficulty in large complex specifications is more a conceptual issue than a syntactic issue.

Having set down the foundation for the Irish School of the *VDM*, it then became a matter of concern to ‘exercise’ the operator calculus. I found it absolutely essential to rework existing published specifications to test the appropriateness of the approach and this I have done with considerable success. Another important issue that had to be dealt with urgently was to provide enough material to exhibit the full thrust and importance of the approach. Since I firmly believe that doing specifications and programming are two sides of the same constructive mathematical activity, I now present a rudimentary theory of tail-recursive forms of algorithms.

# Chapter 4

## Tail-Recursive Algorithms

### 1. Introduction

“More than half a century has passed since the famous papers of Gödel 1931 and Turing 1936-7 that shed so much light on the foundations of mathematics, and that simultaneously promulgated mathematical formalisms for specifying algorithms, in one case via primitive recursive function definitions, and in the other case via Turing machines [...] as a result we now know much better how to do the high-level functional programming of Gödel, and how to do the low-level programming found in Turing’s paper” (Chaitin 1988, 279).

In Chapter 2, I gave recursive definitions of basic *VDM Meta-IV* operators and simple algorithms. In the previous Chapter, I further demonstrated that the *Meta-IV* domains may be considered as monoids, and that homomorphisms of such monoids gave rise to what I have termed naïve recursive algorithms or definitions. I wish to continue to treat of the nature of recursive algorithm in this Chapter. But my concern is always of a conceptual nature. There seems to me to be abundant material on the Theory of Recursive Functions. The rôle that the theory plays or at least ought to play in the conceptual models of those who specify systems and algorithms does not seem to be adequately developed. That is not to say that authors who elaborate on the Theory of Recursive Functions have not sought to develop such conceptual models. For indeed, all who write on functional programming strive to do just that. But such material focuses on ‘programming’, which term triggers connotations of ‘time’ and ‘space’ issues. Moreover, advocates of functional programming have to contend with the mindset of ‘Turing’ programmers, who seem at times to view recursion as something inherently inefficient.

It is my primary concern in this Chapter to demonstrate that recursive al-

## TAIL-RECURSIVE ALGORITHMS

gorithms (functions) are everywhere abundant; that they are used extensively in Mathematics, though sometimes going by different names such as difference equations, recurrence relations, etc.; that they have natural tail-recursive equivalents, which may be directly transcribed as while loops; and that they are much better for reasoning about correctness than equivalent programs. Indeed, such recursive algorithms are ideally suited for the purpose of doing constructive mathematics. But, whereas the functional programming approach binds one with inflexible syntax, and on the other hand, the Theory of Recursive Functions is too much bound up with foundational issues, I will demonstrate that the *Meta-IV* approach is on the middle ground.

Before delving into details, it is worthwhile to recall some classical recursive functions that have been used to caution us in their use in programming. First there is McCarthy's 91-function (Brady 1977, 14)

$$f(x) \triangleq x > 100 \rightarrow x - 10, f(f(x + 11))$$

which is equivalent to

$$g(x) \triangleq x > 100 \rightarrow x - 10, 91$$

From this we are supposed to learn the lesson that having obtained a recursive definition, does not mean that we ought to stop there, nor indeed foolishly suppose that one ought to encode directly from a recursive definition once obtained. Second, there are recursive functions such as the Ackerman Function (Ackermann [1928] 1967, 493) which need to be massaged should one wish to compute them, for whatever reason. The definition given here, taken from Hermes (1969, 84), is *not* the original function of Ackermann but adequately illustrates the point:

$$\begin{aligned} f(0, y) &= y + 1 \\ f(x, 0) &= f(x - 1, 1) \\ f(x, y) &= f(x - 1, f(x, y - 1)) \end{aligned}$$

I have taken the liberty of slightly modifying the notation. Theoretically, Ackermann's function demonstrates that there exist computable functions which can not be obtained by the primitive recursive functions. The nesting of  $f$  may be removed by currying:

$$\begin{aligned}
f[0]y &= y + 1 \\
f[x + 1]0 &= f[x]1 \\
f[x + 1](y + 1) &= f[x] \circ f[x + 1]y
\end{aligned}$$

One will recognise the similarity between this form and that of the parts-explosion algorithm for the bill of material to be discussed in Chapter 5.

### 1.1. Conceptual Errors

It is held by many that recursion is inherently inefficient! Such a myth must be rejected absolutely (c.f., the similar position of Jacques Arzac (1985, 44)). An example often cited in support of such a position is the computation of a Fibonacci number (Ralston 1971, 346; Dromey 1982, 371). Such a position is erroneous! Curiously, it is the Fibonacci function that is chosen to make the false claim and not the more interesting Ackermann function! It all depends on the **form** of the recursive definition. There is, of course, an obvious recursive form for the Fibonacci which is very inefficient. I designate this the naïve recursive form. However, there is also an efficient recursive form that is derivable from it, a form which does not seem to be known generally. The case of the Fibonacci number computation is discussed at some length below.

It also seems to be held by many that there is an inherent difference between recursive algorithms and iterative algorithms. This is also an erroneous position! The conceptual error is probably due to the simple fact that people think of iteration in terms of ‘for loops’ and ‘while loops’. That this should be so is partly due to the association between iteration and indexing. Indeed, there is a very close relationship between recursion and iteration. Consider, for example, the remarks by Peter Henrici on the solution of (non-linear) fixed-point equations that the method of iteration consists in the recursive generation of a sequence of approximations (Henrici 1964, 61). Such fixed-point algorithms are considered below. In addition, the denotational semantics of ‘while loops’ is given in recursive form!

Finally, it ought to be noted that programmers often seem to confuse the terms iterative and imperative, assuming that iterative algorithms must be imperative. That iterative algorithms are usually encoded imperatively is probably the cause behind the confusion. A selection of iterative algorithms in recursive form is presented to dispel such erroneous notions. Imperative encodings have their own par-

## TAIL-RECURSIVE ALGORITHMS

ticular elegance. They are the engineered solutions to problems in computing where the resource constraint has been the actual architecture of von Neuman computers. Imperative programming languages encapsulate in their syntax, the structure and behaviour of such machines—their semantics may be operational, in terms of the von Neuman architecture, or denotational, where recursive forms are paramount.

Recursive forms of algorithms are often preferred precisely because correctness arguments for some are more easily established than comparable imperative algorithms. However, in engineering solutions, it is often a requirement to derive imperative forms from recursive forms. That such derivations preserve correctness is crucial, a concern that has been addressed by many researchers. In this context the work of Arzac (1985) and Burstall and Darlington (1977) is to be noted. The former emphasises the notion of **method** of transformation, whereas the latter tends to concentrate on transformation schemas. Note the application of transformation schemas by McGettrick to Ada (1982, 275–311). In summary then, three *conceptual errors* have been identified:

- (1) ‘recursive algorithms are inherently inefficient’;
- (2) ‘there is an inherent difference between recursive algorithms and iterative algorithms’;
- (3) ‘iterative algorithms are synonymous with imperative algorithms’.

### 1.2. A Selection of ‘Error-correcting’ Algorithms

It is important that the conceptual errors highlighted in the previous section be corrected at once. For this purpose a few algorithms are selected, all of which have efficient recursive forms, in order to demonstrate the points made by counter-example. For each of the errors cited, the following algorithms are offered:

- ‘recursive algorithms are inherently inefficient’: an efficient recursive algorithm for the Fibonacci numbers is presented;
- ‘there is an inherent distinction between recursive algorithms and iterative algorithms’: a class of fixed-point iterative algorithms from numerical analysis is presented;
- ‘iterative algorithms are the same as imperative algorithms’: the algorithm to compute the sine of an angle (in radians) is offered as a counter-example.

## 1.2.1. The Fibonacci Sequence

The usual recurrence relation for the definition of Fibonacci numbers is

$$F_0 = 1$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}, \quad n > 1$$

From this a simple naïve recursive algorithm may be developed, which is then shown to be very inefficient. The computation of such an algorithm may be expressed in formal grammar notation—using a leftmost derivation of a terminal string from a non-terminal using production rules. The number 1 and the operator + are the terminals. The symbols  $F_n, \forall n \in \mathbf{N}$  constitute the set of non-terminals. Finally, the equations above may be interpreted as production rules. The number to be computed is then the start symbol. For example, the computation of  $F_5$  gives:

$$\begin{aligned} F_5 &\Longrightarrow F_4 + F_3 \\ &\Longrightarrow F_3 + F_2 + F_3 \\ &\Longrightarrow F_2 + F_1 + F_2 + F_3 \\ &\Longrightarrow F_1 + F_0 + F_1 + F_2 + F_3 \\ &\Longrightarrow 1 + F_0 + F_1 + F_2 + F_3 \\ &\Longrightarrow 1 + 1 + F_1 + F_2 + F_3 \\ &\Longrightarrow 1 + 1 + 1 + F_2 + F_3 \\ &\Longrightarrow 1 + 1 + 1 + F_1 + F_0 + F_3 \\ &\Longrightarrow \dots \end{aligned}$$

Clearly there is much repetitious computation. Now I have chosen to exhibit the computation as a sequence of rewrites to highlight the conceptual misunderstanding of those who see the recursive definition as *the* algorithm for the Fibonacci numbers. It is a simple fact that useful formal languages may be generated if and only if we have recursive rewrite rules and there different grammars which generate the same language. But no-one claims that such rewrite rules are hopelessly inefficient. The whole Theory of Parsing, which embraces the study of formal grammars, is dedicated to ‘handling’ the recursion.

There is a recursive algorithm for the Fibonacci numbers which is very efficient and which is directly derivable from the above. The algorithm given here is taken from Arzac’s work on the Foundations of Programming (1985, 41). The key ‘eureka’

## TAIL-RECURSIVE ALGORITHMS

step (the term is taken from Burstall and Darlington (1977)) is to define a function  $G_n$  to be the pair of functions  $(F_n, F_{n-1})$ :

$$G_n = (F_n, F_{n-1})$$

Since  $G_0$  is not defined, one uses  $G_1$  to establish the base case:

$$G_1 = (F_1, F_0) = (1, 1)$$

Expanding the definition of  $F_n$  gives

$$\begin{aligned} G_n &= (F_n, F_{n-1}) \\ &= (F_{n-1} + F_{n-2}, F_{n-1}) \end{aligned}$$

and since

$$G_{n-1} = (F_{n-1}, F_{n-2})$$

then we note that  $G_n$  is expressed in the same terms as  $G_{n-1}$ . Specifically, we may write  $F_{n-1} = \pi_1 G_{n-1}$  and  $F_{n-2} = \pi_2 G_{n-1}$ , where  $\pi_1$  and  $\pi_2$  are the usual projection functions that select the first and second component of a cartesian product of order 2, respectively. Therefore

$$G_n = (\pi_1 G_{n-1} + \pi_2 G_{n-1}, \pi_1 G_{n-1})$$

Allowing ourselves a slight abuse of notation, permits us to factor out  $G_{n-1}$ , giving

$$G_n = (\pi_1 + \pi_2, \pi_1) G_{n-1}$$

Hence, the required recursive algorithm is

$$\begin{aligned} G(n) &\triangleq \text{if } n = 1 \text{ then } (1, 1) \\ &\quad \text{else let } (x, y) = G(n - 1) \\ &\quad \quad \text{in } (x + y, x) \end{aligned}$$

with  $F_n = \pi_1 G_n$ . In the Irish School of the VDM I recommend that one also explore the presentation of a solution. In the example in question I wish to eliminate the ‘let’ construct. I have found from experience that performing a test computation, such as computing  $F_5$ , often brings insight. A pattern of computation immediately emerges which leads directly to the following equivalent tail-recursive form

$$\begin{aligned} G: \mathbf{N}_1 &\longrightarrow \mathbf{N}_1^2 \longrightarrow \mathbf{N}_1^2 \\ G[[n]](x, y) &\triangleq (n = 1) \rightarrow (x, y), G[[n - 1]](x + y, x) \end{aligned}$$

with initialisation,  $G(n) = G[[n]](1, 1)$ . From this an equivalent while loop program may be produced:

```

(x, y) ← (1, 1);
while n ≠ 1 do
  (x, y) ← (x + y, x);
  n ← n - 1;
end while;
return (x, y)

```

which is the ‘usual’ imperative solution to the problem. For the purposes of comparison, I present here a similar tail-recursive form for the computation of the greatest common divisor (gcd) of two natural numbers, adopted from (Knuth 1981, 2:320):

$$G: \mathbf{N}^2 \longrightarrow \mathbf{N}^2$$

$$G[-](u, v) \triangleq (v = 0) \rightarrow (u, v), G[-](v, u \bmod v)$$

with initialisation  $\text{gcd}(u, v) = \pi_1 G[-](u, v)$ . The two forms are very similar. I make but one observation. In the latter case, to illustrate the structural similarity between the two algorithms, I have chosen to use a null argument. Its purpose is very similar to the sort of ‘trick’ one uses in mathematics:  $5x^3 - 11 = 5 \cdot x^3 + 0 \cdot x^2 + 0 \cdot x^1 - 11 \cdot x^0$ , to make explicit that which is ‘invisible’.

### 1.2.2. Fixed-point Computation

Consider the problem of solving the equations of the form  $x = f(x)$  where  $f$  is a given function (Henrici 1964, 61). The iterative method consists in choosing a starting value  $x_0$  and then generating a sequence of approximations  $x_1, x_2, \dots$ , from the recurrence relation

$$x_{n+1} = f(x_n), \quad n \geq 0$$

It is not of interest in this work to consider important issues such as well-definedness of the algorithm from the perspective of numerical analysis. Henrici’s work, cited above, gives insight into such matters. What is of interest is the fact that such an iterative algorithm can be expressed very efficiently as a recursive algorithm:

$$\text{fixed\_point}(f, \epsilon, x_{n-1}, x_n) \triangleq \begin{cases} \text{if } |x_n - x_{n-1}| < \epsilon \\ \text{then } x_n \\ \text{else } \text{fixed\_point}(f, \epsilon, x_n, f(x_n)) \end{cases}$$

where  $\epsilon$  is a very small positive number and the terminating condition  $|x_n - x_{n-1}| < \epsilon$  is dictated by a Lipschitz condition.

## TAIL-RECURSIVE ALGORITHMS

It is to be noted that the recurrence relation  $x_n = f(x_{n-1})$  actually covers a wide range of numerical methods. For example, if one defines

$$f(x) = x - \frac{F(x)}{F'(x)}$$

for a differentiable function  $F$ , then Newton's method is obtained.

### 1.2.3. Fixed-point Computation in PROLOG

To illustrate the encoding of a fixed-point algorithm in PROLOG, consider the problem of the solution of Kepler's equation taken from Henrici (1964):

$$m = x - E \sin x$$

Given  $m = 0.8$ ,  $E = 0.2$ , we are required to compute an approximation to  $x$ . The computation may be expressed in fixed point form:

$$x_n = f(x_{n-1})$$

$$x_0 = m$$

where  $f(x) = m + E \sin x$ . First, the generic fixed-point computation may be encoded as

```
fixed_point(F, Epsilon, Xold, Xnew, Xnew) :-  
    Diff is Xnew - Xold,  
    abs(Diff, Absval),  
    Absval < Epsilon.  
fixed_point(F, Epsilon, Xold, Xnew, Result) :-  
    Function = .. [F, Xnew, Val],  
    call(Function),  
    fixed_point(F, Epsilon, Xnew, Val, Result).
```

where

```
abs(X, Y) :-  
    X < 0.0,  
    Y is - X.  
abs(X, X).
```

Then the expression  $m + E \sin x$  is encoded as the predicate

```
kepler(M, E, X, Val) :-  
    Val is M + E * sin(X).
```

and the function  $f$  encoded as

$f(X, Val) :- kepler(0.8, 0.2, X, Val).$

Finally, the computation is triggered by

?- *fixed\_point(f, 0.001, 0.0, 0.8, Result).*

where 0.0 is an arbitrary constant, 0.8 is the first approximation. The computed result is 0.964302.

#### 1.2.4. Infinite Sums

Consider the definition of the sinus function (*sin*) which is formally given as an infinite sum:

$$\sin x \triangleq \sum_{k=1}^{\infty} (-1)^{k+1} \frac{x^{2k-1}}{(2k-1)!}$$

Suppose that one is required to compute an approximation to  $\sin x$  directly from the definition. A natural approach is to consider the solution to be already given by a sequence of partial sums:

$$S = \langle s_1, s_2, \dots, s_{n-1}, s_n, \dots \rangle$$

where

$$\begin{aligned} s_1 &= x \\ s_2 &= x - \frac{x^3}{3!} \\ &\dots \\ s_n &= s_{n-1} + (-1)^{n+1} \frac{x^{2n-1}}{(2n-1)!} \\ &\dots \end{aligned}$$

Then, one might accept the  $n$ th element of the sequence as the appropriate approximation, for some appropriate value of  $n$ . One ought not to be surprised that it is possible to obtain a dreadfully inefficient algorithm (whether recursive or imperative) directly from these considerations. The naïve recursive solution is obviously:

$$\begin{aligned} \sin(x, \epsilon, s_n, t_n) &\triangleq \text{if } \text{abs}(t_n) < \epsilon \text{ then } s_n \\ &\quad \text{else let } t_{n+1} = (-1)^{n+3} \frac{x^{2n+1}}{(2n+1)!} \\ &\quad \text{in } \sin(x, \epsilon, s_n + t_{n+1}, t_{n+1}) \end{aligned}$$

This is a tail-recursive algorithm. However, computation of the expression

$$t_{n+1} = (-1)^{n+3} \frac{x^{2n+1}}{(2n+1)!}$$

## TAIL-RECURSIVE ALGORITHMS

is the source of the inefficiency. The ‘eureka’ step is, of course, to recognise that the  $n$ th term  $t_n$  may be derived from the  $(n - 1)$ th term  $t_{n-1}$ . In other words, the sequence of terms

$$T = \langle t_1, t_2, \dots, t_{n-1}, t_n, \dots \rangle$$

is given by the generating function

$$t_n = \frac{-x^2}{(2n - 1)(2n - 2)} t_{n-1}$$

and, thus, the efficient tail-recursive algorithm is given by

$$\begin{aligned} \text{sin}(x, \epsilon, s_n, t_n) &\triangleq \text{if } \text{abs}(t_n) < \epsilon \text{ then } s_n \\ &\text{else let } t_{n+1} = \frac{-x^2}{(2n - 1)(2n - 2)} t_n \\ &\text{in } \text{sin}(x, \epsilon, s_n + t_{n+1}, t_{n+1}) \end{aligned}$$

### 1.3. The Generic Solution

The concept of sequence (as expressed in the *VDM Meta-IV*) provides a kernel conceptual framework in which all of the above problems may be set. In particular, all of the data and solutions are basically of the form  $\langle a_1, a_2, \dots, a_n, \dots \rangle$  where there is the ‘promise’ of a possibly infinite computation. In such cases, the preferred expression is in reverse form where the first element in the sequence is the latest computed result! Incidentally, this is also a nice model for the interpretation of assignment in imperative programming languages. Essentially, what follows is a formal specification of the solution to the problem which, in a sense, may be deemed to be generic.

For example, the Fibonacci number  $F_n$  is considered to be the first element of the sequence of Fibonacci numbers

$$\langle F_n, F_{n-1}, \dots, 5, 3, 2, 1, 1 \rangle$$

Let us define the Fibonacci number sequence as the domain

$$FIBNUMS = FIBNUM^*$$

Then, given some sequence of Fibonacci numbers, the next number in the sequence may be computed by

$$\begin{aligned} \text{next: } FIBNUMS &\longrightarrow FIBNUMS \\ \text{next}(\Lambda) &\triangleq \langle 1, 1 \rangle \\ \text{next}(\langle a, b \rangle \wedge \tau) &\triangleq \langle a + b \rangle \wedge \langle a, b \rangle \wedge \tau \end{aligned}$$

The important point to note is that no reference is made to the recurrence relations and that the notion of index is missing! In particular, an imperative solution is made clear! The next number in the sequence is obtained by adding the previous two numbers.

For fixed-point algorithms, it is convenient to think in terms of the domain

$$APPROXS = APPROX^*$$

Then the next approximation in the sequence is given by

$$\text{next}: (\mathbf{R} \xrightarrow{m} \mathbf{R}) \times \mathbf{R} \longrightarrow APPROXS \longrightarrow APPROXS$$

$$\begin{aligned} \text{next}[f, \epsilon]\Lambda &\triangleq \langle x_0 \rangle \\ \text{next}[f, \epsilon](\langle x \rangle \wedge \tau) &\triangleq \langle f(x) \rangle \wedge \langle x \rangle \wedge \tau \end{aligned}$$

Therefore, the computation of the fixed-point solution may be specified by

$$\text{fixed\_point}: (\mathbf{R} \xrightarrow{m} \mathbf{R}) \times \mathbf{R} \longrightarrow APPROXS \longrightarrow APPROX$$

$$\begin{aligned} \text{fixed\_point}[f, \epsilon]\langle x_n, x_{n-1} \rangle \wedge \tau &\triangleq \text{if } |x_n - x_{n-1}| < \epsilon \text{ then } x_n \\ &\quad \text{else let } xs = \text{next}[f, \epsilon]\langle x_n, x_{n-1} \rangle \wedge \tau \\ &\quad \text{in fixed\_point}[f, \epsilon]xs \end{aligned}$$

Turning now to the computation of the sinus function, one has the domain

$$PARTIALSUMS = PARTIALSUM^*$$

The next partial sum may be computed by

$$\begin{aligned} \text{next}: TERM \longrightarrow PARTIALSUMS \longrightarrow PARTIALSUMS \\ \text{next}[t]\Lambda &\triangleq \langle t \rangle \\ \text{next}[t](\langle s \rangle \wedge \tau) &\triangleq \langle s + t \rangle \wedge \langle s \rangle \wedge \tau \end{aligned}$$

where there is a sequence of terms, expressed by the domain

$$TERMS = TERM^*$$

and for which a next operation is also appropriate

$$\begin{aligned} \text{next}: \mathbf{R} \times \mathbf{N}_1 \longrightarrow TERMS \longrightarrow TERMS \\ \text{next}[x, 1]\Lambda &\triangleq \langle x \rangle \\ \text{next}[x, n](\langle t \rangle \wedge \tau) &\triangleq \langle \frac{-x^2}{(2n-1)(2n-2)}t \rangle \wedge \langle t \rangle \wedge \tau \end{aligned}$$

Again it is important to note that indexing plays no part in the consideration of the computation of the next partial sum. In the case of the computation of the next

## TAIL-RECURSIVE ALGORITHMS

term, the index must be explicitly included for the purpose of computing the value of  $(2n - 1)(2n - 2)$ .

I have exhibited some efficient tail-recursive algorithms to lay to rest some conceptual misunderstandings that still seem to be prevalent today. But the real point of this Chapter is, on the one hand, to continue the theoretical groundwork already laid down in Chapter 3 and to demonstrate that such theory has very practical consequences. I will show the correspondence between homomorphisms and tail-recursive forms and demonstrate that the latter enjoy elegant properties as much as the former.

### 2. Basic Numerical Algorithms

Before proceeding to discuss tail-recursive algorithms over the *VDM Meta-IV* domains, it seems to me to be appropriate to consider first some simple well known functions on natural numbers and thereby provide a basis for what is to follow. Those which I have chosen are (i) the exponential function—a homomorphism, (ii) the factorial function, and (iii) multiplication—an endomorphism.

#### 2.1. The Exponential Function

In this subsection I wish to exhibit clearly the distinction between curried functions and tail-recursive functions expressed in curried form. It may happen that mere currying of a function gives a tail-recursive form as exemplified by the *plus* function in Chapter 2. It is not always the case. Moreover, there is a clear distinction to be made between tail-recursive functions which are equivalent to while loop programs and the notion of efficient computation (in terms of time/space tradeoff). The exponential function *exp* on the natural numbers is ideal to bring out these distinctions. It may be defined by the recurrence relations:

$$\begin{aligned}n^0 &= 1 \\n^{i+1} &= n^i \times n, \quad i \in \mathbf{N}\end{aligned}$$

where, in general, one has the relation

$$n^{i+j} = n^i \times n^j$$

which suggests a homomorphism. But one might ask, of what is  $exp$  a homomorphism? Are we speaking about the monoid of natural numbers and, if so, what should be our stand with respect to  $n = 0$ ? From the point of view of the Theory of Recursive functions,  $exp$  is primitive recursive and in order that it fit into the usual pattern of such functions one may adopt the convention that  $0^0 = 1$  (Hermes 1969, 64; Péter 1967, 24). This seems to me to be very artificial. Indeed, even the case  $n^0 = 1$ ,  $n \neq 0$  also seems somewhat contrived, at least to the initiate. The exponential function, as defined above, may be given a very precise interpretation that justifies the latter and avoids the anomaly  $0^0$ :

**THEOREM 4.1.** *The exponential function  $exp$  is a homomorphism from the monoid with operators  $(\mathbf{N}_1, (\mathbf{N}, +, 0))$  onto the monoid  $(\mathbf{N}_1, \times, 1)$ .*

Formally

$$\begin{aligned} exp(n, i + j) &= exp(n, i) \times exp(n, j) \\ exp(n, 0) &= 1 \end{aligned}$$

with signature

$$exp: (\mathbf{N}_1, (\mathbf{N}, +, 0)) \longrightarrow (\mathbf{N}_1, \times, 1)$$

Recall that this form of definition of homomorphism does not immediately give a constructive algorithm. In order to compute  $exp$  it is necessary to establish the constructive case. Substituting  $j = 1$  gives

$$exp(n, i + 1) = exp(n, i) \times n$$

where the significant relation is  $exp(n, 1) = n$  and this is precisely the information conveyed by the original recurrence relations. I have already mentioned that this particular monoid with operators is the semi-ring  $(\mathbf{N}, +, \times, 0, 1)$ .

**Alternative Notation:** In the definition of the exponential algorithm given above, one notes that the argument  $n$  remains constant throughout. This feature may be emphasised by currying the exponential homomorphism and using a special bracketing notation to encapsulate the argument  $n$  thus:

$$\begin{aligned} exp[[n]](i + j) &= exp[[n]]i \times exp[[n]]j \\ exp[[n]]0 &= 1 \end{aligned}$$

with signature

$$exp: \mathbf{N}_1 \longrightarrow ((\mathbf{N}, +, 0) \longrightarrow (\mathbf{N}_1, \times, 1))$$

## TAIL-RECURSIVE ALGORITHMS

and for  $j = 1$ , one has

$$\text{exp}[[n]](i + 1) = \text{exp}[[n]]i \times n$$

Thus we have the interpretation that  $\text{exp}[[n]]$  is a ‘variable function that depends on  $n$  for its form’, to use the words of Schönfinkel ([1924] 1967, 359). In our terminology  $\text{exp}[[n]]$  is an indexed set of homomorphisms. It is important to note that, although  $\text{exp}[[n]]$  is in curried form, it is **not** tail-recursive, a fact which is immediately obvious from its signature.

Now, whereas specifications might be presented in linear form with a particular view to an easy transition to a programming language, it is frequently the case that a subscripted or superscripted form might prove more expressive. For example, the fact that  $\text{exp}[[n]]$  is really an indexed homomorphism is better expressed as  $\text{exp}_n$ , even though the former is more suited to programming (at least until such time as we really are able to program in 2-dimensional form). Using such subscripted notation, the constructive step of the  $\text{exp}[[n]]$  homomorphism takes on the more elegant form

$$\text{exp}_n(i + 1) = \text{exp}_n(i) \times n$$

It should not be supposed that once we have found the constructive form, then we may proceed to an implementation. The experience of the Fibonacci function is a precursor of possibilities. Indeed we shall turn to consider the derivation of a tail-recursive form as befits the title of the Chapter. However, before doing so it is crucial to grasp that constructive mathematics (and therefore the derivation of efficient correct computational forms) does not proceed independently of the underlying mathematics. In the case in hand, we observe that there are properties of the  $\text{exp}$  function which are *not* captured by the homomorphism definition, to wit:

$$\text{exp}_{mn}(i) = \text{exp}_m(i) \times \text{exp}_n(i)$$

and consequently

$$\text{exp}_n(2i) = \text{exp}_{n^2}(i)$$

which leads to an implementation that is far more efficient than the tail-recursive form to be discussed.

2.1.1. Tail-Recursion via the Arzac and Kodratoff Method

The construction of tail-recursive forms from recursive functions is frequently viewed as an ‘implementation’ technique, one which ultimately leads to imperative programs. Such a transformation is considered to be a means whereby one obtains an efficient program by avoiding the use of a stack while at the same time preserving correctness. Burstall and Darlington considered the problem in the seventies and gave us the term ‘eureka’ step and the technique of ‘unfold/fold’ (1977). Their domain of concern was functional programming.

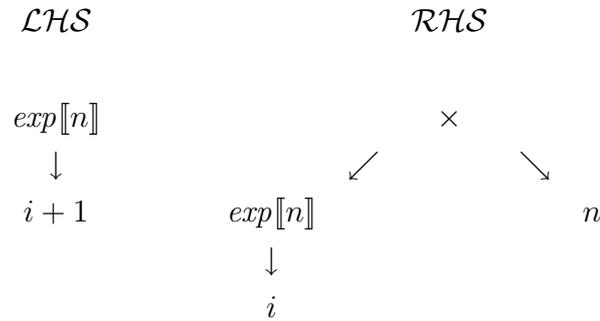
A similar concern, and one which was all the more urgent, occupied the attention of PROLOG programmers. The reason for the urgency lay in the simple fact that there are not really any imperative constructs in the language. One could always ‘fake’ such constructs by using ‘assert’ and ‘retract’. But such an approach would be totally against the spirit of PROLOG. The ‘cut’ was one of the mechanisms that provided efficient execution of programs, a mechanism that ‘interfered’ with the conceptual model of logic programming. The discovery of the difference list which led to very efficient programs was a clear breakthrough. It enhanced rather than interfered with the conceptual model. Similarly, construction of tail-recursive forms was an enhancement. Strangely, it was not until the *third* edition of Clocksin and Mellish, the classic introductory text on PROLOG, that tail-recursion as a programming technique was introduced (1987). The term used was ‘accumulator’, one that seems to be standard with respect to PROLOG (Sterling and Shapiro 1986, 125). The only reference to tail-recursion in Hogger (1984, 220) is to the work of Bruynooghe in ‘implementation technology’ in 1977.

How does a programmer construct a tail-recursive form from a recursive algorithm? From a conceptual point view, the work of Arzac and Kodratoff must be singled out (1982). I present their approach in the concrete case of the *exp* function. In the Arzac and Kodratoff method, one considers the left hand side and right hand side of the recursive definition:

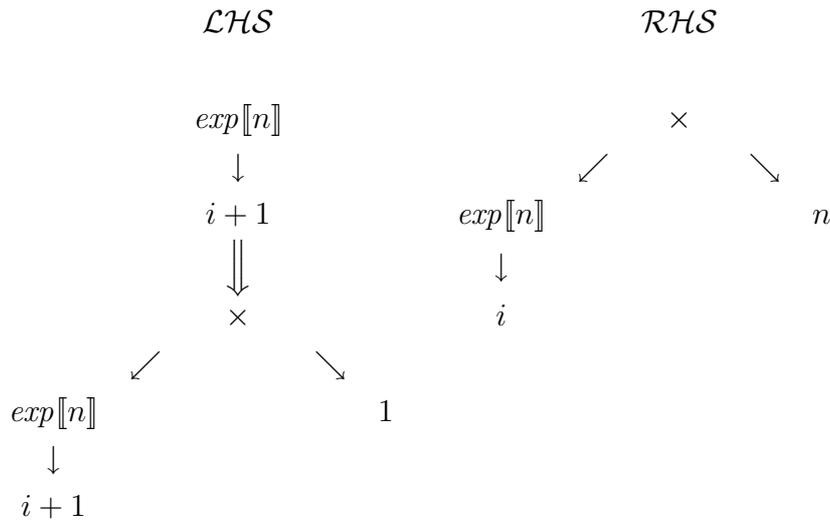
$$\text{exp}[n](i + 1) = \text{exp}[n]i \times n$$

## TAIL-RECURSIVE ALGORITHMS

and represents them by corresponding trees:



The left hand side is generalised appropriately in order to obtain a tree structure similar to that of the right hand side and at the same time to preserve the equivalent value of the original left hand side. Specifically, the root of the tree must be the multiplication operator, with the original  $exp[[n]]$  forming the left hand operand. In order to conserve equivalence, the identity under multiplication must be chosen for the right hand operator. The resulting transformation gives the tree:



Finally, the new general tree is named:

$$\overline{exp}[[n]](i, u) \triangleq exp[[n]]i \times u$$

where I have chosen to mark the tail-recursive form  $\overline{exp}[[n]]$  to distinguish it clearly from the original  $exp$  function. In practice, when programming in PROLOG, the same name may be used since the forms are distinguished by the number of their arguments, an important pragmatic consideration in the discipline of software engineering as applied to PROLOG. It is also important to note that this equation is the basis for the subsequent establishment of proofs. Now, setting  $u = 1$ , the

identity element of the monoid  $(\mathbf{N}_1, \times, 1)$ , one obtains the initialisation condition:

$$\text{exp}[n]i = \overline{\text{exp}}[n](i, 1)$$

Thus far, the Arzac and Kodratoff method has replaced the ‘eureka’ step. The simplicity of the method is primarily due to the shift in notational forms from a linear form to a 2-dimensional form. Conceptually, the function is a tree and vice-versa. It is the 2-dimensional tree that permits one to see the obvious structural development. Now using the unfold/fold method of Burstall and Darlington immediately leads to

case  $i = 0$ :

$$\begin{aligned} \overline{\text{exp}}[n](0, u) &= \text{exp}[n]0 \times u \\ &= u \end{aligned}$$

Writing, this in the form  $\overline{\text{exp}}[n][0]u = u$ , implies that  $\overline{\text{exp}}[n][0]$  is an identity function. Alternatively, the  $[n]$  and  $[0]$  may be merged to give the function  $\text{exp}[n, 0]$ , or even more appropriately  $\text{exp}_n[0]$ .

case  $i = j + 1$ :

$$\begin{aligned} \overline{\text{exp}}[n](j + 1, u) &= \text{exp}[n](j + 1) \times u \\ &= (\text{exp}[n]j \times n) \times u \\ &= \text{exp}[n]j \times (n \times u) \\ &= \overline{\text{exp}}[n](j, n \times u) \end{aligned}$$

Thus, again using the ‘extended’ curried notation one has

$$\overline{\text{exp}}[n, j + 1]u = \overline{\text{exp}}[n, j](n \times u)$$

But,  $\overline{\text{exp}}[n, 1]u = n \times u$ . Therefore,

$$\overline{\text{exp}}[n, j + 1]u = \overline{\text{exp}}[n, j] \circ \overline{\text{exp}}[n, 1]u$$

from which one deduces  $\overline{\text{exp}}[n, j + 1] = \overline{\text{exp}}[n, j] \circ \overline{\text{exp}}[n, 1]$  and, in general

$$\overline{\text{exp}}[n, i + j] = \overline{\text{exp}}[n, i] \circ \overline{\text{exp}}[n, j]$$

The function  $\overline{\text{exp}}[n, i]$  has the signature:

$$\overline{\text{exp}}[n, i]: (\mathbf{N}_1, (\mathbf{N}, +, 0)) \longrightarrow ((\mathbf{N}_1, \times, 1) \longrightarrow (\mathbf{N}_1, \times, 1))$$

## TAIL-RECURSIVE ALGORITHMS

It is straightforward to show that the set of all functions  $\overline{exp}[n, i]$ , denoted  $\overline{Exp}$ , is a commutative monoid of functions under composition  $\circ$ , with identity element  $\overline{exp}[n, 0]$ , denoted  $I_{\overline{exp}}$ . Moreover, it is epimorphic to the monoid  $(\mathbf{N}_1, \times, 1)$ .

**THEOREM 4.1.** *There is an epimorphism from  $(\overline{Exp}, \circ, I_{\overline{exp}})$  to  $(\mathbf{N}_1, \times, 1)$ .*

*Proof:* Define the function  $f: (\overline{Exp}, \circ, I_{\overline{exp}}) \longrightarrow (\mathbf{N}_1, \times, 1)$  by  $f: \overline{exp}[n, i] \mapsto n^i$ .

Then

$$f(\overline{exp}[n, 0]) = n^0 = 1$$

and

$$\begin{aligned} f(\overline{exp}[n, i] \circ \overline{exp}[n, j]) &= f(\overline{exp}[n, i + j]) \\ &= n^{i+j} \\ &= n^i \times n^j \\ &= f(\overline{exp}[n, i]) \times f(\overline{exp}[n, j]) \end{aligned}$$

Furthermore,  $f$  yields an equivalence relation on  $\overline{Exp}$ , denoted by  $E_f$ :

$$\overline{exp}[n, i] E_f \overline{exp}[m, j] \iff f(\overline{exp}[n, i]) = f(\overline{exp}[m, j])$$

The equivalence class of  $\overline{exp}[m, j]$  under  $E_f$ , is denoted  $[n]$ , where

$$[n] = \{\overline{exp}[m, j] \mid m^j = n\}$$

Then the set of equivalence classes under  $E_f$  forms a monoid called the quotient monoid which may be denoted by  $(\overline{Exp} \setminus E_f, \otimes, [1])$ , where the operation  $\otimes$  is defined by:

$$\begin{aligned} - \otimes - &: \overline{Exp} \times \overline{Exp} \longrightarrow \overline{Exp} \\ [m] \otimes [n] &\triangleq [m \times n] \end{aligned}$$

which establishes the theorem

**THEOREM 4.2.**  $(\overline{Exp} \setminus E_f, \otimes, [1]) \cong (\mathbf{N}_1, \times, 1)$ .

From a practical point of view, the significance of this result is that  $\overline{exp}[n, i]u$  is exactly the invariant of the corresponding while loop program which is used in conventional proofs of correctness. To establish such an invariant using the approach advocated by Dijkstra and Gries is non-intuitive. That it emerges naturally as a consequence of the Theory of Tail-recursive algorithms is significant. To illustrate this point I give here the while loop program with invariant  $k$ :

```

u ← 1;
while i ≠ 0 do
  -- k =  $\overline{exp}[n, i]u = n^i \cdot u$ 
  u ← n × u;
  i ← i - 1;
  -- k =  $\overline{exp}[n, i - 1](n \times u) = n^{i-1} \cdot (n \times u)$ 
end while
return u

```

## 2.2. An Alternative Version of Exp

We have considered a recursive definition of the *exp* function and shown how one may obtain a tail-recursive form equivalent to a while loop program. Such a tail-recursive form is efficient. However, I have noted that this is not the best that we can do. Recall that

$$\mathit{exp}[mn]i = \mathit{exp}[m]i \times \mathit{exp}[n]i$$

from which one may deduce that the exponential function *exp* may be defined by the recurrence relations:

$$n^0 = 1$$

$$n^{2i} = (n^2)^i$$

$$n^{2i+1} = (n^2)^i \times n$$

Denoting this form of the exponential by *exp'* to distinguish it from the version in the previous subsection one obtains:

$$\begin{aligned} \mathit{exp}'(n, i) &\triangleq \text{if } i = 0 \text{ then } 1 \\ &\quad \text{else if } \text{odd}(i) \\ &\quad \quad \text{then } \mathit{exp}'(n^2, i \text{ div } 2) \times n \\ &\quad \quad \text{else } \mathit{exp}'(n^2, i \text{ div } 2) \end{aligned}$$

This is the basis for a more efficient tail-recursive form of the exponential function. Details of its computational complexity are given in Arzac (1985, 38). To obtain the tail-recursive form I give a direct translation of the recurrence relations in curried form:

$$\mathit{exp}'[[n]]0 = 1$$

$$\mathit{exp}'[[n]]2i = \mathit{exp}'[[n^2]]i$$

$$\mathit{exp}'[[n]](2i + 1) = \mathit{exp}'[[n^2]]i \times n$$

## TAIL-RECURSIVE ALGORITHMS

where the signature is

$$\text{exp}' : \mathbf{N}_1 \longrightarrow (\mathbf{N}, +, 0) \longrightarrow (\mathbf{N}_1, \times, 1)$$

It may be shown that  $\text{exp}'\llbracket n \rrbracket$  is a homomorphism. The corresponding tail-recursive form, derived via the Arzac and Kodratoff method, is given by

$$\overline{\text{exp}'}\llbracket n, i \rrbracket u = \text{exp}'(n, i) \times u$$

Appropriate substitutions give

$$\overline{\text{exp}'}\llbracket n, 0 \rrbracket u = u$$

$$\overline{\text{exp}'}\llbracket n, 2i \rrbracket u = \overline{\text{exp}'}\llbracket n^2, i \rrbracket u$$

$$\overline{\text{exp}'}\llbracket n, 2i + 1 \rrbracket u = \overline{\text{exp}'}\llbracket n^2, i \rrbracket \circ \overline{\text{exp}'}\llbracket n, 1 \rrbracket u$$

with signature

$$\overline{\text{exp}'} : (\mathbf{N}_1, (\mathbf{N}, +, 0)) \longrightarrow ((\mathbf{N}_1, \times, 1) \longrightarrow (\mathbf{N}_1, \times, 1))$$

Finally, one has the monoid  $(\overline{\text{Exp}'}, \circ, I_{\overline{\text{exp}'}})$  which is epimorphic to  $(\mathbf{N}_1, \times, 1)$ :

$$f : (\overline{\text{Exp}'}, \circ, I_{\overline{\text{exp}'}}) \longrightarrow (\mathbf{N}_1, \times, 1) \text{ such that } f : \overline{\text{exp}'}\llbracket n, i \rrbracket \mapsto n^i$$

### 2.2.1. A PROLOG Encoding for Version 1

I have already given a while loop program for this algorithm. I now demonstrate that the same specification may readily be transcribed into PROLOG. First, the clauses for the tail-recursive version of the exponential are encoded using the equations:

$$\overline{\text{exp}}\llbracket n, 0 \rrbracket u = u$$

$$\overline{\text{exp}}\llbracket n, j \rrbracket u = \overline{\text{exp}}\llbracket n, j - 1 \rrbracket (n \times u)$$

to give

$$\text{exp}(N, 0, U, U).$$

$$\text{exp}(N, J, U, \text{Result}) :-$$

$$J1 \text{ is } J - 1,$$

$$NU \text{ is } N * U,$$

$$\text{exp}(N, J1, NU, \text{Result}).$$

Then, the connection between the 'usual' exponential function and the tail-recursive form is given via the equation:

$$\text{exp}\llbracket n \rrbracket i = \overline{\text{exp}}\llbracket n, i \rrbracket 1$$

which may be transcribed as

$\text{exp}(N, I, \text{Result}) :- \text{exp}(N, I, 1, \text{Result}).$

Computation is triggered by a query such as

$?- \text{exp}(2, 24, \text{Result}).$

giving the result 16777216.

As a final remark on the exponential function, let me add that there is a great wealth of material on the various forms of algorithms for the efficient computation thereof, material that is of both theoretical as well as of pragmatic interest (Knuth 1981, 2:441–66). To complement the theoretical material cited, I have recast some of it as algorithmic specifications in the Irish *VDM* and included it in Chapter 5.

### 2.3. Factorial Function

There is something remarkable about the fact that the factorial function is frequently to be found in introductions to computing and algorithms. What is remarkable is that it is useless *per se*! Practically speaking only a few values can be computed for  $n$  in the range  $\{1 \dots 10\}$ , say, unless special big number arithmetic algorithms are employed. Nevertheless, it is of considerable importance for several reasons which shall become clear. First, it is used by some authors to exhibit the alleged inefficiency of recursion (Ralston 1971, 346; Dromey 1982, 370). Consequently, I feel obligated to refute such allegations here. Second, it is so well-known that it is ideal to exhibit important relationships such as the correspondence between the denotational semantics of the while loop and the tail-recursive form where the ‘accumulator’ is exhibited to correspond to ‘memory’ and thus justify the remark that a tail-recursive form is indeed equivalent to a while loop. Third, I use the tail-recursive form of the factorial to justify yet again an earlier remark on loop invariants.

The factorial function may be expressed in the usual form by recurrence relations:

$$1! = 1$$

$$(n + 1)! = (n + 1) \times n!, \quad n \in \mathbf{N}_1$$

Note that the usual convention  $0! = 1$  has been omitted. Knuth justifies its use by his convention on ‘vacuous products’ (1973, 1:45). It is really unnecessary and to a certain extent spoils the theory to be developed. Arzac makes the remark that the convention is pointless (1985, 28). In addition, from the perspective of mathematical

## TAIL-RECURSIVE ALGORITHMS

analysis, the factorial function generalises to the gamma function  $\Gamma(x)$  over the real numbers excluding zero and the negative integers,  $\mathbf{R} \setminus \{0, -1, -2, \dots\}$  (Knuth 1973, 49, 479). Viewed as a complex function of one variable, the gamma function,  $\Gamma(z)$ , is a meromorphic function of  $z = x + iy$  with simple poles at  $z = 0, -1, -2, \dots$  (Lösch 1966, 4). I, therefore, find it extremely difficult to justify resorting to the definition  $0! = 1$ .

From the recurrence relations, the usual recursive algorithm may be derived:

$$fac(1) = 1$$

$$fac(n + 1) = (n + 1) \times fac(n)$$

where the signature is given by

$$fac: \mathbf{N}_1 \longrightarrow (\mathbf{N}_1, \times, 1)$$

In other words, *fac* is considered to be an injective function from the positive natural numbers into the multiplicative monoid of positive natural numbers.

### 2.3.1. Tail-Recursive Algorithm

The corresponding tail-recursive algorithm (using the Arzac and Kodratoff method) is:

$$\overline{fac} \llbracket n + 1 \rrbracket u = \overline{fac} \llbracket n \rrbracket ((n + 1) \times u)$$

$$\overline{fac} \llbracket 1 \rrbracket u = u$$

where the signature of  $\overline{fac}$  is given by

$$\overline{fac}: \mathbf{N}_1 \longrightarrow ((\mathbf{N}, +, 0) \longrightarrow (\mathbf{N}, +, 0))$$

The relationship between the tail-recursive form and the recursive algorithm is given by

$$\overline{fac} \llbracket n \rrbracket u = u \times fac(n)$$

Now, since *fac*(*n*) is just the factorial function *n*!, this leads immediately to

**THEOREM 4.1.**  $\overline{fac} \llbracket n \rrbracket: u \mapsto u \times n!$ .

The proof is obvious by induction. Moreover, the functional  $\overline{fac} \llbracket n \rrbracket$  may be regarded as the number *n*!. Then, taking multiplication as the outer law, we have the following:

THEOREM 4.2. *The functional  $\overline{fac}[n]$ ,  $n \in \mathbf{N}_1$  is an endomorphism of  $(\mathbf{N}, +, 0)$ .*

*Proof*

For the base case we obviously have

$$\begin{aligned}\overline{fac}[n]0 &= 0 \times fac(n) \\ &= 0\end{aligned}$$

and in the general case

$$\begin{aligned}\overline{fac}[n](u + v) &= (u + v) \times fac(n) \\ &= (u \times fac(n)) + (v \times fac(n)) \\ &= \overline{fac}[n]u + \overline{fac}[n]v\end{aligned}$$

Let  $\overline{Fac}$  denote the set of all such endomorphisms. Then, one has

THEOREM 4.3. *The structure  $(\overline{Fac}, (\mathbf{N}, +, 0))$  is a monoid with operators.*

**Remarks:**

- $\overline{fac}[1]$  is an identity function:

$$\overline{fac}[1]u = u \iff \overline{fac}[1] = I_{\overline{fac}}$$

- The 1–1 mapping  $\overline{fac}[n] \mapsto n$  shows that the set  $\overline{Fac}$  is denumerable.
- The law of composition in  $\overline{Fac}$  is multiplication, the outer law of the monoid with operators  $(\overline{Fac}, (\mathbf{N}, +, 0))$ .

Considering composition of  $\overline{fac}[3]$  and  $\overline{fac}[4]$ , say, one has an endomorphism  $\overline{fac}[4] \circ \overline{fac}[3]$  which is not a member of  $\overline{Fac}$ . Indeed, using the correspondence  $\overline{fac}[3] \mapsto 3!$ ,  $\overline{fac}[4] \mapsto 4!$ , it is obvious that  $\overline{fac}[4] \circ \overline{fac}[3] \mapsto n$  where  $5! < n < 6!$ . Thus,  $\overline{Fac}$  is not closed under  $\circ$ , in the sense that a pair of factorials under multiplication is not a factorial. However, we may construct the closure of  $\overline{Fac}$  under  $\circ$ , denoted  $\overline{Fac}^C$ , by agreeing that if  $m, n \in \overline{Fac}^C$ , then  $m \times n \in \overline{Fac}^C$ . The structure  $(\overline{Fac}^C, (\mathbf{N}, +, 0))$  is a monoid with operators, of which  $(\overline{Fac}, (\mathbf{N}, +, 0))$  is a substructure. In addition,  $(\overline{Fac}^C, \circ, I_{\overline{fac}})$  is isomorphic to a submonoid of  $(\mathbf{N}_1, \times, 1)$ .

## TAIL-RECURSIVE ALGORITHMS

### 2.4. Tail-recursion and While loops

The passage from a tail-recursive form to a while loop program is immediate. To demonstrate the equivalence of the two representations I choose to present the ‘usual’ denotational semantics of a while loop and to recast the tail-recursive form of the factorial as a semantic function over abstract syntax. First, using the standard notions of environment,  $ENV$ , and store,  $STORE$ , details concerning which are to be found in Appendix B, I will define a state as

$$\varsigma \in STATE :: ENV \times STORE$$

Then, defining the while loop as the syntactic domain

$$While :: Exp \times Stat$$

the semantics of the while loop may be presented abstractly by

$$\begin{aligned} \mathcal{M}: While &\longrightarrow STATE \longrightarrow STATE \\ \mathcal{M}[[mk-While(e, s)]]_{\varsigma} &\triangleq \\ \neg \mathcal{M}_{bool}[[e]]_{\varsigma} & \\ \rightarrow \varsigma & \\ \rightarrow \mathcal{M}[[mk-While(e, s)]] \circ \mathcal{M}_{stat}[[s]]_{\varsigma} & \end{aligned}$$

where  $\mathcal{M}_{bool}$  and  $\mathcal{M}_{stat}$  denote the semantic functions for boolean expressions and statements, respectively. It seems to be superfluous to remark that the semantic function is in tail-recursive form! We may exploit this duality between the denotational semantics style and tail-recursive forms to express the latter as a semantic function over abstract syntax. Specifically, I define the syntactic domain

$$Fac :: \mathbf{N}_1$$

Then, the tail-recursive factorial function may be written as

$$\begin{aligned} \mathcal{M}: Fac &\longrightarrow \mathbf{N}_1 \longrightarrow \mathbf{N}_1 \\ \mathcal{M}[[mk-Fac(n)]]_u &\triangleq \\ \mathcal{M}_{\neq 1}[[n]]_u & \\ \rightarrow u & \\ \rightarrow \mathcal{M}[[mk-Fac(n-1)]] \circ \mathcal{M}_{\times}[[n]]_u & \end{aligned}$$

where I have chosen to employ semantic functions  $\mathcal{M}_{\neq 1}$  and  $\mathcal{M}_{\times}$  to denote the test for inequality with respect to 1 and multiplication, respectively. Comparing

the semantics of the while loop and the ‘semantics’ of the factorial function, we observe that the variable  $u$ , the accumulator, corresponds to the state. Moreover, the transcription of the tail-recursive form of the factorial function into a while loop is dictated by the denotational semantics of the former:

```

 $u \leftarrow 1;$ 
while  $n \neq 1$  do
  --  $k = \overline{Fac}[[n]]u = n! \cdot u$ 
   $u \leftarrow n \times u;$ 
   $n \leftarrow n - 1;$ 
  --  $k = \overline{Fac}[[n - 1]](n \times u) = (n - 1)! \cdot (n \times u)$ 
end while
return  $u$ 

```

Note that I have also included the invariant of the while loop,  $k = \overline{Fac}[[n]]u = n! \cdot u$ , which is simply the tail-recursive function itself.

## 2.5. Combinatorial Functions

To conclude this brief overview of the factorial function, I include some remarks on its occurrence in the definition of the binomial coefficient:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

One does not compute  $\binom{n}{k}$  in this form. For computational purposes, recurrence relations such as

$$\begin{aligned} \binom{n}{0} &= 1 \\ \binom{n}{n} &= 1 \\ \binom{n}{k} &= \frac{n-k+1}{k} \binom{n}{k-1}, \quad k \in \{1 \dots n-1\} \end{aligned}$$

may be used. In addition, since  $k$  divides  $(n-k+1)\binom{n}{k-1}$ , integer arithmetic may be used in computations. Finally, it is to be noted that  $\binom{n}{k} = \binom{n}{n-k}$  and in computing such a binomial coefficient,  $\binom{n}{j}$  is chosen, where  $j = \min(k, n-k)$ .

Denoting  $\binom{n}{k}$  by  $C(n, k)$ , the tail-recursive algorithm given by the Arzac and

## TAIL-RECURSIVE ALGORITHMS

Kodratoff method is

$$\begin{aligned}\overline{C}[[n, 0]]u &\triangleq u \\ \overline{C}[[n, k]]u &\triangleq \overline{C}[[n, k-1]]\frac{n-k+1}{k} \times u\end{aligned}$$

where the relationship between  $C(n, k)$  and  $\overline{C}[[n, k]]$  is

$$C(n, k) = \overline{C}[[n, k]]1$$

The similarity between this form and that of the first tail-recursive form for the exponential function is striking. Finally, from a conceptual point of view, it ought to be noted that there are algorithms in which binomial coefficients are computed implicitly. The *de Casteljau* algorithm for the computation of a point on a Bézier curve, which will be discussed in Chapter 7, is a classic example.

### 2.6. Multiplication

Multiplication of natural numbers may be formally defined in terms of addition.

Denoting multiplication by the function *mul*, one has the obvious results:

$$\begin{aligned}mul: \mathbf{N} \times \mathbf{N}_1 &\longrightarrow \mathbf{N} \\ mul(m, 1) &= m \\ mul(m, n) &= mul(m, n-1) + m\end{aligned}$$

This is just a rewrite of the classic definition of multiplication as a primitive recursive function (see (Hermes 1969, 64) for example). Gödel took this for granted in his famous 1931 paper (Gödel [1931] 1967, 603), probably on account of the fact that Dedekind had already used it in 1888.

Note carefully that I have chosen to use 1 in the base case instead of 0. I have done this in order to stay within the notion of the natural numbers as a semiring. Note that the second argument to *mul* acts like a counter. Now, it should be fairly obvious, either by looking at the signature or by considering the defining equations given above, that there is a ‘natural’ curried form for *mul*:

$$\begin{aligned}mul: \mathbf{N}_1 &\longrightarrow \mathbf{N} \longrightarrow \mathbf{N} \\ mul[[1]]m &= m \\ mul[[n]]m &= mul[[n-1]](n+m)\end{aligned}$$

which is structurally similar to the tail-recursive form for the factorial function, the difference being that the latter has a multiplication operator in place of the addition operator above. Moreover, it is clear that

$$\begin{aligned} \text{mul}\llbracket n \rrbracket(p + q) &= \text{mul}\llbracket n \rrbracket p + \text{mul}\llbracket n \rrbracket q \\ \text{mul}\llbracket n \rrbracket 0 &= 0 \end{aligned}$$

Clearly,  $\text{mul}\llbracket n \rrbracket$  is a homomorphism:

$$\text{mul}\llbracket n \rrbracket: (\mathbf{N}, +, 0) \longrightarrow (\mathbf{N}, +, 0)$$

or, if one prefers, a homomorphism from the monoid with operators  $(\mathbf{N}_1, (\mathbf{N}, +, 0))$  to the monoid  $(\mathbf{N}, +, 0)$ . Note that I have just demonstrated that the curried form is the computational equivalent of a positive natural number as an endomorphism of  $(\mathbf{N}, +, 0)$  where the outer law is multiplication.

Starting with the original recursive definition and using the Arzac and Kodratoff method, a tail-recursive version is readily obtained:

$$\overline{\text{mul}}\llbracket n, m \rrbracket u \triangleq \text{mul}(n, m) + u$$

where  $\overline{\text{mul}}$  is given by

$$\begin{aligned} \overline{\text{mul}}\llbracket 1, m \rrbracket u &= m + u \\ \overline{\text{mul}}\llbracket n, m \rrbracket u &= \overline{\text{mul}}\llbracket n - 1, m \rrbracket (m + u) \end{aligned}$$

Here I have chosen to switch the order of the arguments. Moreover, using the base equation  $\text{mul}(m, 0) = 0$  gives

$$\overline{\text{mul}}\llbracket 0, m \rrbracket u = u$$

and, therefore, the signature may be chosen to be

$$\overline{\text{mul}}: \mathbf{N} \times \mathbf{N} \longrightarrow ((\mathbf{N}, +, 0) \longrightarrow (\mathbf{N}, +, 0))$$

It is easy to establish the following two properties:

$$\begin{aligned} \overline{\text{mul}}\llbracket 0, m \rrbracket u = u &\iff \overline{\text{mul}}\llbracket 0, m \rrbracket = I_{\overline{\text{mul}}}. \\ \overline{\text{mul}}\llbracket n, p + q \rrbracket &= \overline{\text{mul}}\llbracket n, p \rrbracket \circ \overline{\text{mul}}\llbracket n, q \rrbracket. \end{aligned}$$

Consequently, there is an epimorphism  $f: (\overline{\text{Mul}}, \circ, I_{\overline{\text{mul}}}) \longrightarrow (\mathbf{N}, +, 0)$  defined by

$$f: \overline{\text{mul}}\llbracket n, m \rrbracket \mapsto n \times m$$

which gives

$$\begin{aligned} f(\overline{\text{mul}}\llbracket n, p + q \rrbracket) &= f(\overline{\text{mul}}\llbracket n, p \rrbracket) + f(\overline{\text{mul}}\llbracket n, q \rrbracket) \\ f(\overline{\text{mul}}\llbracket 0, n \rrbracket) &= 0 \end{aligned}$$

Moreover,  $f$  induces an equivalence relation  $E_f$  on the set  $\overline{\text{Mul}}$  of all tail-recursive functions:

$$\overline{\text{mul}}\llbracket m, n \rrbracket E_f \overline{\text{mul}}\llbracket m \times n, 1 \rrbracket \iff f(\overline{\text{mul}}\llbracket m, n \rrbracket) = f(\overline{\text{mul}}\llbracket m \times n, 1 \rrbracket)$$

## TAIL-RECURSIVE ALGORITHMS

Denote by  $[mn]$  the equivalence class

$$[mn] = \{\overline{mul}[[r, s] \mid \overline{mul}[[r, s] E_f \overline{mul}[[m \times n, 1]]]\}$$

This leads immediately to

THEOREM 4.1.  $(\overline{Mul} \setminus E_f, \circ, [0]) \cong (\mathbf{N}, +, 0)$ .

From a primitive recursive function expressed in one representation, we moved to a curried form which itself was tail-recursive, and thence to another tail-recursive form given by the Arzac and Kodratoff approach. Although the function was very elementary—multiplication, the resulting algebraic results were striking. I have refrained from giving equivalent while loop programs or making remarks on invariants in order to highlight the fact that in the theory of tail-recursive functions one may do some very interesting algebra. We are now prepared to focus on the tail-recursive forms that occur in the Irish School of the *VDM*.

### 3. The Free Monoid Homomorphisms

Having considered the application of the Arzac and Kodratoff method to some basic algorithms on the natural numbers to derive corresponding tail-recursive forms, attention will now be focused on the application of that same method to sequences. One reason for doing so, is to establish a basis for the efficient implementation of the ‘functional subset’ of the *VDM Meta-IV* in PROLOG, particularly in view of the fact that the latter does not really support assignment (even though assert and retract clauses may be used for that purpose). Such a functional subset is completely adequate for the purposes of executable specifications using *Meta-IV*. A note on the phrase ‘functional subset’ is needed. In the traditional *VDM Meta-IV* an imperative notation was also provided. Instead of adding to the power of the *Meta-IV* notation, I firmly believe that it does just the opposite. There is **no place** for such imperative notation in the Irish School of the *VDM*.

One approach is to consider the basic operations on sequences on a case by case basis, and indeed, from a pedagogical point of view this is the preferred approach.

## The Free Monoid Homomorphisms

However, as indicated earlier, the theory of sequences is remarkably well developed. Therefore, as a starting point, let us consider a particular class of operations—the  $\Sigma^*$ -homomorphisms, such as *len* and *elems*. The general approach which I present, will demonstrate the possibility of using a language such as PROLOG to write ‘generic’ algorithms and thereby give some insight into the nature of genericity itself, a feature of programming languages such as Ada.

The basic theoretical result is that all monoid homomorphisms of the free monoid  $(\Sigma^*, \wedge, \Lambda)$  onto  $(M, \oplus, u)$  may be represented by the  $\varphi$  homomorphism:

$$\varphi: (\Sigma^*, \wedge, \Lambda) \longrightarrow (M, \oplus, u)$$

where

$$\varphi(\sigma \wedge \tau) = \varphi(\sigma) \oplus \varphi(\tau)$$

$$\varphi(\langle e \rangle) = F(e)$$

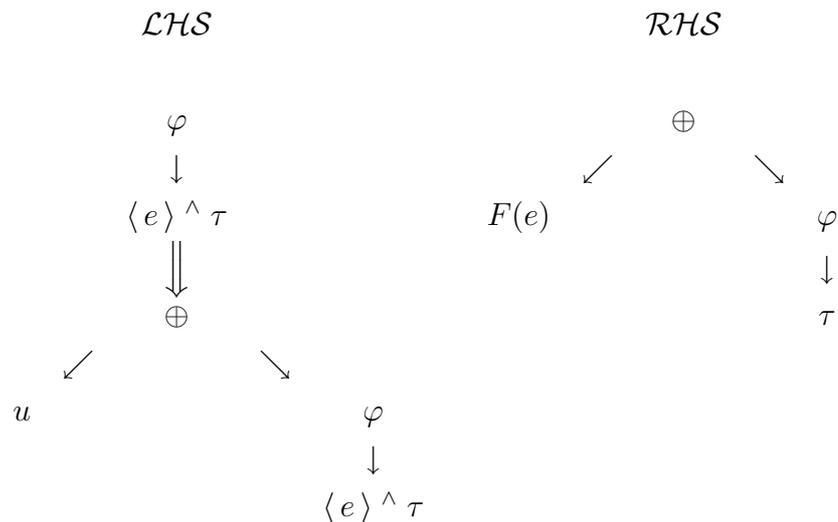
$$\varphi(\Lambda) = u$$

By choosing  $\sigma = \langle e \rangle$ , one immediately obtains the constructive form:

$$\begin{aligned} \varphi(\langle e \rangle \wedge \tau) &= \varphi(\langle e \rangle) \oplus \varphi(\tau) \\ &= F(e) \oplus \varphi(\tau) \end{aligned}$$

from which it is possible to derive an equivalent general tail-recursive algorithm.

Using the Arzac and Kodratoff method



one obtains the tail-recursive form:

$$\Phi(\sigma, x) = x \oplus \varphi(\sigma)$$

Note the order of the operands in the expression  $x \oplus \varphi(\sigma)$ . One would perhaps prefer to write  $\varphi(\sigma) \oplus x$  in analogy to the forms that I presented earlier in the Chapter.

## TAIL-RECURSIVE ALGORITHMS

However, in the general case that  $(M, \oplus, u)$  is non-commutative, one is forced to adhere to the order as given. Now, to express the definition of the  $\varphi$  homomorphism in terms of the tail-recursive form  $\Phi$  one substitutes the identity element  $u$  for  $x$  giving:

$$\begin{aligned}\Phi(\sigma, u) &= u \oplus \varphi(\sigma) \\ &= \varphi(\sigma)\end{aligned}$$

i.e.,

$$\varphi(\sigma) = \Phi(\sigma, u)$$

Finally, starting with the definition:

$$\Phi(\sigma, x) = x \oplus \varphi(\sigma)$$

one seeks to eliminate the  $\varphi$  homomorphism in order to obtain a definition solely in terms of  $\Phi$ . There are two cases. Substituting  $\Lambda$  for  $\sigma$  gives

$$\begin{aligned}\Phi(\Lambda, x) &= x \oplus \varphi(\Lambda) \\ &= x \oplus u \\ &= x\end{aligned}$$

Now consider the general case:

$$\begin{aligned}\Phi(\sigma \wedge \tau, x) &= x \oplus \varphi(\sigma \wedge \tau) && \text{-- unfold} \\ &= x \oplus (\varphi(\sigma) \oplus \varphi(\tau)) && \text{-- } \varphi \text{ is a homomorphism} \\ &= (x \oplus \varphi(\sigma)) \oplus \varphi(\tau) && \text{-- } \oplus \text{ is associative} \\ &= \Phi(\tau, x \oplus \varphi(\sigma)) && \text{-- fold}\end{aligned}$$

Then, the tail-recursive form of the  $\varphi$  homomorphism is

$$\begin{aligned}\Phi(\langle e \rangle \wedge \tau, x) &= \Phi(\tau, x \oplus \varphi(\langle e \rangle)) \\ &= \Phi(\tau, x \oplus F(e))\end{aligned}$$

Note that the entire development above signals the centrality of the (not necessarily commutative) monoid in computing, as indicated by (i) the importance of the use of the identity element  $u$ , and (ii) the mention of the associative law in the comment above. Later in the Chapter, I demonstrate that the transformation may be applied to the weaker structure of the semigroup.

### 3.1. Properties of the Tail-recursive form

Consider the general tail-recursive form of free monoid homomorphisms  $\varphi(\sigma) = \Phi(\sigma, u)$ , where:

$$\Phi(\langle e \rangle \wedge \tau, x) = \Phi(\tau, x \oplus F(e))$$

$$\Phi(\langle e \rangle, x) = x \oplus F(e)$$

$$\Phi(\Lambda, x) = x$$

Using, the currying technique, the  $\Phi$  algorithm may be written as:

$$\Phi[\Lambda]x = x$$

$$\Phi[\langle e \rangle]x = x \oplus F(e)$$

$$\begin{aligned} \Phi[\langle e \rangle \wedge \tau]x &= \Phi[\tau](x \oplus F(e)) \\ &= \Phi[\tau] \circ \Phi[\langle e \rangle]x \end{aligned}$$

with the signature of  $\Phi$  written in the form

$$\Phi: (\Sigma^*, \wedge, \Lambda) \longrightarrow ((M, \oplus, u) \longrightarrow (M, \oplus, u))$$

Clearly,  $\Phi[\Lambda]$  is the identity function of  $M \longrightarrow M$  and from the expression

$$\Phi[\langle e \rangle \wedge \tau]x = \Phi[\tau] \circ \Phi[\langle e \rangle]x$$

one may deduce the general law of composition

$$\Phi[\sigma \wedge \tau] = \Phi[\tau] \circ \Phi[\sigma]$$

Then, the set of all tail-recursive homomorphisms of the free monoid  $(\Sigma^*, \wedge, \Lambda)$ , denoted  $\Phi$  forms a monoid under composition which is isomorphic to the free monoid  $(\Sigma^*, \wedge, \Lambda)$ :

**THEOREM 4.1.**  $(\Phi, \circ, \Phi[\Lambda]) \cong (\Sigma^*, \wedge, \Lambda)$ .

## TAIL-RECURSIVE ALGORITHMS

### 3.2. PROLOG Implementation

One might choose to implement each of the  $\Sigma^*$ -homomorphisms, *len*, *elems*, etc., separately. In fact, from a pedagogical point of view, that is the recommended starting point. However, with a little reflection and effort, a generic implementation may be constructed. First, it is clear from the expression,  $\varphi(\sigma) = \Phi[\sigma]u$ , that the identity element  $u$  of  $(M, \oplus, u)$  must be supplied as a formal parameter. Similarly, from  $\Phi[\langle e \rangle \wedge \tau]x = \Phi[\tau](x \oplus F(e))$ , we conclude that both  $F$  and  $\oplus$  must be formal parameters. Consequently, the top-level PROLOG clause of  $\Sigma^*$ -homomorphisms is given by

$$\begin{aligned} \text{phi}(F, Op, Id, Seq, Result) :- \\ \quad \text{Constant} =.. [Id, Val], \\ \quad \text{call}(\text{Constant}), \\ \quad \text{phi}(F, Op, Seq, Val, Result). \end{aligned}$$

where  $Op = \oplus$  and  $Id = u$ . Since I intend to supply the identity element as a constant function by name, then I must evaluate it: ' $\text{call}(\text{Constant})$ '. The actual transcription of the tail-recursive form is then straightforward:

$$\begin{aligned} \text{phi}(F, Op, [], U, U). \\ \text{phi}(F, Op, [H | T], U, V) :- \\ \quad \text{Function} =.. [F, H, E], \\ \quad \text{call}(\text{Function}), \\ \quad \text{Operation} =.. [Op, U, E, U1], \\ \quad \text{call}(\text{Operation}), \\ \quad \text{phi}(F, Op, T, U1, V). \end{aligned}$$

To demonstrate the use of this generic encoding I present some simple instantiations:

EXAMPLE 4.1. To compute the length of a sequence, one may use

$$\text{len}(\text{Seq}, N) :- \text{phi}(\text{one}, \text{sum}, \text{zero}, \text{Seq}, N).$$

where

$$\begin{aligned} \text{zero}(0). \\ \text{one}(X, 1). \\ \text{sum}(X, Y, Z) :- Z \text{ is } X + Y. \end{aligned}$$

EXAMPLE 4.2. To compute the set of elements in a sequence one may use

$$\mathit{elems}(\mathit{Seq}, \mathit{Set}) :- \mathit{phi}(\mathit{singleton}, \mathit{union}, \mathit{nullset}, \mathit{Seq}, \mathit{Set}).$$

where

$$\begin{aligned} \mathit{nullset}([]). \\ \mathit{singleton}(X, [X]). \\ \mathit{union}([], S, S). \\ \mathit{union}(S, [], S). \\ \mathit{union}([H | T], [H], [H | T]). \\ \mathit{union}([H | T], [E], [H | T1]) :- \mathit{union}(T, [E], T1). \end{aligned}$$

EXAMPLE 4.3. To compute the hash value of a word one may use

$$\mathit{hash}(\mathit{Seq}, N) :- \mathit{phi}(\mathit{inject}, \mathit{modplus}, \mathit{zero}, \mathit{Seq}, N).$$

where

$$\begin{aligned} \mathit{inject}(X, Y) :- \mathbf{name}(X, [Y]). \\ \mathit{modplus}(X, Y, Z) :- \mathit{modplus}(101, X, Y, Z). \\ \mathit{modplus}(P, X, Y, Z) :- Z \mathbf{is} (X + Y) \mathbf{mod} P. \end{aligned}$$

One familiar with the current techniques in the encoding of abstract data types will recognise the significance of the PROLOG encodings given above. More importantly, from a conceptual point of view, the very general abstraction of the generic code presented signals the kind of difficulty expected in developing generic modules in programming languages—the more generic the code the less obvious is the correspondence between the names used and those of the actual problem domain in which the instantiations are to be deployed. In the case at hand, the name *phi*, for example, abstracts from the names *len*, *elems*, etc. Even in the case of instantiations, one may have to resort to abstract names—the *inject* operation in the hash function example. These conceptual difficulties only arise at the encoding stage—due to the limitations of the ASCII character set. The corresponding VDM specification is absolutely unambiguous and clear.

I do not feel obliged to present *all* the details of the encoding of the functional subset of the *Meta-IV*. Such encoding may be carried out by any competent undergraduate student of computer science who is familiar with the notion of tail-recursion and who has access to the definitions given in Chapter 2. Indeed, an encoding in

## TAIL-RECURSIVE ALGORITHMS

imperative programming languages is also a simple matter, given the relationship between tail-recursive forms and while loop programs presented in this Chapter. There are, however, two further aspects which I would like to address: the notion of reduction,  $\oplus/\$ , and function application,  $f^*$ .

### 4. Reduction

The *VDM Meta-IV* operators of distributed union of sets,  $\cup/\$ , distributed concatenation of sequences,  $\wedge/\$ , and distributed extension of maps,  $\cup/\$ , were defined as reductions in Chapter 2. In this section I examine reduction in considerable detail. From a computational perspective, reduction has a natural tail-recursive form. But one may obtain other efficient tail-recursive forms by applying the Arzac and Kordatoff method. Whereas here I am mostly concerned with *executable* specifications, I do hasten to point out again that reduction is one of those indispensable operators of the general operator calculus for the Irish *Meta-IV*.

The reduction operator has had a long history in computing. It is probably most frequently associated with APL. In his book on the subject, Iverson defined reduction to be an operation which “is applied to all components of a vector to produce a result of a simpler structure” (Iverson 1962, 16). Thus, if  $\mathbf{x} = (x_1, x_2, \dots, x_j, \dots, x_n)$  is a vector of (real) numbers, then reduction with respect to addition  $+$  gives the sum of the elements:

$$+/\mathbf{v} = x_1 + x_2 + \dots + x_j + \dots + x_n$$

Backus calls it ‘Insert’ in his definition of the functional programming language FP (Backus 1978). In Backus’ notation the above vector sum would be written

$$(/+): \langle x_1, x_2, \dots, x_j, \dots, x_n \rangle = x_1 + x_2 + \dots + x_j + \dots + x_n$$

Lest one suppose that I am indeed just programming, I have deliberately chosen to use  $+/\$  in *Meta-IV* specifications in place of either  $+/\$  or  $/+$ . Thus, although I speak of executable specifications, I do not necessarily mean that the notation *per se* is executable. Rather I mean that I am dealing with constructive computational

mathematics. One has always the option of using the FP or APL forms for reduction. But reduction can also be efficiently executed in PROLOG or Ada for example.

I believe that it is inconceivable that there should exist a useful functional programming language in which reduction was not either explicitly or implicitly provided. Such is its fundamental importance. With respect to the sequence type of the *VDM Meta-IV*, and by that I mean in any Theory of Sequences whatsoever, there is a simple fundamental theorem.

**THEOREM 4.1. (The Reduction Theorem)** *Every monoid  $(M, \oplus, u)$  may be extended to the free monoid  $(M^*, \wedge, \Lambda)$  such that the monoid homomorphism*

$$\oplus / : (M^*, \wedge, \Lambda) \longrightarrow (M, \oplus, u)$$

*extends the binary operation  $\oplus$  to sequences of elements from  $M$ .*

In conceptual model terminology, reduction is to be interpreted as a mechanism for extending a binary operation on a pair of elements to a non-empty sequence of elements of arbitrary length. Formally, the  $\oplus /$  homomorphism is defined by:

$$\begin{aligned} \oplus / (\sigma \wedge \tau) &= \oplus / \sigma \oplus \oplus / \tau \\ \oplus / \langle e \rangle &= e \\ \oplus / \Lambda &= u \end{aligned}$$

A remark is necessary on the form  $\oplus / \Lambda = u$ . It is included simply for completion. In this respect it is similar to the conventions that  $0^0 = 1$  for the primitive recursive function *exp*, and  $fac(0) = 1$  for the factorial function. In practice, from an algorithmic point of view, this line of the definition need never be used. Elementary sequence operators may be *defined* in terms of reduction.

**EXAMPLE 4.1.** The *len* homomorphism of a sequence may be defined as a reduction  $len = + / F^*$ , where  $F: \sigma \mapsto 1$ .

**EXAMPLE 4.2.** The *elems* homomorphism may be defined by  $elems = \cup / F^*$ , where  $F: \sigma \mapsto \{\sigma\}$ .

**EXAMPLE 4.3.** The *items* homomorphism may be defined by  $items = \oplus / F^*$ , where  $F: \sigma \mapsto [\sigma \mapsto 1]$ .

## TAIL-RECURSIVE ALGORITHMS

The  $\oplus/$  homomorphism has a natural tail-recursive form. Setting  $\sigma = \langle a, b \rangle$ , one obtains:

$$\begin{aligned}
 \oplus/(\langle a, b \rangle \wedge \tau) &= \oplus/\langle a, b \rangle \oplus \oplus/\tau \\
 &= (\oplus/\langle a \rangle \oplus \oplus/\langle b \rangle) \oplus \oplus/\tau \\
 &= (a \oplus b) \oplus \oplus/\tau \\
 &= \oplus/\langle a, b \rangle \oplus \oplus/\tau \\
 &= \oplus/(\langle a \oplus b \rangle \wedge \tau)
 \end{aligned}$$

I need to say exactly what I mean here by *natural* tail-recursive form. If we consider the signature of  $\oplus/$ :

$$\oplus/ : (\Sigma^*, \wedge, \Lambda) \longrightarrow (\Sigma^*, \wedge, \Lambda)$$

then it would seem that it does not conform with the usual signature that one expects to find with a tail-recursive form. But this is illusory. I may choose, if I wish, to include a null argument in my definition to obtain exactly the standard form I require. One should not suppose that the concatenation operator  $\wedge$  calls for explicit concatenation. Again I need to repeat that this is a specification and, in the case of PROLOG, pattern matching does lead to a tail-recursion as specified.

However, using the Arzac and Kodratoff approach, there are other (conventional) tail-recursive forms of reduction that can be developed, thus demonstrating that recursive algorithms may have multiple tail-recursive forms. Consequently, to the existing conceptual model of reduction as binary operator ‘extendor’ may be added the concept of efficient recursive execution. Many useful functions may be derived from the reduction operator (cf. (Iverson 1979, 1980)). Some simple examples follow.

EXAMPLE 4.4. The homomorphism  $+/ : (\mathbf{N}^*, \wedge, \Lambda) \longrightarrow (\mathbf{N}, +, 0)$  computes the sum of a sequence of natural numbers.

EXAMPLE 4.5. The homomorphism  $\times/ : (\mathbf{N}_1^*, \wedge, \Lambda) \longrightarrow (\mathbf{N}_1, \times, 1)$  computes the product of a sequence of positive natural numbers.

Naturally, there are equivalent reduction homomorphisms for the other number sets  $\mathbf{Z}, \mathbf{Q}$ , and  $\mathbf{R}$ . From an earlier remark that  $\oplus/\Lambda = u$  is redundant, then the notion of reduction may be widened to semigroups:

**THEOREM 4.2. (The Reduction Theorem for Semigroups)** *Every semigroup  $(M, \oplus)$  may be extended to the semigroup  $(M^+, \oplus)$  such that the semigroup homomorphism*

$$\oplus / : (M^+, \wedge) \longrightarrow (M, \oplus)$$

*extends the binary operation  $\oplus$  to non-empty sequences of elements from  $M$ .*

**EXAMPLE 4.6.** Let  $\downarrow$  and  $\uparrow$  denote the binary operators min and max, respectively. Then the reductions  $\downarrow / : (\mathbf{N}^+, \wedge) \longrightarrow (\mathbf{N}, \downarrow)$  and  $\uparrow / : (\mathbf{N}^+, \wedge) \longrightarrow (\mathbf{N}, \uparrow)$  compute the min and max of sequences of natural numbers, respectively.

**EXAMPLE 4.7.** If we denote the representation of a matrix  $M$  in *Meta-IV* as a sequence of sequences then  $\downarrow / \uparrow / M$  gives the minmax function.

#### 4.1. Other Tail-recursive Versions of Reduction

Should one wish to provide for reduction in an imperative programming language such as Ada then it is essential to develop a conventional tail-recursive form of reduction for efficiency purposes. First, let us consider the expression

$$\begin{aligned} \oplus / (\langle e \rangle \wedge \tau) &= \oplus / \langle e \rangle \oplus \oplus / \tau \\ &= e \oplus \oplus / \tau \end{aligned}$$

Using the Arzac and Kodratoff approach, one has the general form

$$\overline{\text{reduce}}[\sigma]x = x \oplus \oplus / \sigma$$

In the case of monoids, one has

$$\begin{aligned} \overline{\text{reduce}}[\Lambda]x &= x \oplus \oplus / \Lambda \\ &= x \end{aligned}$$

and for semigroups,

$$\begin{aligned} \overline{\text{reduce}}[\langle e \rangle]x &= x \oplus \oplus / \langle e \rangle \\ &= x \oplus e \end{aligned}$$

Now, substituting  $\sigma = \langle e \rangle \wedge \tau$ , one obtains the recursive case:

$$\begin{aligned} \overline{\text{reduce}}[\langle e \rangle \wedge \tau]x &= x \oplus \oplus / (\langle e \rangle \wedge \tau) \\ &= x \oplus (\oplus / \langle e \rangle \wedge \oplus / \tau) \\ &= x \oplus (e \oplus \oplus / \tau) \\ &= (x \oplus e) \oplus \oplus / \tau \\ &= \overline{\text{reduce}}[\tau](x \oplus e) \end{aligned}$$

## TAIL-RECURSIVE ALGORITHMS

It may readily be shown that  $\overline{\text{reduce}}[\Lambda]$  is the identity function and

$$\begin{aligned}\overline{\text{reduce}}[\sigma \wedge \tau] &= \overline{\text{reduce}}[\tau] \circ \overline{\text{reduce}}[\sigma] \\ \overline{\text{reduce}}[\langle e \rangle \wedge \tau] &= \overline{\text{reduce}}[\tau] \circ \overline{\text{reduce}}[\langle e \rangle]\end{aligned}$$

However, if one considers the expression:

$$\oplus / (\langle a, b \rangle \wedge \tau) = (a \oplus b) \oplus \oplus / \tau$$

one may use the Arzac and Kodratoff method to obtain yet another tail-recursive form:

$$\begin{aligned}\overline{\text{reduce}}[\langle a, b \rangle \wedge \tau]x &= \overline{\text{reduce}}[\tau](x \oplus (a \oplus b)) \\ \overline{\text{reduce}}[\langle a, b \rangle]x &= x \oplus (a \oplus b) \\ \overline{\text{reduce}}[\langle e \rangle]x &= x \oplus e\end{aligned}$$

with the initial condition  $x = u$  for monoids. This is even more efficient than the previous form developed and is analogous to the development of the version of the *exp* function that is based on even and odd powers of  $n$ . Essentially, it considers pairs of elements of a sequence at a time. Incidentally, such efficiency may equally be applied to any function over sequences.

### 4.2. Prolog Implementation

The natural tail-recursive version of reduce may be encoded in PROLOG as follows:

```
reduce(Op, [E], E).
reduce(Op, [A, B | Tail], Result) :-
    Function =.. [Op, A, B, Val],
    call(Function),
    reduce(Op, [Val | Tail], Result).
```

To illustrate the application of the reduce predicate consider the computation of the sum of a sequence of natural numbers:

```
?- reduce(sum, [1, 2, 3, 4], Result).
```

where the relevant clause of sum is

```
sum(X, Y, Z) :- nonvar(X), nonvar(Y), Z is X + Y.
```

Note, incidentally, the use of the **nonvar** predicate to ensure that  $X$  and  $Y$  are ground terms, i.e., are not variables when this clause is invoked. Then, of course, the binary operator may be extended at the implementation level, thus

$sum(Seq, Val) :- reduce(sum, Seq, Val).$

Let me make a remark on this form from the perspective of a conceptual model. Considering a sequence of elements as a data stream to be processed, then the execution of the natural tail-recursive algorithm may be viewed as a stack-based execution mechanism where the first element is interpreted as the top of the stack and evaluation consists in combining the next element with the top of the stack and replacing the top of the stack with the result. An imperative encoding to be given below faithfully captures this notion.

The PROLOG encoding of the first Arzac and Kodratoff tail-recursive form of reduce is very similar to that of the ‘natural’ tail-recursive form:

$reduce(Op, [E], X, XE) :-$   
      $Function = .. [Op, X, E, XE],$   
     **call**( $Function$ ).  
 $reduce(Op, [E | Tail], X, Result) :-$   
      $Function = .. [Op, X, E, XE],$   
     **call**( $Function$ ),  
      $reduce(Op, Tail, XE, Result).$

The modification to the definition of sum in terms of these new reduce clauses is straight-forward. Note that the only real difference between these two tail-recursive forms, is the occurrence of an extra argument—the accumulator—in the second form and the extra computational overhead in the base case. The details of the PROLOG encoding for the third tail-recursive form should be obvious. An imperative encoding is now given as promised.

#### 4.3. Imperative Encodings

In this section, the goal is to consider the reification of reduce under the transformation of a sequence  $\sigma \in (\Sigma^*, \wedge, \Lambda)$  to a sequence slice  $\sigma[1 \dots n]$  of fixed length  $n$ . Recall that the notation for slices was derived from that of a sequence considered as a vector in Chapter 2. Here, I view the sequence as a map and extend the notation to slices appropriately. Such reification will lead directly to an imperative encoding where the sequence is accessed by index. Consider the specification

$$\begin{aligned} \oplus / \langle a, b \mid \tau \rangle &= \oplus / \langle a \oplus b \mid \tau \rangle \\ \oplus / \langle e \rangle &= e \end{aligned}$$

## TAIL-RECURSIVE ALGORITHMS

where, under the pattern matching assumption, all cases are covered. Then this may be interpreted as an *in situ* reduction in the sense that the first location of the sequence is used to accumulate the result. Thus, a possible reification is

$$\text{reduce}(\sigma[i \dots n]) = \text{reduce}(\sigma[i \mapsto \sigma[i] \oplus \sigma[i + 1]] \cup \sigma[i + 2 \dots n])$$

and, in general,

$$\text{reduce}(\sigma[1] \cup \sigma[i \dots n]) = \text{reduce}(\sigma[1 \mapsto \sigma[1] \oplus \sigma[i]] \cup \sigma[i + 1 \dots n])$$

Clearly, one has the initialisation ‘ $i \leftarrow 2$ ;’ and the loop construct is

```
for  $i := [2 \dots n]$  do
   $\sigma[1] \leftarrow \sigma[1] \oplus \sigma[i]$ ;
end for
return  $\sigma[1]$ 
```

Note, of course, that this is equivalent to the (slightly) optimised form

```
 $u \leftarrow \sigma[1]$ ;
for  $i \in [2 \dots n]$  do
   $u \leftarrow u \oplus \sigma[i]$ ;
end for
return  $u$ 
```

Note that this is a ‘standard’ imperative encoding technique, used for example in computing the sum of the elements of an array. The variable  $u$  clearly functions as an accumulator. But observe that the initialisation condition is ‘ $u \leftarrow \sigma[1]$ ;’ and not the usual ‘ $u \leftarrow 0$ ;’ that one finds in text books, for example. This might seem to be a ‘trivial’ observation. However, in the case of an Ada generic unit, it does mean that it is unnecessary to provide a formal generic parameter for the zero of the structure. This is but one more simple illustration of the practical importance of approaching imperative programming from a study of the theory of tail-recursive algorithms.

From the tail-recursive form generated by the Arzac and Kodratoff method, one may derive a different imperative encoding, taking pairs of elements at a time:

$$\text{reduce}(\sigma[1 \dots n])x = \text{reduce}(\sigma[3 \dots n])(x \oplus (\sigma[1] \oplus \sigma[2]))$$

and, in general,

$$\text{reduce}(\sigma[i \dots n])x = \text{reduce}(\sigma[i + 2 \dots n])(x \oplus (\sigma[i] \oplus \sigma[i + 1]))$$

The base equations may be transformed into

case  $len \sigma$  is odd:  $reduce(\sigma[n \dots n])x = x \oplus \sigma[n - 1]$

case  $len \sigma$  is even:  $reduce(\sigma[n - 1 \dots n])x = x \oplus (\sigma[n - 1] \oplus \sigma[n])$

and thus the optimised imperative encoding is

```

x ← u;
i ← 1;
while i < n do
  x ← x ⊕ σ[i] ⊕ σ[i + 1];
  i ← i + 2;
end while
if even(n)
  then return x
  else return x ⊕ σ[n]

```

#### 4.4. Extension of the Reduction Operator

I have already demonstrated that  $\Sigma^*$ -homomorphisms may be defined in terms of reduction. In view of its foundational importance, a natural question arises. Is it possible to extend the definition of reduction and achieve an even greater unification?

Consider once again the specification of reduce:

$$\oplus / \langle \langle a, b \rangle \wedge \tau \rangle = \oplus / \langle \langle a \oplus b \rangle \wedge \tau \rangle$$

$$\oplus / \langle a, b \rangle = a \oplus b$$

$$\oplus / \langle e \rangle = e$$

The binary operator  $\oplus$  may be defined to be a function (Iverson 1979):

$$f(x, y) = x \oplus y$$

Then the reduction homomorphism may be written in the form

$$f / \langle \langle a, b \rangle \wedge \tau \rangle = f / \langle \langle f(a, b) \rangle \wedge \tau \rangle$$

$$f / \langle a, b \rangle = f(a, b)$$

$$f / \langle e \rangle = e$$

Thus functions of two arguments may frequently be treated as binary operators. Consequently they may be extended to sequences of arguments using reduction. We have already seen this to be the case for addition and multiplication of numbers, maximum and minimum of numbers, distributed concatenation of sequences, distributed union of sets and distributed extend of maps. It is frequently the case

## TAIL-RECURSIVE ALGORITHMS

that this *Gestalt* shift between function and operator is missed. A plausible reason has to do with the choice of notation and naming. Whereas it is straightforward to invent names at will for functions, the number of operator symbols is limited, as much by technology as by the concerns for clarity. One more example should suffice. Let  $\text{gcd}$  denote the function that computes the greatest common divisor of a pair of numbers. Then  $\text{gcd}^{\text{cd}}/$  is the extension of  $\text{gcd}$  to a sequence of arguments, the algorithm being of the form cited by Knuth in (1981, 2:324).

### 4.5. Application to Polynomial Evaluation

Now let us consider the application of reduction to the evaluation of a polynomial. It is assumed that a polynomial is to be represented as a sequence of coefficients. Thus  $P_3(x) = a_3x^3 + a_2x^2 + a_1x + a_0$  is represented as  $\langle a_3, a_2, a_1, a_0 \rangle$ . Note that the “natural” representation of such a polynomial is in reverse order  $\langle a_0, a_1, a_2, a_3 \rangle$ , where a polynomial is considered to be an ‘infinite’ sequence of coefficients ‘most of which are zero’ (Mac Lane and Birkhoff 1979, 105).

Now, if one defines a function  $h(x, a, b) = ax + b$ , i.e.,  $h[x](a, b) = ax + b$ , then the extended reduction algorithm

$$\begin{aligned} h[x]/(\langle a, b \rangle \wedge \tau) &= h[x]/(\langle h[x](a, b) \rangle \wedge \tau) \\ h[x]/\langle e \rangle &= e \end{aligned}$$

is Horner’s method (i.e., nested multiplication) for polynomial evaluation.

### 4.6. Prolog Implementation

To handle polynomial evaluation as described above, an extension to the reduce predicate is required:

```
reduce([H, X], [E], E).
reduce([H, X], [E, F | Tail], Result) :-
    Function = .. [H, X, E, F, Val],
    call(Function),
    reduce([H, X], [Val | Tail], Result).
```

Then the function  $h$  may be defined by

```
h(X, A, B, Result) :-
    XA is X * A,
    Result is XA + B.
```

and a predicate for polynomial evaluation may be presented in the form:

$$\text{polyeval}(\text{Poly}, X, \text{Val}) :- \text{reduce}([h, X], \text{Poly}, \text{Val}).$$

This is another simple example of the power of PROLOG to capture very concisely the encoding of generic algorithms.

#### 4.7. The Restricted Reduction Operator

The term reduction, used in the previous sections, has signified the reduction of a structure of elements to a single element by repeated application of an operator  $\oplus$  over the sequence. Such reduction is that of Iverson and Backus. However, there are many algorithms which reduce structures in size, one of the largest classes being those which reduce sequences by length 1. Such algorithms operate on pairs of elements. Let  $f: \Sigma \times \Sigma \rightarrow T$  be a total function in  $F = \Sigma \times \Sigma \xrightarrow{m} T$ , i.e.,  $F = T^{\Sigma^2}$ .

Then the function  $f$  may be applied to juxtaposed pairs of elements in a sequence.

Formally,  $//: F \rightarrow (\Sigma^*, \wedge) \rightarrow (T^*, \wedge)$ :

$$f//\langle a, b \mid \tau \rangle = \langle f(a, b) \rangle \wedge f//\langle b \mid \tau \rangle$$

$$f//\langle a, b \rangle = f(a, b)$$

$$f//\langle e \rangle = e$$

Note carefully the first line of the definition. It is this which distinguishes this class of algorithms from ‘ordinary’ reduction. I have called it a restricted reduction operator. One might quibble with the choice of terminology. There is an equally good case to be made for calling it an extended application operator. In addition I have chosen to use the form  $\langle e \mid \tau \rangle$  in place of  $\langle e \rangle \wedge \tau$  to represent a non-empty sequence.

EXAMPLE 4.8. A polyline is defined to be a sequence of points  $p$ ,  $\text{len } p \geq 2$ . Let  $(-, -)$  denote the function that constructs an ordered pair from its arguments. For instance,  $(-, -)[x, y] = (x, y)$ . Then  $(-, -)//p$  specifies the sequence of line segments that a computer graphics system might be expected to produce from a polyline.

Yet a further extension is obtained by considering parameterised functions  $f: \mathbf{R} \rightarrow \Sigma \times \Sigma \rightarrow T$  with

$$f[[u]]//\langle a, b \rangle \wedge \tau = \langle f[[u]](a, b) \rangle \wedge f[[u]]//\langle b \mid \tau \rangle$$

$$f[[u]]//\langle e \rangle = \Lambda$$

An alternative notation for  $f[[u]]$  which I use frequently in manuscript form is  $f_u$ .

## TAIL-RECURSIVE ALGORITHMS

EXAMPLE 4.9. Let  $\mathcal{L}_u$  denote a linear interpolation operator indexed with respect to some parameter  $u$ . Then typical applications of the latter are

- $\mathcal{L}_u: (a, b) \mapsto (1-u)a + ub$  is used in the *de Casteljau* algorithm for Bézier curve computation;
- $\mathcal{L}_u: (f(a), f(b)) \mapsto \frac{u}{2}(f(a) + f(b))$  is the trapezoidal rule used in numerical integration.

### 4.8. Imperative Encoding

Assuming a fixed dimensional sequence  $\sigma[1 \dots n]$  of length  $n$ , and that elements of  $\sigma[1 \dots n]$  are to be overwritten on the left, then an imperative encoding of the restricted reduction operator is given by the following:

```
-- f//:  $\sigma[1 \dots n] \mapsto \tau[1 \dots n - 1]$   
for  $i = [1 \dots n - 1]$  do  
     $\sigma[i] \leftarrow f(\sigma[i], \sigma[i + 1]);$   
end for  
return  $\sigma[1 \dots n - 1]$ 
```

Note that the resultant sequence is  $\sigma[1 \dots n - 1]$ , i.e., the index  $n - 1$  must be recorded lest there be subsequent processing. A remark on the comment at the head of the encoding is necessary. From a practical point of view, in developing encodings which overwrite structures, I always use an intermediate structure first. In this example, the key lines of the intermediate stage would be ‘ $\tau[i] \leftarrow f(\sigma[i], \sigma[i + 1]);$ ’, and ‘return  $\tau[1 \dots n - 1]$ ’. Only when I am satisfied that I understand the direction of overwriting, as expressed by the forms of the index, do I produce the final encoding. There are more detailed examples of this in Chapter 7 on curves and surfaces.

## 5. Application

The reduction operator effectively reduces a non-empty sequence of elements to a singleton sequence of one element. I remarked that the extended reduction operator, which reduces a sequence of  $n$  elements to a sequence of  $n - 1$  elements, might also be referred to as an extended application operator. The application operator in question is, of course, the  $-^*$  functor. It is one of the basic functionals of FP (Backus 1978).

Recall that if  $f: \Sigma \longrightarrow T$  is a total function in the function set  $F = T^\Sigma$ , then the function  $f$  may be applied to sequences of elements  $e \in \Sigma$ :

$$\begin{aligned} f^*(\sigma \wedge \tau) &= f^*\sigma \wedge f^*\tau \\ f^*\Lambda &= \Lambda \end{aligned}$$

with the constructive step given by

$$f^*(\langle e \rangle \wedge \tau) = \langle f(e) \rangle \wedge f^*\tau$$

The use of notation such as  $f^*$  to denote application of  $f$  over sequences may appear a little strange to the programmer. An alternative notation is to use an operator symbol such as  $\alpha$ . Thus, the application of  $f$  over a sequence may be defined by:

$$\begin{aligned} \alpha[f](\langle e \rangle \wedge \tau) &= \langle f(e) \rangle \wedge \alpha[f]\tau \\ \alpha[f]\Lambda &= \Lambda \end{aligned}$$

The form  $f^*$  is simply a very expressive notational variant of  $\alpha[f]$ . Another variant might just as well be  $\alpha_f$ . It is also instructive to write out in full the signature of the application operator:

$$\alpha: T^\Sigma \longrightarrow \Sigma^* \longrightarrow T^*$$

where  $f \in T^\Sigma$  is one particular total function.

I would like to make some remarks on the interpretations of the *form* of the application operator presented. Specifically

$$\alpha[f](\langle e \rangle \wedge \tau) = \langle f(e) \rangle \wedge \alpha[f]\tau$$

suggests that application is an iterator over the sequence with respect to the function  $f$ . Indeed this is a frequent occurrence in algorithmic specifications. One may also interpret the form as the processing of an input sequence (or stream),  $\langle e \rangle \wedge \tau$ , by a function  $f$  and the production of the corresponding output sequence. It is

## TAIL-RECURSIVE ALGORITHMS

the very insistence on the notion of specification that permits such wide-ranging observations.

Just as in the case of the reduction operator, the above recursive definition of the application operator is a natural tail-recursive form. This is exhibited most clearly by the corresponding PROLOG encoding given next.

### 5.1. Prolog Implementation

The encoding of  $\alpha$  in PROLOG is rather straightforward and should not require any comment:

```

apply(F, [], []).
apply(F, [H | T], [Val | Result]) :-
    Function =.. [F, H, Val],
    call(Function),
    apply(F, T, Result).

```

### 5.2. Tail-recursive Form

Again, just as in the case of the reduction operator, we can invoke the Arzac and Kodratoff method to obtain a tail-recursive form of  $\alpha$  which uses an accumulator to give

$$\bar{\alpha}[[f, \langle e \rangle \wedge \tau]]x = \bar{\alpha}[[f, \tau]](x \wedge \langle f(e) \rangle)$$

$$\bar{\alpha}[[f, \Lambda]]x = x$$

with signature

$$\bar{\alpha}: \mathcal{P}(\Sigma \xrightarrow{m} T) \longrightarrow \Sigma^* \longrightarrow T^*$$

where the relationship between the tail-recursive function  $\bar{\alpha}$  and the original application operator is

$$\bar{\alpha}[[f, \sigma]]x = x \wedge \alpha[[f]]\sigma$$

Setting  $x = \Lambda$  gives  $\alpha[[f]]\sigma = \bar{\alpha}[[f, \sigma]]\Lambda$ .

### 5.3. Imperative Encoding

The reification of  $\alpha$  with respect to the transformation of sequences to sequence slices is also rather trivial. First, in the general case, assume  $f: \Sigma \longrightarrow T$ . Considering the definition  $f^*(\langle e \rangle \wedge \tau) = \langle f(e) \rangle \wedge f^*\tau$ , one replaces the anonymous sequence  $\langle e \rangle \wedge \tau$  with the named sequence slice  $\sigma[i \dots n]$ , and the resultant sequence  $\langle f(e) \rangle$  with  $\tau[i \mapsto f(\sigma[i])]$ . Thus,

$$f^*\sigma[i \dots n] = \tau[i \mapsto f(\sigma[i])] \cup f^*\sigma[i + 1 \dots n]$$

This translates immediately to the imperative encoding:

```

for  $i = [1 \dots n]$  do
   $\tau[i] \leftarrow f(\sigma[i]);$ 
end for
return  $\tau[1 \dots n]$ 

```

Given the particular case that  $f: \Sigma \longrightarrow \Sigma$  and the constraint to overwrite, one has:

```

for  $i = [1 \dots n]$  do
   $\sigma[i] \leftarrow f(\sigma[i]);$ 
end for
return  $\sigma[1 \dots n]$ 

```

### 5.4. Extended Application

The very notion that  $f$  is unary in the expression  $f^*$  or  $\alpha[f]$  suggests an extension of the application operator to binary functions (or operators). For example, let  $\mathbf{v}_1$  and  $\mathbf{v}_2$  denote two  $n$ -dimensional vectors over the real numbers  $\mathbf{R}$ , represented by sequences. Define the binary operator of addition over reals as the function  $g: \mathbf{R}^2 \longrightarrow \mathbf{R}$  such that  $g(x, y) \mapsto x + y$ . Then the vector sum  $\mathbf{v}_1 \oplus \mathbf{v}_2$  may be defined as the application  $\alpha[g](\mathbf{v}_1, \mathbf{v}_2)$ . In detail

$$\begin{aligned} \alpha[g](\langle x \rangle \wedge \sigma, \langle y \rangle \wedge \tau) &= \langle g(x, y) \rangle \wedge \alpha[g](\sigma, \tau) \\ &= \langle x + y \rangle \wedge \alpha[g](\sigma, \tau) \\ \alpha[g](\Lambda, \Lambda) &= \Lambda \end{aligned}$$

In the previous two Chapters, there were many instances of specifications which required the combination of both reduction and application operators. Let me conclude this section by giving yet one more—the inner product of two vectors. Define

## TAIL-RECURSIVE ALGORITHMS

the function  $g: (x, y) \mapsto x \cdot y$  which gives the product of two real numbers. Then  $^+/\alpha[[g]](\mathbf{v}_1, \mathbf{v}_2)$  is the inner product of two vectors.

### 6. Summary

In concentrating on reduction and application operators within the context of executable *Meta-IV* specifications, I may have given the impression of reinventing APL or FP. For, to a large extent, both are essentially just languages ideally suited for executable *Meta-IV* specifications based on the sequence domain. Were I to include more material on sets and maps, then I would possibly give the impression of reinventing SETL. The focus in this Chapter has been placed elsewhere. I have demonstrated that the theory of tail-recursive algorithms is just as interesting from a mathematical viewpoint as the algebra of monoids and their homomorphisms. From the perspective of bridge-building from representations to encodings, I have shown that the same specification may equally well be transformed into a while loop program or into a sequence of PROLOG clauses.

Probably the most important result, from the point of view of conceptual models, is the *Gestalt* shift from the denotational semantics of imperative programs to the expression of tail-recursive algorithms and vice-versa. It was just this simple observation that led to the ‘discovery’ that a tail-recursive form gave the while loop invariant.

The name ‘tail-recursive’ comes from the notion of the tail of a list or sequence and I have been for the most part interested in algorithms over sequences in the latter part of this Chapter. However, looking back at Chapter 2, I invariably employed the term when speaking about sets and maps as well. For example, I would consider reduction over sets and maps as being tail-recursive. If ever confusion should arise from its use then I would probably adopt some term such as *T*-recursion, where the *T* is intended to denote capital  $\tau$ .

The world of computing is a veritable tower of Babel of programming languages and formal specification languages. There is an enormous wealth of learning hidden

in the complexities of specific notations. It is tedious to have to learn some new notation in order to grasp the essentials of what might appear to be very interesting worthwhile algorithms. Rather than invent another specification language, I felt that it was more important to build on well-established ground—the *VDM Meta-IV*. However, in doing so, I also wanted to be sure that nothing significant was omitted, that the ‘operator philosophy’ behind languages such as APL and FP might also be embraced.

Having demonstrated how the notation of the Irish School of the *VDM* might be employed effectively in the presentation of algorithms, I must now introduce some of the essentials of the *method* of the Irish School. I have said much about the relation between *Meta-IV* and algebra. It is now time to look at the other side of the bridge—to the world of applications.



# Chapter 5

## The Method of the Irish School

### 1. Introduction

Published papers on algorithms, whether as articles in journals, or as sections of books, are apt to fall foul of a Lakatos-type criticism that they follow a certain obligatory style of presentation, the ‘deductivist style’:

“This style starts with a painstakingly stated list of *axioms*, *lemmas* and/or *definitions*. The axioms and definitions frequently look artificial and mystifyingly complicated. One is never told how these complications arose. The list of axioms and definitions is followed by the carefully worded *theorems*. These are loaded with heavy-going conditions; it seems impossible that anyone should ever have guessed them. The theorem is followed by the *proof*” (Lakatos 1976, 142).

In the field of computing, one finds not only the deductivist style but also the ‘pragmatist style’ wherein algorithms are already programs, either presented in some pseudo-code such as that derived from a member of the ALGOL 60 family or an actual programming language.

The use of formal methods to specify algorithms as used in this thesis is an attempt to demonstrate that there is a bridge, a golden mean, between the deductivist style and the pragmatist style. Even the very juxtaposition of ‘specification’ and ‘algorithm’ may provoke criticism.

What are the axioms, lemmas, definitions, theorems, and proofs of formal specifications? Note that I do **not** say formal *methods*. For here I am defining precisely my own formal method, an extension to the *VDM* and one that relies heavily on the inspiration of Lakatos, Pólya and Popper.

# THE METHOD OF THE IRISH SCHOOL

## 1.1. Conjecture, Proof and Counter-example

Consider the specification of a dictionary. Whether one views dictionaries as books or uses dictionaries on computers, either properly understood as analogues of the book kind or another name for symbol tables, say, then the following conjecture might be put forward

CONJECTURE 5.1. *A dictionary may be modelled as a map from words to sets of definitions.*

Formally

$$DICT = WORD \xrightarrow{m} \mathcal{P}DEF$$

Note that I have avoided mentioning the adverb ‘adequately’. I do not say whether the domains are finite or infinite. I am happy to suppose that they are finite, or if you will, denumerably infinite. How shall I test this conjecture? Can I give a proof? What form should the proof take? If I find a proof, what would it prove?

I can at least begin by giving examples of dictionaries that fit the specification and consider whether they support the conjecture or refute it. This is the exploratory phase of problem solving (Pólya [1962] 1981, 2:104).

EXAMPLE 5.1. The dictionary  $\delta = \theta$  is a perfectly adequate dictionary with no words and no definitions. It is in fact the cheapest possible one that you could buy.

EXAMPLE 5.2. Consider the dictionary  $\delta$  such that there are words  $w_j, w_k \in \delta$  with the properties that  $\delta(w_j) \neq \emptyset$ ,  $\delta(w_k) \neq \emptyset$ , and  $\delta(w_j) \cap \delta(w_k) \neq \emptyset$ . Such a dictionary has the form

$$\delta = [\dots, w_j \mapsto \{\dots, d, \dots\}, \dots, w_k \mapsto \{\dots, d, \dots\}, \dots]$$

This is a dictionary such that two distinct words  $w_j, w_k$ , have at least one definition in common. Such words are called synonyms.

EXAMPLE 5.3. It is possible to have a dictionary of words, none of which have definitions

$$\delta = [w \mapsto \emptyset \mid w \in WORD]$$

This is clearly isomorphic to the ‘spelling-checker’ dictionary which we have already considered at length.

It usually does not take long for someone to object that the model does not ‘adequately’ cover a real dictionary as book where the words are arranged in lexicographic order. Maps do not have order. This may at first be considered as a counter-example. It pinpoints clearly the rôle that the concept of order plays in the concept of dictionary. To see that order is incidental to the concept of dictionary requires some convincing argument. It is precisely here that the essence of abstraction is made apparent and the rôle that the *VDM* plays in building up conceptual models with respect to formal specification.

Accepting that one wishes to proceed with the notion of ordered dictionary, then the new model

$$DICT_1 = (WORD \times \mathcal{P}DEF)^*$$

might be introduced. This might be stated as

CONJECTURE 5.2. *An ordered dictionary may be modelled as a sequence of ordered pairs  $(w, ds)$ , where  $w$  is a word and  $ds$  the set of corresponding definitions.*

But immediately I have problems to solve for it specifies more than I really want. For instance, this domain includes

$$\delta_1 = \langle (w, \{d\}), (w, \{d\}), (w, \{d, d'\}) \rangle$$

Effectively my model is a relation and it permits duplicates. What shall we do about this? (Incidentally, the PROLOG database, which is just a dictionary in our sense but modelled slightly differently to  $DICT_1$ , may in fact be implemented this way. A user may inadvertently or deliberately introduce the same fact twice). In the classical *VDM* it is conventional to restrict the domain of validity of the conjecture by introducing an invariant.

In this Chapter, I am going to focus on two application areas: a file system case study, and a bill of materials, the specifications of which have been developed and published by others. Both examples have had a major formative influence on the development of the method of the Irish School of the *VDM*. Equally important, especially because of its simplicity, has been an in-depth study of the dictionary model. It was the primary benchmark for testing the applicability of the operator calculus and details may be found in Appendix A.

# THE METHOD OF THE IRISH SCHOOL

## 2. Invariants

In the *VDM Meta-IV* a domain invariant is used to specify the properties that a particular domain must always satisfy. Let *MODEL* be the definition of some domain given in the form

$$MODEL = \dots$$

The corresponding specification of the invariant has the form

$$\begin{aligned} \text{inv-}MODEL: MODEL &\longrightarrow \mathbf{B} \\ \text{inv-}MODEL(\mu) &\triangleq \dots \end{aligned}$$

EXAMPLE 5.4. Consider the model  $MODEL = \Sigma^*$ . This unrestricted domain, i.e., without an invariant, is exactly that of the free monoid over the alphabet  $\Sigma$ , the basis of the theory of Formal Languages.

EXAMPLE 5.5. Let  $MODEL_0 = \mathcal{P}\Sigma$  denote some domain. If we wish to consider the sequence as an ‘implementation’ model for sets, then as a first approximation we might wish to consider  $MODEL_1 = \Sigma^*$  subject to the invariant

$$\text{inv-}MODEL_1(\sigma) \triangleq |elems \sigma| = |\sigma|$$

which states that the domain of sequences we are considering is restricted to exactly those which do not contain duplicates. I have chosen to use  $\sigma \in \Sigma^*$  rather than  $\mu_1 \in MODEL_1$  for the argument of the invariant. There are many different ways in which to specify exactly this invariant. For example, why not choose the following?

$$\begin{aligned} \text{inv-}MODEL_1(\Lambda) &\triangleq true \\ \text{inv-}MODEL_1(\langle e \rangle \wedge \tau) &\triangleq \neg \chi_\tau(e) \wedge \text{inv-}MODEL_1(\tau) \end{aligned}$$

My current choice of the form of the invariant is exactly due to the form of the usual retrieve function that maps sequences to sets:

$$\text{retr-}MODEL_0(\mu_1) \triangleq elems \mu_1$$

Let  $\sigma, \sigma'$  denote two sequences such that  $\sigma'$  is derived from  $\sigma$  by removing all duplicates. Since,  $elems \sigma = elems \sigma'$ , and, in general,  $\sigma \neq \sigma'$ , then the invariant is ‘stronger’ than the retrieve function. The relationship between the invariant and

the retrieve function is exhibited by the commuting diagram:

$$\begin{array}{ccc}
 \mathcal{P}\Sigma & \xrightarrow{\text{card}} & \mathbf{N} \\
 \uparrow \text{elems} & & \uparrow \leq \\
 \Sigma^* & \xrightarrow{\text{len}} & \mathbf{N}
 \end{array}$$

In other words, in the general case, it is always true that the cardinality of the set of elements of a sequence is less than or equal to the length of the sequence:

$$|\text{elems } \sigma| \leq |\sigma|$$

All that the invariant essentially says is that we are only prepared to admit consideration of sequences which satisfy the equality relation given above.

EXAMPLE 5.6. One of the classic published *VDM* specifications is that of a file system (Bjørner and Jones 1982, 353), an invariant of which is discussed at length here.

A complete analysis of this example is presented in Appendix C. It is my belief that specifications, once written and proven, are never revisited for the purpose of re-proving and, if indeed they are revisited for such a purpose, no new methods for proof are ever applied, or if they are, they are never published. Contrast this sorry plight of affairs with the history of ‘proving’ and repolishing in Mathematics (cf., Lakatos 1976). In order to grasp written *VDM Meta-IV* specifications I have been obliged to seek out new methods of proof, being completely dissatisfied with the non-intuitive proofs published. That I am obliged to teach such material has been a driving force behind the search. In the example under consideration, I discuss the method of attack.

At the most abstract level, the file system  $FS_0$  is given by the domain equations

$$FS_0 = Fn \xrightarrow{m} FILE$$

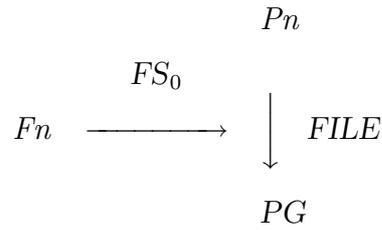
$$FILE = Pn \xrightarrow{m} PG$$

subject to the invariant

$$\text{inv-}FS_0(\varphi) \triangleq \text{true}$$

Now, the important point to note here is that *any* instance of the model  $FS_0$  satisfies the invariant. All of the difficulties that I have had with the file system case study may be traced back to this invariant which is erroneous, as will be demonstrated later. This model of the file system  $FS_0$  may be depicted graphically by

## THE METHOD OF THE IRISH SCHOOL

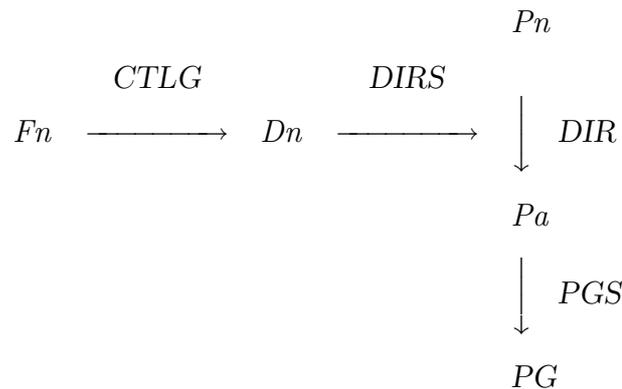


At the first level of reification, the model  $FS_1$  is introduced with the domain equations

$$\begin{aligned}
 FS_1 &:: CTLG \times DIRS \times PGS \\
 CTLG &= Fn \xleftrightarrow[m]{\leftrightarrow} Dn \\
 DIRS &= Dn \xleftrightarrow[m]{\leftrightarrow} DIR \\
 PGS &= Pa \xrightarrow[m]{\rightarrow} PG \\
 DIR &= Pn \xleftrightarrow[m]{\leftrightarrow} Pa
 \end{aligned}$$

Here again we have a subtle error. In their concern to be precise, the authors chose to use injective map notation. This causes a technical problem when a ‘create’ file operation is invoked twice in succession.

From the perspective of *theorem* and *proof*, each model is a separate domain of discourse with respect to the operations that one expects to perform on a file system. It is my concern here to investigate the domain of discourse of the ‘union’ of both models and to consider initially the rôle of the invariant and the retrieve function. Before introducing these latter specifications, I give a diagram which exhibits more clearly the structure of the domain  $FS_1$ :



Let me now first present the invariant  $inv\text{-}FS_1$  as published with slight modifications to accommodate the notation of the Irish School of the VDM. I use Greek letters to denote maps:  $\kappa \in CTLG$ ,  $\tau \in DIRS$ ,  $\delta \in DIR$ ,  $\varpi \in PGS$  and Roman letters for elements of the other domains. The invariant now reads

$$\begin{aligned}
 \text{inv-}FS_1: FS_1 &\longrightarrow \mathbf{B} \\
 \text{inv-}FS_1(\kappa, \tau, \varpi) &\triangleq \\
 &(\text{rng } \kappa = \text{dom } \tau) \\
 &\wedge \cup / \{ \text{rng } \delta \mid \delta \in \text{rng } \tau \} = \text{dom } \varpi \\
 &\wedge (\forall pa \in \text{dom } \varpi)(\exists! dn \in \text{dom } \tau)(pa \in \text{rng } \tau(dn))
 \end{aligned}$$

where  $\cup /$  denotes the distributed union of a set of sets. The corresponding retrieve function is

$$\begin{aligned}
 \text{retr-}FS_0: FS_1 &\longrightarrow FS_0 \\
 \text{retr-}FS_0(\kappa, \tau, \varpi) &\triangleq \\
 &[fn \mapsto [pn \mapsto \varpi((\tau(\kappa(fn))))(pn)) \mid pn \in \text{dom } \tau(\kappa(fn))] \mid fn \in \text{dom } \kappa]
 \end{aligned}$$

In my analysis of the invariant I am concerned to exhibit a close correspondence between it and the retrieve function. In the process, I expect to arrive at a better invariant and a better retrieve function, better in the sense that they do not rely on implicit forms. I shall consider the invariant to be composed of three subinvariants, each of which is separated by the ‘logical and’ and shall assume that they are mutually independent. My rationale for doing so is that the invariant is like a theorem or conjecture and the subparts are the lemmas and I fully intend to follow the heuristic method of ‘Proofs and Refutations’ in tackling the problem (Lakatos 1976, 127). The *cognoscenti* of the case study will immediately express horror at the suggestion, for, as shall be seen, the second and third parts are interrelated. But I shall blithely pretend that it is not the case, for indeed, that is what usually happens in the ‘discovery’ of specifications.

## 2.1. Subinvariant 1

With the aid of the appropriate diagrams, it is clear that the expression

$$\text{rng } \kappa = \text{dom } \tau$$

in the invariant is simply a statement that the pair of maps  $(\kappa, \tau)$  is composable. Therefore,  $DIRS \circ CTLG$  is a domain of composable maps and, consequently

$$DIRS \circ CTLG = Fn \overset{m}{\leftrightarrow} DIR$$

I am, therefore, always justified in writing  $\tau \circ \kappa$  at will. Comparing this with the first model, it is evident that this is structurally similar to  $FS_0$ . There is the worrying part that a retrieve function applied to  $\tau \circ \kappa$  would seem to give me  $\varphi_0$ . I conclude

## THE METHOD OF THE IRISH SCHOOL

that there is something fundamentally wrong conceptually with the way in which the two models have been constructed. But that is a separate issue.

### 2.2. Subinvariant 2

Now consider the second expression in the invariant:

$$\cup/\{rng \delta \mid \delta \in rng \tau\} = dom \varpi$$

The first question that one asks is obviously ‘what does it mean’? Formal specifications such as this are usually difficult to grasp at first glance. Even this much is true of the specifications that I write myself and then try to understand much later. How should one proceed? Let us begin by assuming that there are no typographical errors. To solve the problem we must look at the evidence. The occurrence of ‘rng’ on the left-hand-side and ‘dom’ on the right-hand-side suggests that it is a statement about the composability of maps. Consequently, I deduce that the statement is simply that the pair of maps,  $(\varphi, \varpi)$ , is composable, i.e.,  $rng \varphi = dom \varpi$ , where

$$rng \varphi = \cup/\{rng \delta \mid \delta \in rng \tau\}$$

This supposition seems to be confirmed by referring back to the diagram. The next problem to be addressed is the identification of the map  $\varphi$ . At this stage I turn to some examples.

#### 2.2.1. Exploring the Problem Domain

Since  $\tau$  is a directory system, then  $rng \tau$  is a set of directories.

COUNTER-EXAMPLE 1. Consider the directory system  $\tau = [dn_1 \mapsto \delta_1, dn_2 \mapsto \delta_2]$ , where

$$\delta_1 = [pn_1 \mapsto pa_1, pn_2 \mapsto pa_2]$$

$$\delta_2 = [pn_2 \mapsto pa_1, pn_3 \mapsto pa_3]$$

Then  $rng \tau = \{\delta_1, \delta_2\}$ .

This is actually a counter-example to the third part of the invariant, which we shall see later, expressly prohibits that two different directories share the same page. Now from our example, we have  $rng \delta_1 = \{pa_1, pa_2\}$  and  $rng \delta_2 = \{pa_1, pa_3\}$ . Consequently, the map  $\varphi$  seems to be some sort of combination of  $\delta_1$  and  $\delta_2$ . I might stop here, having convinced myself that the statement in question is one of the composability of maps. But, I must continue with the problem, for I wish to

show conclusively, that I can replace the given statement with  $\text{rng } \varphi = \text{dom } \varpi$ . In other words, I wish to construct the invariant in another way.

It is clear that neither  $(\delta_1, \varpi)$  nor  $(\delta_2, \varpi)$  are pairs of composable maps. Now we have another problem to address: what sort of combination of  $\delta_1$  and  $\delta_2$  will give  $\varphi$ ? The example has been so constructed that  $\delta_1 \neq \delta_2$ , which is essential in order to have a directory system of two distinct elements and  $\text{dom } \delta_1 \cap \text{dom } \delta_2 \neq \emptyset$ , which rules out the possibility of using the extend operator. (We shall return to this latter point, for it is the key to the solution of the problem that I am seeking). It may also be demonstrated that neither map override nor map symmetric difference will do. In fact, everything that I try seems doomed to failure. The stumbling block lies in the fact that  $\text{dom } \delta_1 \cap \text{dom } \delta_2 \neq \emptyset$ . Any combination of  $\delta_1$  and  $\delta_2$  will appear to give a relation. But a relation may be modelled as a map, an important subject which is dealt with later in the Chapter. Thus, I introduce the operator  $\uplus$  which I have yet to define, and writing the directory maps in the form

$$\begin{aligned}\delta_1 &= [pn_1 \mapsto \{pa_1\}, pn_2 \mapsto \{pa_2\}] \\ \delta_2 &= [pn_2 \mapsto \{pa_1\}, pn_3 \mapsto \{pa_3\}]\end{aligned}$$

I have  $\varphi = \delta_1 \uplus \delta_2 = [pn_1 \mapsto \{pa_1\}, pn_2 \mapsto \{pa_1, pa_2\}, pn_3 \mapsto \{pa_3\}]$  and the pair of maps  $(\delta_1 \uplus \delta_2, \varpi)$  is composable. Without going into the details here of the mechanism of composition, let me just say that it is similar to the manner in which one approaches the composition of transition functions in non-deterministic finite state machines. For the example in question

$$\varpi \circ \varphi = [pn_1 \mapsto \{pg_1\}, pn_2 \mapsto \{pg_1, pg_2\}, pn_3 \mapsto \{pg_2\}]$$

I have found the  $\varphi$  that I was seeking. But as is customary, solving one problem usually leads to more.

The next immediately obvious problem is that a map of the form  $\delta_1 = [pn_1 \mapsto \{pa_1\}, pn_2 \mapsto \{pa_2\}]$  does not belong to the domain  $DIR = Pn \xleftrightarrow{m} Pa$ . But I am encouraged with the progress that I have made so far and thus introduce an auxilliary domain of ‘extended’ directories into the specification:

$$XDIR = Pn \xrightarrow{m} \mathcal{P}Pa$$

Although expressed in map form,  $XDIR$  is, in fact, a domain of relations. I must now provide a transfer function that maps a directory  $\delta$  into an extended directory  $\xi$ .

## THE METHOD OF THE IRISH SCHOOL

We have seen this sort of thing before. Let  $f: e \mapsto \{e\}$ . Then  $(-, f)$  is the required transformation. Let  $\xi = (-, f)\delta$ . Then  $\cup / \text{rng } \xi = \text{rng } \delta$  follows. A notational point is in order here. I will have need to refer to the transformation  $(-, f)$  frequently in subsequent analysis. Instead of writing  $(-, f)\delta$ , I prefer to identify the transformation with the target element, writing  $\xi = \xi(\delta)$ . This is a classical mathematical convention.

Referring back to the original problem posed by the invariant, we are even more encouraged to proceed. The final task to be accomplished is the definition of the  $\uplus$  operator. Again looking ahead to the section on relations in the Chapter, we note that it has already been done. The  $\uplus$  operator is just the generalisation of the  $\oplus$  operator used in the ‘reltomap’ conversion algorithm. For completeness, I re-present it here in the form of the tail-recursive algorithm ‘merge’ over the domain *XDIR*:

$$\begin{aligned} \text{merge: } XDIR &\longrightarrow XDIR \longrightarrow XDIR \\ \text{merge}[\theta]\xi &\triangleq \xi \\ \text{merge}[[pn \mapsto pas] \cup \xi_1]\xi_2 &\triangleq \\ \chi[[pn]\xi_2 & \\ \rightarrow \text{merge}[\xi_1](\xi_2 + [pn \mapsto \xi_2(pn) \cup pas]) & \\ \rightarrow \text{merge}[\xi_1](\xi_2 \cup [pn \mapsto pas]) & \end{aligned}$$

Then  $\xi_1 \uplus \xi_2 \triangleq \text{merge}[\xi_1]\xi_2$ . Naturally, I can demonstrate that  $\uplus$  enjoys nice properties such as commutativity, associativity, etc. However, this would take me too far afield from my present concerns here.

### 2.2.2. Reconstruction of the Subinvariant

Let us now reconstruct the second expression in the invariant

$$\cup / \{\text{rng } \delta \mid \delta \in \text{rng } \tau\} = \text{dom } \varpi$$

Starting with a directory system

$$\tau = [dn_1 \mapsto \delta_1, dn_2 \mapsto \delta_2, \dots, dn_j \mapsto \delta_j, \dots, dn_t \mapsto \delta_t]$$

and applying the range operator gives

$$\text{rng } \tau = \{\delta_1, \delta_2, \dots, \delta_j, \dots, \delta_t\}$$

Conversion of each directory to an extended directory is given by an application of

the transformation  $\xi$

$$\begin{aligned} \mathcal{P}\xi \circ \text{rng } \tau &= \{\xi(\delta_1), \xi(\delta_2), \dots, \xi(\delta_j), \dots, \xi(\delta_t)\} \\ &= \{\xi_1, \xi_2, \dots, \xi_j, \dots, \xi_t\} \end{aligned}$$

and finally applying the reduction  $\uplus/$  gives the result

$$\varphi = \uplus/ \circ \mathcal{P}\xi \circ \text{rng } \tau$$

and the second expression in the invariant may be written as

$$\text{rng}(\uplus/ \circ \mathcal{P}\xi \circ \text{rng } \tau) = \text{dom } \varpi$$

This result has extremely important consequences for the conceptual model of the person writing formal specifications in the *VDM Meta-IV*. From both the domain equations

$$\begin{aligned} \text{DIR} &= Pn \xleftrightarrow[m]{\leftrightarrow} Pa \\ \text{PGS} &= Pa \xrightarrow[m]{\rightarrow} PG \end{aligned}$$

and the corresponding diagram one would be misled entirely into supposing that  $\text{PGS} \circ \text{DIR}$  is a domain of composable maps where

$$\text{PGS} \circ \text{DIR} = Pn \xrightarrow[m]{\rightarrow} PG$$

and conclude that  $\text{PGS} \circ \text{DIR}$  is structurally similar to *FILE* in the first model. It is clear from an analysis of the invariant that this is entirely wrong! This is somehow disquieting. Can it be an adverse reflection on the *Meta-IV* notation or the inadequacy of diagrams?

### 2.2.3. The Retrieve Function

Turning now to consider the retrieve function in the light of results achieved so far, we observe that for all  $fn \in \text{dom } \kappa$ , it is clear that  $\tau \circ \kappa(fn) = \delta$  and for all  $pn \in \text{dom } \delta$ ,  $\varpi(\delta(pn)) \in \text{PG}$ . Therefore, the retrieve function may be simplified to

$$\begin{aligned} \text{retr-FS}_0(\kappa, \tau, \varpi) &\triangleq \\ &\text{let } \delta = \tau \circ \kappa(fn) \text{ in} \\ &\text{let } pg = \varpi(\delta(pn)) \text{ in} \\ &[fn \mapsto [pn \mapsto pg \mid pn \in \text{dom } \delta] \mid fn \in \text{dom } \kappa] \end{aligned}$$

This is certainly much simpler than the original but still not in the form that I need for my operator calculus. I would have liked to have been able to write  $\varpi \circ \delta(pn)$  instead of  $\varpi(\delta(pn))$ . But we see that this would be an error. The pair of maps

## THE METHOD OF THE IRISH SCHOOL

$(\delta, \varpi)$  are not composable, except in a trivial system. Clearly, I can compose  $(\kappa, \tau)$  to give me half of the result that I require. But how do I obtain the rest? I can not use the  $\varphi$  map that I have constructed, since it is in reality a relation and the desired result must be a finite function in the usual sense. On the other hand it does have the nice property of ‘merging’ or ‘gluing’ together separate functions. It probably ought to be explored as an alternative means to obtain, at little cost, some of the results we require. Let us extend the example that we have used so far:

$$\begin{aligned}\kappa &= [fn_1 \mapsto dn_1, fn_2 \mapsto dn_2] \\ \varpi &= [pa_1 \mapsto pg_1, pa_2 \mapsto pg_2, pa_3 \mapsto pg_2]\end{aligned}$$

Now composing the ‘front-end’ gives

$$\tau \circ \kappa = [fn_1 \mapsto \delta_1, fn_2 \mapsto \delta_2]$$

and the back-end will be expected to give

$$\begin{aligned}\varpi_1 \circ \delta_1 &= [pn_1 \mapsto pg_1, pn_2 \mapsto pg_2] \\ \varpi_2 \circ \delta_2 &= [pn_2 \mapsto pg_1, pn_3 \mapsto pg_2]\end{aligned}$$

where  $\varpi_1$  and  $\varpi_2$  are submaps of  $\varpi$ . Referring back to the analysis of the invariant, it is clear that for the retrieve function I must avoid composing all the directories in  $\tau$ . Instead, I now must identify a splitting of  $\varpi$ . This involves application of the restriction operator. Clearly,

$$\begin{aligned}\varpi_1 &= \text{rng } \delta_1 \triangleleft \varpi = [pa_1 \mapsto pg_1, pa_2 \mapsto pg_2] \\ \varpi_2 &= \text{rng } \delta_2 \triangleleft \varpi = [pa_2 \mapsto pg_2, pa_3 \mapsto pg_2]\end{aligned}$$

The obvious solution then appears to be

$$\begin{aligned}\varphi_0 &= [fn_1 \mapsto \varpi_1 \circ \delta_1, fn_2 \mapsto \varpi_2 \circ \delta_2] \\ &= [fn_1 \mapsto (\text{rng } \delta_1 \triangleleft \varpi) \circ \delta_1, fn_2 \mapsto (\text{rng } \delta_2 \triangleleft \varpi) \circ \delta_2]\end{aligned}$$

The final immediate problem with respect to the retrieve function is to express this result in a single compact form. Or is it? Let us stop here and try to solve some other related problems. Historically, I turned my attention to verifying that the commands of version 1 of the file system preserved subinvariant 1. As I was doing so, the current problem was at the back of my mind. But even more niggling was the strange third part which I now address.

### 2.3. Subinvariant 3

Whatever about my dislike for the form of the second part of the invariant, the form of the third part looked most discouraging:

$$(\forall pa \in dom \varpi)(\exists! dn \in dom \tau)(pa \in rng \tau(dn))$$

I dislike  $\forall$  and particularly  $\exists$  as they are against the spirit of the constructive method I propose. Worse than asserting that something exists, we have insistence on uniqueness as well  $\exists!$ . However, I must confess that in developing an initial specification, I too use these logical operators, ultimately replacing them with the operators of the Irish School in order to do proofs. To assist me in unravelling the mystery of this specification, I studied at length the accompanying descriptive passage in the original text: “every page, understood as page address, is described in exactly one directory (that is belongs to exactly one file)”. Then I turned to my general example:

$$\tau = [dn_1 \mapsto \delta_1, dn_2 \mapsto \delta_2, \dots, dn_i \mapsto \delta_i, \dots, dn_j \mapsto \delta_j, \dots, dn_t \mapsto \delta_t]$$

$$dom \tau = \{dn_1, dn_2, \dots, dn_i, \dots, dn_j, \dots, dn_t\}$$

But the directory system  $\tau$  is injective! Therefore, corresponding to  $dom \tau$  we have

$$rng \tau = \{\delta_1, \delta_2, \dots, \delta_i, \dots, \delta_j, \dots, \delta_t\}$$

and the statement that there is a unique  $dn_j$  implies that there is a corresponding unique  $\delta_j$ . Without loss of generality, let us suppose that  $dn_j$  is the unique directory name asserted. Then since  $\delta_j$  is also an injective mapping, the third part of the invariant asserts that all the page addresses in  $dom \varpi$  are partitioned, each partition being denoted by  $rng \delta_j = rng \tau(dn_j)$ . Now working backwards and again using the fact that  $\delta_j$  is injective, we arrive at the conclusion that  $\tau$  is partitioned. Consequently, for every pair  $\delta_i$  and  $\delta_j$ , it must be the case that  $dom \delta_i \cap dom \delta_j = \emptyset$  and so I resolve my problem with the second part of the invariant.

#### 2.3.1. Subinvariant 2 Revisited

Since the directory system is partitioned, then each pair of directories is disjoint and so all the directories can be combined under distributed extend:

$$rng \tau = \{\delta_1, \delta_2, \dots, \delta_i, \dots, \delta_j, \dots, \delta_t\}$$

$$\cup / \circ rng \tau = \varphi$$

## THE METHOD OF THE IRISH SCHOOL

Hence the second subinvariant is simply

$$\text{rng } \circ^{\cup} / \circ \text{rng } \tau = \text{dom } \varpi$$

and this is very nearly the same as that given. Now the obvious question that must be posed: ‘Is my understanding of the invariant correct with respect to the original problem domain—that of specifying the file system’? For me as an individual, it does not matter, since doing specifications is, in reality, a social activity. In the event that I am correct, then the original invariant for the file system  $FS_0$  is incorrect, and I do believe that to be the case. If I am incorrect with respect to the intentions of the original authors, then I have developed an interesting theory of file systems. Let us now return to the first abstract model and make the appropriate corrections.

### 3. The Modified File System Model

In order that the file system  $FS_1$  be a faithful reification of the first abstract model,  $FS_0$ , given by the domain equations

$$\varphi \in FS_0 = Fn \xrightarrow{m} FILE$$

$$\mu \in FILE = Pn \xrightarrow{m} PG$$

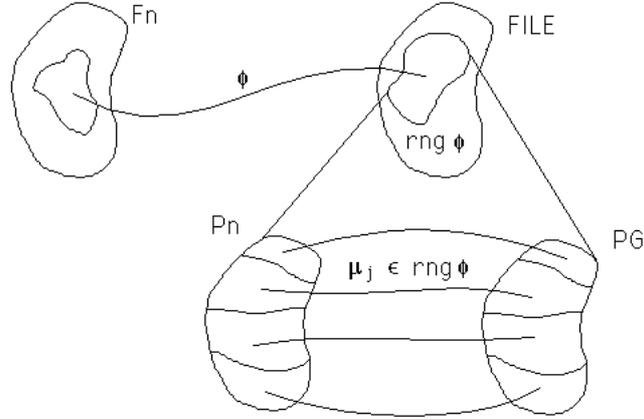
it is necessary to specify an invariant which states that *every page belongs exactly to one file*, i.e., that if we denote the set of all files in a file system,  $\varphi$ , by  $F = rng \varphi$ , then the set of all pages in  $\cup / \circ \mathcal{P} rng F$  is partitioned into the set

$$\{\varpi_j \mid \varpi_j = rng \mu_j \in \mathcal{P} rng F\}$$

Considering the inverse image,  $\varpi_j^{-1}$ , of a set of pages in the partition, then  $\cup / \circ \mathcal{P} dom F$  is partitioned into the set

$$\{\varpi_j^{-1} \circ \varpi_j \mid \varpi_j \in \mathcal{P} rng F\}$$

This invariant may be illustrated by the following diagram:



Formally, the invariant is given by

$$inv-FS_0: FS_0 \longrightarrow \mathbf{B}$$

$$inv-FS_0(\varphi) \triangleq$$

$$\Delta / \circ \mathcal{P} rng \circ rng \varphi = \cup / \circ \mathcal{P} rng \circ rng \varphi$$

$$\wedge |\oplus / \mathcal{P} j \circ \mathcal{P} rng \circ rng \varphi| = |\cup / \circ \mathcal{P} rng \circ rng \varphi|$$

$$\wedge \Delta / \circ \mathcal{P} dom \circ rng \varphi = \cup / \circ \mathcal{P} dom \circ rng \varphi$$

$$\wedge |\oplus / \mathcal{P} j \circ \mathcal{P} dom \circ rng \varphi| = |\cup / \circ \mathcal{P} dom \circ rng \varphi|$$

$$\wedge \wedge / \circ (dom - = (-)^{-1} \circ rng -)^* \circ \circ' / \circ \mathcal{P} j \circ rng \varphi$$

## THE METHOD OF THE IRISH SCHOOL

where the last clause expresses the fact that for every file  $\mu_j$  in the file system, the inverse image of its range,  $\mu_j^{-1} \circ \text{rng } \mu_j$ , is equal to its domain,  $\text{dom } \mu_j$ , i.e.,  $\forall \mu_j \in \text{rng } \varphi$ , let  $\varpi_j = \text{rng } \mu_j$  in  $\text{dom } \mu_j = \mu_j^{-1} \varpi_j$ .

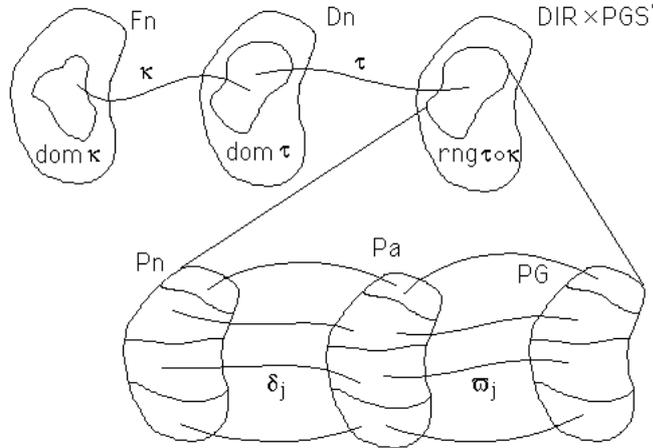
The invariant also relies on the *VDM* domain of the bag or multiset to be discussed in the next section. The injection operator,  $j$ , maps a set  $S$  into a singleton bag,  $[S \mapsto |S|]$ ; the operator  $\oplus/$  constructs a bag from a set of bags and then we take the size of the bag. But there is also an overloading of the injection operator in the invariant. The last clause uses the sequence injection operator,  $j: e \mapsto \langle e \rangle$  and  $\diamond$  denotes a unique concatenation operator. In this manner we are able to express the simple fact that the sum of the cardinalities of the sets in a partition of a set  $S$  is equal to the cardinality of the set  $S$ .

Now that we have settled the matter of the invariant for the most abstract model of the file system, we ought to take a close look at the operations at this level, with a particular view to ensure that they preserve the invariant. A full discussion of the details are given in Appendix C. Here, I wish to focus on the ‘put’ operation, which when given a file name,  $fn$ , a page name,  $pn$ , and a page  $pg$ , updates the file system  $\varphi$ .

### 3.1. The Put Command

Given a specification of an abstract model,  $MODEL_0$ , one of the salient features of the method of the Irish School is to construct a reification,  $MODEL_1$ , using both invariants and the retrieve function as guidelines, such that application of the operator calculus is tractable. In particular, when specifying the semantics of an operation on  $MODEL_0$ , it must be the image of the corresponding operation on  $MODEL_1$ , under the retrieve function. As we have already noted above, there is an important inter-relationship between an invariant and a retrieve function.

Having identified and resolved a basic problem in the published specification of the file system, the next major problem to be solved is to modify  $FS_1$ , such that it is a faithful reification of  $FS_0$ . Specifically, this entails reorganising the model such that the concept of partitioning, i.e., that every page must belong to only one file, is clearly exhibited and not buried within an invariant. I decided that the best way to accomplish this was to construct an isomorphic copy of  $FS_1$  which is illustrated by the diagram:



The domain equations of the new model are

$$\begin{aligned} \varphi \in FS'_1 &= CTLG_1 \times DIRS'_1 \\ \kappa \in CTLG_1 &= Fn \xrightarrow{m} Dn \\ \tau \in DIRS'_1 &= Dn \xrightarrow{m} (DIR_1 \times PGS'_1) \\ \delta \in DIR_1 &= Pn \xrightarrow{m} Pa \\ \varpi \in PGS'_1 &= Pa \xrightarrow{m} PG \end{aligned}$$

The conceptual model underlying  $FS'_1$  is based entirely on the notion of a pair of

## THE METHOD OF THE IRISH SCHOOL

composable maps,  $(f, g)$ , where  $\text{rng } f = \text{dom } g$ . As in the case of  $FS_1$ , I will of course ensure that the pair,  $(\kappa, \tau)$ , is composable and thence that  $\tau \circ \kappa$  is always well-defined:

$$\text{inv-}FS'_1(\kappa, \tau) \triangleq \text{rng } \kappa = \text{dom } \tau$$

In addition, for a given file system  $\varphi$ , the set  $\text{rng } \tau \circ \kappa$  consists of pairs of composable maps  $(\delta, \varpi)$ , which may be specified by

$$\text{inv-}FS'_1(\kappa, \tau) \triangleq \wedge / \circ ((\pi_1 - = \pi_2 -) \circ (\text{rng}, \text{dom}))^* \circ \diamond' / \circ \mathcal{P}J \circ \text{rng}(\tau \circ \kappa)$$

There is, of course, much more to the invariant than this; I must also now state that  $CTLG_1$ ,  $DIRS'_1$ , and  $DIR_1$  are injective, since I am using the general map domain notation. These and other details are in Appendix C.

The next step in the method is to verify that my new model is indeed isomorphic to  $FS_1$ . In particular, given file systems  $(c, ds, ps) \in FS_1$  and  $(\kappa, \tau) \in FS'_1$ , I must present a bijective function  $h: FS_1 \longrightarrow FS'_1$ , such that  $h: (c, ds, ps) \mapsto (\kappa, \tau)$ . Then, I must verify that there is a 1–1 correspondence for each operation in the model. Only the function  $h$  is presented in this Chapter. The other details are in Appendix C. To avoid confusion I have resorted to the use of roman letters for the original  $FS_1$ . In practice, the *discovery* of the bijective function  $h$  is the result of much drawing and sketching, usually in colour, of experimentation with concrete examples from the problem domain and application of various operators. Indeed, doing specifications often consists in such activity, which can not be adequately described and portrayed.

Since the function  $h$  is bijective, it should not matter whether one starts with  $FS_1$  or  $FS'_1$ . In practice, I found it easier to construct  $h: FS'_1 \longrightarrow FS_1$  first. The starting point is an example such as

$$\text{rng } ds = \{\delta_1, \delta_2, \dots, \delta_j, \dots, \delta_n\}$$

Then I need to restrict  $ps$  with respect to the range of each of the directories  $\delta_j$  to give

$$\{\text{rng } \delta_1 \triangleleft ps, \text{rng } \delta_2 \triangleleft ps, \dots, \text{rng } \delta_j \triangleleft ps, \dots, \text{rng } \delta_n \triangleleft ps\}$$

This is the actual form of  $ps$  in the original file system. In my new model, the structure corresponding to  $\text{rng } ds$  is

$$\text{rng } \tau = \{(\delta_1, \varpi_1), (\delta_2, \varpi_2), \dots, (\delta_j, \varpi_j), \dots, (\delta_n, \varpi_n)\}$$

Clearly, if I apply the projection function  $\pi_1$ , I obtain

$$\mathcal{P}\pi_1 \circ \text{rng } \tau = \text{rng } ds$$

Similarly, application of  $\pi_2$  will give the set of maps

$$\{\varpi_1, \varpi_2, \dots, \varpi_j, \dots, \varpi_n\}$$

and, thus, the original  $ps$  is obtained by

$$\cup / \circ \mathcal{P}\pi_2 \circ \text{rng } \tau = ps$$

I have nearly completed the task. What I have achieved is a pair of functions that map  $\text{rng } \tau$  to  $\text{rng } ds$  and  $ps$ . I must find the pair of functions that map  $\tau$  to  $ds$  and  $ps$ . For this purpose, I introduce another example

$$\tau = [dn_1 \mapsto (\delta_1, \varpi_1), dn_2 \mapsto (\delta_2, \varpi_2), \dots, dn_j \mapsto (\delta_j, \varpi_j), \dots, dn_n \mapsto (\delta_n, \varpi_n)]$$

I clearly need to use the map functor  $(- \xrightarrow{m} -)$ . To obtain  $ds$  from  $\tau$  is trivial:

$$ds = (- \xrightarrow{m} \pi_1)\tau$$

Applying  $(- \xrightarrow{m} \pi_2)$  and then taking the range leads to the other solution

$$ps = \cup / \circ \text{rng} \circ (- \xrightarrow{m} \pi_2)\tau$$

Consequently, the bijective function  $h: FS'_1 \longrightarrow FS_1$  consists of three distinct functions: the identity function which maps  $c$  to  $\kappa$ , the function  $(- \xrightarrow{m} \pi_1)$  which maps  $\tau$  to  $ds$ , and  $\cup / \circ \text{rng} \circ (- \xrightarrow{m} \pi_2)$  which maps  $\tau$  to  $ps$ :

$$\begin{aligned} h: FS'_1 &\longrightarrow FS_1 \\ h(\kappa, \tau) &\triangleq (\mathcal{I}(\kappa), (- \xrightarrow{m} \pi_1)(\tau), \cup / \circ \text{rng} \circ (- \xrightarrow{m} \pi_2)(\tau)) \end{aligned}$$

Clearly, the inverse function  $h^{-1}: FS_1 \longrightarrow FS'_1$  will also consist of the identity function  $\mathcal{I}: \kappa \mapsto c$ . To obtain the rest, note that if

$$ds = [dn_1 \mapsto \delta_1, dn_2 \mapsto \delta_2, \dots, dn_j \mapsto (\delta_j, \varpi_j), \dots, dn_n \mapsto \delta_n]$$

then we may apply a pair of functions to obtain

$$\begin{aligned} (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} \text{rng}) ds &= [dn_1 \mapsto \text{rng } \delta_1 \triangleleft ps, dn_2 \mapsto \text{rng } \delta_2 \triangleleft ps, \\ &\dots, dn_j \mapsto \text{rng } \delta_j \triangleleft ps, \dots, dn_n \mapsto \text{rng } \delta_n \triangleleft ps] \end{aligned}$$

Thus, all that I have to do is to find an expression that combines maps such as  $[\dots, a_j \mapsto b_j, \dots]$  and  $[\dots, a_j \mapsto c_j, \dots]$  to give  $[\dots, a_j \mapsto (b_j, c_j), \dots]$ . For this

## THE METHOD OF THE IRISH SCHOOL

express purpose, I use the operator symbol,  $\bowtie$ , where for a pair of maps  $\mu_1$  and  $\mu_2$  in  $X \xrightarrow{m} Y$ , such that  $\text{dom } \mu_1 = \text{dom } \mu_2$ , then  $(-\xrightarrow{m} \pi_1)(\mu_1 \bowtie \mu_2) = \mu_1$  and  $(-\xrightarrow{m} \pi_2)(\mu_1 \bowtie \mu_2) = \mu_2$ . Consequently, the required inverse function  $h^{-1}$  is given by

$$\begin{aligned} h^{-1}: FS_1 &\longrightarrow FS'_1 \\ h^{-1}(c, ds, ps) &\triangleq (\mathcal{I}(c), ds \bowtie (-\xrightarrow{m} \triangleleft ps) \circ (-\xrightarrow{m} \text{rng})ds) \end{aligned}$$

which basically concludes the constructive proof. Now we are ready to consider the put operation. The abstract syntax is quite simply

$$Put_0 = Put'_1 = Put_1 \quad :: \quad Fn \times Pn \times PG$$

where I have indicated that the same syntax is applicable to all three models. The semantic functions for each model are as follows:

$$\begin{aligned} Put_0: Fn \times Pn \times PG &\longrightarrow FS_0 \longrightarrow FS_0 \\ Put_0 \llbracket fn, pn, pg \rrbracket \varphi &\triangleq \\ \chi \llbracket fn \rrbracket \varphi \wedge \chi \llbracket pn \rrbracket \varphi(fn) & \\ \rightarrow \varphi + [fn \mapsto \varphi(fn) + [pn \mapsto pg]] & \\ \chi \llbracket fn \rrbracket \varphi \wedge \neg \chi \llbracket pn \rrbracket \varphi(fn) & \\ \rightarrow \varphi + [fn \mapsto \varphi(fn) \cup [pn \mapsto pg]] & \end{aligned}$$

with the additional pre-condition

$$\text{pre-}Put_0 \llbracket fn, pn, pg \rrbracket \varphi \triangleq pg \notin \cup / \circ \mathcal{P} \text{rng}(\{\varphi(fn)\}) \triangleleft \text{rng } \varphi$$

which is necessary to satisfy the invariant that every page shall belong to exactly one file. I deal with the model  $FS_1$  next:

$$\begin{aligned} Put_1: Fn \times Pn \times PG &\longrightarrow FS_1 \longrightarrow FS_1 \\ Put_1 \llbracket fn, pn, pg \rrbracket (c, ds, ps) &\triangleq \\ \chi \llbracket fn \rrbracket c \wedge \chi \llbracket pn \rrbracket ds \circ c(fn) & \\ \rightarrow (c, ds, ps + [(ds \circ c(fn))pn \mapsto pg]) & \\ \chi \llbracket fn \rrbracket c \wedge \neg \chi \llbracket pn \rrbracket ds \circ c(fn) & \\ \rightarrow \text{let } pa \in Pa \setminus \text{dom } ps \text{ in} & \\ (c, ds + [c(fn) \mapsto ds \circ c(fn) \cup [pn \mapsto pa]], ps \cup [pa \mapsto pg]) & \end{aligned}$$

Once more I need an extra constraint on  $pg$  to guarantee that the invariant still holds, i.e., that “every page, understood as page-address, is described in exactly one

directory (that is belongs to exactly one file)". The form of the pre-condition will be similar to that for the operation  $Put_0$ :

$$\begin{aligned} pre\text{-}Put_1 \llbracket fn, pn, pg \rrbracket (c, ds, ps) &\triangleq \\ pg \notin rng((\cup / \circ \mathcal{P} \text{rng}(\{ds \circ c(fn)\} \Leftarrow rng \ ds \circ c)) \triangleleft ps) & \end{aligned}$$

It is constructed in the following manner. Since  $rng \ ds \circ c$  is the set of all directories in the file system, and  $ds \circ c(fn)$  is the directory affected by the  $Put_1$  operation, then we must ensure that  $pg$  is not at the end of any arrows emanating from the set of directories  $\{ds \circ c(fn)\} \Leftarrow rng \ ds \circ c$ . To specify this, I first identify the set of all page addresses in use by these directories,  $\cup / \circ \mathcal{P} \text{rng}(\{ds \circ c(fn)\} \Leftarrow rng \ ds \circ c)$ , and restrict the  $ps$  map accordingly. Then the given page,  $pg$ , must not occur in the range of this map. Finally, for the new model  $FS'_1$ , we have

$$\begin{aligned} Put'_1: Fn \times Pn \times PG &\longrightarrow FS'_1 \longrightarrow FS'_1 \\ Put'_1 \llbracket fn, pn, pg \rrbracket (\kappa, \tau) &\triangleq \\ \chi \llbracket fn \rrbracket \kappa \wedge \chi \llbracket pn \rrbracket \pi_1(\tau \circ \kappa(fn)) & \\ \rightarrow \text{let } (\delta, \varpi) = \tau \circ \kappa(fn) \text{ in} & \\ (\kappa, \tau + [\kappa(fn) \mapsto (\delta, \varpi + [\delta(pn) \mapsto pg])]) & \\ \chi \llbracket fn \rrbracket \kappa \wedge \neg \chi \llbracket pn \rrbracket \pi_1(\tau \circ \kappa(fn)) & \\ \rightarrow \text{let } (\delta, \varpi) = \tau \circ \kappa(fn) \text{ in} & \\ \text{let } pa \in Pa \setminus dom \cup / \circ \mathcal{P} \pi_2 \circ rng \ \tau \circ \kappa \text{ in} & \\ (\kappa, \tau + [\kappa(fn) \mapsto (\delta \cup [pn \mapsto pa], \varpi \cup [pa \mapsto pg])]) & \end{aligned}$$

and we also need the pre-condition

$$\begin{aligned} pre\text{-}Put'_1 \llbracket fn, pn, pg \rrbracket (\kappa, \tau) &\triangleq \\ pg \notin rng(\cup / \circ \mathcal{P} \pi_2(\{\tau \circ \kappa(fn)\} \Leftarrow rng \ \tau \circ \kappa)) & \end{aligned}$$

Specifications for each of the other operations are constructed in a similar fashion. What is striking is the apparent complexity of the expressions, though they are no more complex than comparable expressions in, say, the differential or integral calculus. The next task is to conduct proofs.

# THE METHOD OF THE IRISH SCHOOL

## 3.2. Theorem and Proof

When we examine the inter-relationships between the three models in question, and look closely at the nature of the ‘put’ operation, we will notice that the relevant facts may be summarised by a set of commuting diagrams grouped in the following manner

$$\begin{array}{ccccc}
 FS_0 & \xrightarrow{Lkp_0[[fn]]} & FILE & \xrightarrow{Upd_0[[pn, pg]]} & FILE \\
 \uparrow \mathfrak{R}'_{10} & & \uparrow \mathfrak{R}_1 & & \uparrow \mathfrak{R}_1 \\
 CTLG \times DIRS & \xrightarrow{Lkp'_1[[fn]]} & DIR \times PGS' & \xrightarrow{Upd'_1[[pn, pg]]} & DIR \times PGS' \\
 \uparrow \mathfrak{R}'_{11} & & \uparrow \mathfrak{R}_2 & & \uparrow \mathfrak{R}_2 \\
 CTLG \times DIRS \times PGS & \xrightarrow{Lkp_1[[fn]]} & DIR \times PGS & \xrightarrow{Upd_1[[pn, pg]]} & DIR \times PGS
 \end{array}$$

where I have indicated that the put operation essentially consists of a lookup with respect to a file name, followed by an update with respect to page name and page. Before considering each of the commuting squares in turn, I present one of the major results of the research.

The retrieve function  $\mathfrak{R}'_{11}$  denotes the bijective mapping that gives the isomorphism between my new model  $FS'_1$  and the original reification  $FS_1$ . From Appendix C, the retrieve function  $\mathfrak{R}'_{10}$  is given by

$$\begin{aligned}
 \mathfrak{R}'_{10}: FS'_1 &\longrightarrow FS_0 \\
 \mathfrak{R}'_{10}(\kappa, \tau) &\triangleq (- \xrightarrow{m} \circ)(\tau \circ \kappa)
 \end{aligned}$$

Thus, the problem of finding an operator form for the retrieve function,  $\mathfrak{R}_{10}$ , that retrieves  $FS_0$  from  $FS_1$ , and which was a major concern identified earlier, is given by  $\mathfrak{R}'_{10} \circ \mathfrak{R}'_{11}$ , where

$$\mathfrak{R}'_{10} \circ \mathfrak{R}'_{11}(c, ds, ps) = (- \xrightarrow{m} \circ)((ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} \text{rng})ds) \circ c)$$

This is one of the primary theorems of the File System case study, considered as a theory. Now we may turn our attention to each of the four squares in the commuting diagram given. Viewing them analogously to the four quadrants of the cartesian plane, I will number them 1, 2, 3, 4, in counter-clockwise order. Let us look at

square 1. For convenience, I consider only the case:  $fn \in \text{dom } \varphi$  and  $pn \in \text{dom } \varphi(fn)$ :

$$\begin{array}{ccc} \varphi(fn) = \mu & \xrightarrow{Upd_0[[pn, pg]]} & \mu + [pn \mapsto pg] \\ \uparrow \mathfrak{R}_1 & & \uparrow \mathfrak{R}_1 \\ \tau \circ \kappa(fn) = (\delta, \varpi) & \xrightarrow{Upd'_1[[pn, pg]]} & (\delta, \varpi + [\delta(pn) \mapsto pg]) \end{array}$$

Formally, I wish to establish the result that

$$\mathfrak{R}_1 \circ Upd'_1[[pn, pg]] = Upd_0[[pn, pg]] \circ \mathfrak{R}_1$$

where  $\mathfrak{R}_1: (\delta, \varpi) \mapsto \varpi \circ \delta$ . Looking at the left-hand-side first, we immediately have

$$\begin{aligned} (\mathfrak{R}_1 \circ Upd'_1[[pn, pg]])(\delta, \varpi) &= \mathfrak{R}_1(\delta, \varpi + [\delta(pn) \mapsto pg]) \\ &= (\varpi + [\delta(pn) \mapsto pg]) \circ \delta \end{aligned}$$

and, similarly, for the right-hand-side

$$\begin{aligned} (Upd_0[[pn, pg]] \circ \mathfrak{R}_1)(\delta, \varpi) &= Upd_0[[pn, pg]](\varpi \circ \delta) \\ &= (\varpi \circ \delta) + [pn \mapsto pg] \end{aligned}$$

I prefer to present the ‘problem to solve’ as a conventional hypothesis/conclusion pair:

HYPOTHESIS 5.1.  $\mathfrak{R}_1: (\delta, \varpi) \mapsto \varpi \circ \delta$ .

CONCLUSION 5.1.  $(\varpi + [\delta(pn) \mapsto pg]) \circ \delta = (\varpi \circ \delta) + [pn \mapsto pg]$ .

*Proof:* This is a simple application of the operator calculus:

$$\begin{aligned} (\varpi + [\delta(pn) \mapsto pg]) \circ \delta &= (\varpi + [\delta(pn) \mapsto pg]) \circ (\delta + [pn \mapsto \delta(pn)]) \\ &= (\varpi \circ \delta) + ([\delta(pn) \mapsto pg] \circ [pn \mapsto \delta(pn)]) \\ &= (\varpi \circ \delta) + [pn \mapsto pg] \end{aligned}$$

subject to certain constraints to be mentioned. It was precisely by studying the conditions under which application of the operator calculus was valid that much of the theory of Chapters 2 and 3 was developed. I might simply present the constraints for the particular case in question. However, much more is to be learned by considering the general case which concerns a homomorphism of a monoid of composable maps:

$$\begin{array}{ccc} FILE^2 & \xrightarrow{+} & FILE^2 \\ \uparrow \mathfrak{R}_1^2 & & \uparrow \mathfrak{R}_1^2 \\ (DIR \times PGS)^2 & \xrightarrow{+} & (DIR \times PGS)^2 \end{array}$$

and we want to examine the validity of the lemma:

## THE METHOD OF THE IRISH SCHOOL

LEMMA 5.1.  $(\varpi_1 + \varpi_2) \circ (\delta_1 + \delta_2) = (\varpi_1 \circ \delta_1) + (\varpi_2 \circ \delta_2)$ .

First, let us check that if  $(\delta_1, \varpi_1)$  and  $(\delta_2, \varpi_2)$  are pairs of composable maps, then  $(\delta_1 + \delta_2, \varpi_1 + \varpi_2)$  is composable. In other words, let us determine if  $DIR \times PGS$  is indeed a monoid under the override operation. By definition,  $rng \delta_1 = dom \varpi_1$  and  $rng \delta_2 = dom \varpi_2$ . Since  $dom$  is a monoid homomorphism of  $(PGS, +, \theta)$ , then  $dom(\varpi_1 + \varpi_2) = dom \varpi_1 \cup dom \varpi_2$ . But, in general, for any pairs of maps  $\mu_1, \mu_2 \in X \xrightarrow{m} Y$ , we have  $rng(\mu_1 + \mu_2) \subseteq rng \mu_1 \cup rng \mu_2$ . We have equality *only* in the case that  $X \xrightarrow{m} Y$  denotes 1–1 maps. But this is precisely the case for the *DIR* domain and the condition is part of the invariant. Although the condition is necessary, it is not sufficient. Let us exhibit a simple counter-example. For the pair of injective maps

$$\begin{aligned}\delta_1 &= [pn_1 \mapsto pa_1, pn_2 \mapsto pa_2] \\ \delta_2 &= [pn_1 \mapsto pa_2]\end{aligned}$$

we have  $\delta_1 + \delta_2 = [pn_1 \mapsto pa_2, pn_2 \mapsto pa_2]$ , which is not injective! Thus, in general, the composition of a pair of injective maps under the override operation is not injective, and consequently,  $(DIR, +, \theta)$  is not a monoid, if we insist on the property of 1–1 map. We can construct an appropriate monoid if we restrict the definition of the override operator, denoted  $\oplus$ , such that  $rng(\delta_1 \oplus \delta_2) = rng \delta_1 \cup rng \delta_2$ , i.e., use the result we want to obtain, to give the monoidal structure  $(DIR, \oplus, \theta)$ . In the file system, it is precisely the put operation that builds up this structure. Thus, we have discovered another significant result: that the *Meta-IV* override operator is too weak for the domain *DIR*. In fact, we have done this sort of thing before when we considered the symmetric difference operator for maps to be the appropriate setting in which to study the *Meta-IV* extend operator and we shall see the same sort of development when we look at the bag or multiset.

Returning to the lemma in question, we have determined the conditions of validity for the use of the override operator for the *DIR* domain, and we now have the result

$$\begin{aligned}rng(\delta_1 + \delta_2) &= rng \delta_1 \cup rng \delta_2 \\ &= dom \varpi_1 \cup dom \varpi_2 \\ &= dom(\varpi_1 + \varpi_2)\end{aligned}$$

Therefore,  $\delta_1 + \delta_2$  and  $\varpi_1 + \varpi_2$  are composable. The operator calculus may now

be employed to establish the lemma

$$\begin{aligned}
 (\varpi_1 + \varpi_2) \circ (\delta_1 + \delta_2) &= ((\text{dom } \varpi_2 \triangleleft \varpi_1) \cup \varpi_2) \circ ((\text{dom } \delta_2 \triangleleft \delta_1) \cup \delta_2) \\
 &= ((\text{dom } \varpi_2 \triangleleft \varpi_1) \circ (\text{dom } \delta_2 \triangleleft \delta_1)) \cup (\varpi_2 \circ \delta_2) \\
 &= (\text{dom } \delta_2 \triangleleft (\varpi_1 \circ \delta_1)) \cup (\varpi_2 \circ \delta_2) \\
 &= (\text{dom}(\varpi_2 \circ \delta_2) \triangleleft (\varpi_1 \circ \delta_1)) \cup (\varpi_2 \circ \delta_2) \\
 &= (\varpi_1 \circ \delta_1) + (\varpi_2 \circ \delta_2)
 \end{aligned}$$

and that basically completes the proof for square 1. Technically, we ought to repeat the proof for the case:  $fn \in \text{dom } \varphi$  and  $pn \notin \text{dom } \varphi(fn)$ . But the details are similar. Let us now look at square 2:

$$\begin{array}{ccc}
 \varphi & \xrightarrow{Lkp_0 \llbracket fn \rrbracket} & \varphi(fn) = \mu \\
 \uparrow \mathfrak{R}'_{10} & & \uparrow \mathfrak{R}_1 \\
 (\kappa, \tau) & \xrightarrow{Lkp'_1 \llbracket fn \rrbracket} & \tau \circ \kappa(fn) = (\delta, \varpi)
 \end{array}$$

Here, we want to establish the result

$$\mathfrak{R}_1 \circ Lkp'_1 \llbracket fn \rrbracket = Lkp_0 \llbracket fn \rrbracket \circ \mathfrak{R}'_{10}$$

From the left-hand-side we obtain the result that  $(\mathfrak{R}_1 \circ Lkp'_1 \llbracket fn \rrbracket)(\kappa, \tau) = \mathfrak{R}_1(\tau \circ \kappa(fn))$  and it would appear that we can proceed no further. The right-hand-side gives

$$\begin{aligned}
 (Lkp_0 \llbracket fn \rrbracket \circ \mathfrak{R}'_{10})(\kappa, \tau) &= Lkp_0 \llbracket fn \rrbracket (- \xrightarrow{m} \circ)(\tau \circ \kappa) \\
 &= ((- \xrightarrow{m} \circ)(\tau \circ \kappa))(fn)
 \end{aligned}$$

which gives us the equality

$$\mathfrak{R}_1(\tau \circ \kappa(fn)) = ((- \xrightarrow{m} \circ)(\tau \circ \kappa))(fn)$$

which is of fundamental significance. Words are almost superfluous. In considering the particular file  $\mu$  identified by the expression  $\tau \circ \kappa(fn) = (\delta, \varpi)$ , and, of course,  $\mathfrak{R}_1(\delta, \varpi) = \varpi \circ \delta = \mu$ , we may obtain the same result by first composing all pairs  $(\delta, \varpi)$  in  $\tau \circ \kappa$  to give  $\varphi$ , and then using  $fn$  to pick out the required file  $\varphi(fn) = \mu$ . For a long time I had supposed that in the original model,  $(c, ds, ps) \in FS_1$ , the composition  $ds \circ c$  corresponded to  $\varphi$ , which is entirely wrong. There is an ‘inner composition’ as well as the more obvious ‘outer’ composition. We ought now to look at the squares in the lower quadrants, squares 3 and 4, respectively. The goal here is to verify that the put operation preserves the bijective mapping between  $FS'_1$  and

## THE METHOD OF THE IRISH SCHOOL

$FS_1$  and advance the argument that the two models are isomorphic. For square 4 we have

$$\begin{array}{ccc} \tau \circ \kappa(fn) = (\delta, \varpi) & \xrightarrow{Upd'_1[[pn, pg]]} & (\delta, \varpi + [\delta(pn) \mapsto pg]) \\ \uparrow \mathfrak{R}'_{11} & & \uparrow \mathfrak{R}_2 \\ (ds \circ c, ps) = (\delta, ps) & \xrightarrow{Upd_1[[pn, pg]]} & (\delta, ps + [\delta(pn) \mapsto pg]) \end{array}$$

where  $\mathfrak{R}_2(\delta, ps) = (\delta, \text{rng } \delta \triangleleft ps)$ . It may be shown that the ‘problem to solve’ may be written in the form

**HYPOTHESIS 5.2.**  $\mathfrak{R}_2: (\delta, ps) \mapsto (\delta, \text{rng } \delta \triangleleft ps)$ .

**CONCLUSION 5.2.**  $\text{rng } \delta \triangleleft (ps + [\delta(pn) \mapsto pg]) = (\text{rng } \delta \triangleleft ps) + [\delta(pn) \mapsto pg]$ .

*Proof:* Since  $\triangleleft_S$  is an endomorphism of  $(X \xrightarrow{m} Y, +, \theta)$ , i.e.,

$$\triangleleft_S (\mu_1 + \mu_2) = \triangleleft_S \mu_1 + \triangleleft_S \mu_2$$

then the proof is immediate

$$\begin{aligned} \text{rng } \delta \triangleleft (ps + [\delta(pn) \mapsto pg]) &= (\text{rng } \delta \triangleleft ps) + (\text{rng } \delta \triangleleft [\delta(pn) \mapsto pg]) \\ &= (\text{rng } \delta \triangleleft ps) + [\delta(pn) \mapsto pg] \end{aligned}$$

This is just a particular instance of the lemma

**LEMMA 5.2.**  $\text{rng } \delta \triangleleft (ps_1 + ps_2) = (\text{rng } \delta \triangleleft ps_1) + (\text{rng } \delta \triangleleft ps_2)$ .

The final part of the proof concerns square 3:

$$\begin{array}{ccc} (\kappa, \tau) & \xrightarrow{Lkp'_1[[fn]]} & \tau \circ \kappa(fn) = (\delta, \varpi) \\ \uparrow \mathfrak{R}'_{11} & & \uparrow \mathfrak{R}_2 \\ (c, ds, ps) & \xrightarrow{Lkp_1[[fn]]} & (ds \circ c, ps) = (\delta, ps) \end{array}$$

where  $\mathfrak{R}'_{11}(c, ds, ps) = (c, ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} \text{rng } ds))$ . We have to show that

$$\mathfrak{R}_2 \circ Lkp_1[[fn]] = Lkp'_1[[fn]] \circ \mathfrak{R}'_{11}$$

First, the left-hand-side gives us

$$\begin{aligned} (\mathfrak{R}_2 \circ Lkp_1[[fn]])(c, ds, ps) &= \mathfrak{R}_2(ds \circ c(fn), ps) \\ &= (ds \circ c(fn), \text{rng } ds \circ c(fn) \triangleleft ps) \end{aligned}$$

From the right-hand-side we obtain

$$\begin{aligned} (Lkp'_1 \llbracket fn \rrbracket \circ \mathfrak{R}'_{11})(c, ds, ps) &= Lkp'_1 \llbracket fn \rrbracket (c, ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} rng) ds) \\ &= (ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} rng) ds) \circ c(fn) \end{aligned}$$

Therefore, we have to ‘prove’ that the following equality holds

$$(ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} rng) ds) \circ c(fn) = (ds \circ c(fn), rng ds \circ c(fn) \triangleleft ps)$$

I enclose ‘prove’ in quotes precisely because it has already been proven by construction of the bijective mapping. If we regard  $(ds \circ c(fn), rng ds \circ c(fn) \triangleleft ps)$  as the range of a singleton map in  $DIRS'$ , then we may formally write

$$\begin{aligned} \tau &= [dn \mapsto (ds \circ c(fn), rng ds \circ c(fn) \triangleleft ps)] \\ &= [dn \mapsto ds \circ c(fn)] \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} rng) [dn \mapsto ds \circ c(fn)] \end{aligned}$$

and we have a result similar to that which we obtained for square 2. In this case, we are using a file name to identify a pair  $(\delta = ds \circ c(fn), \varpi = rng ds \circ c(fn) \triangleleft ps)$ . Thus, we have proved that, with respect to the put operation, the new model,  $FS'_1$  is isomorphic to the original  $FS_1$ , that  $FS'_1$  is a faithful reification of  $FS_0$ , and consequently that  $FS_1$  is also a faithful reification of  $FS_0$ . Similar results for the other operations are presented in Appendix C.

In the process of carrying out the proofs, the *real* structure of both  $FS_0$  and  $FS_1$  has been exposed and certain inadequacies and defects identified and remedied. Probably the most interesting result of the effort is the exposition of the sort of real algebra that underlies such specifications. We will now turn to a different application domain—a bill of material, a specification of which has also appeared in the published *VDM* literature. It is also noteworthy that the development to be presented grew out of a deep dissatisfaction with another invariant. The problem domain involves both the domain of bags and the domain of relations which are basic to the Irish School.

# THE METHOD OF THE IRISH SCHOOL

## 4. Domain of Bags

There is a certain æsthetics in good specifications which is reflected in the very simplicity of form. In practice, specifications in the *VDM Meta-IV* tend to appear complex and verbose, the very verbosity obscuring an inner simplicity of structure that must be brought out. Part of the method of the Irish School is an insistence on seeking out such inner simplicity, with due consideration of the application of Ockham's Razor. True to the spirit of Lakatos, that the method of discovery is as important as the result discovered, I begin this section with a simple problem that I addressed—the formal specification of the board game SCRABBLE (which is a registered trademark of Spears Games).

I have found that the game of SCRABBLE provides a natural setting for a conceptual model of a bag. SCRABBLE is a board game for 2–4 players, each of which has a rack containing at most 7 tiles. A tile is either blank (which may be used to represent any letter, has the value 0 and lexicographically is the last 'letter' in the alphabet) or represents a letter and its corresponding predetermined value. A player makes a move by choosing tiles from the rack and placing them judiciously on the board to form one or more recognised words, computing the appropriate score according to a set of rules and replenishing the rack by selecting from a bag the number of tiles placed.

Both the bag of tiles and the rack are, in fact, bags in the mathematical sense. The bag or multiset is a 'primitive' domain of  $\mathcal{Z}$  (Hayes 1987, 16) but not of the *VDM*. Now what exactly do I mean when I say that the bag is a primitive domain of  $\mathcal{Z}$  and not a primitive domain of the *VDM*? Just this, that in  $\mathcal{Z}$ , there is a special bracketting notation to denote an explicit bag. There is no such counterpart in the *VDM*. But I have spoken of the *VDM* as if it were some fixed corpus of domains, operations, and notation. However, as I have already noted previously, the originators of the *VDM* did not foresee that it should be so. One is free to evolve new notations as one feels the need—in the spirit of mathematics. For after all, the *VDM Meta-IV* is but a distinguished body of discrete mathematics.

There is strictly speaking no need to use a special bracketting notation to denote

an explicit bag. The bag may be modelled in the *VDM Meta-IV* by a map:

$$BAG = \Sigma \xrightarrow[m]{} \mathbf{N}_1$$

where  $\Sigma$  is the ‘domain of interest’. Just as a map or finite function forms a special class of sets of ordered pairs, so a bag is a special class of map where the range or codomain is the set of positive natural numbers. One recalls that the sequence is another special class of map where the domain ranges over positive natural numbers. It is precisely this similarity between sequences and bags that might suggest the need for special distinguishing notation for the latter. Here I must mention the remark by Knuth that although a bag is formally equivalent to a map, this formal equivalence is of “little or no practical value for creative reasoning” (1981, 2:636). I would vigorously dispute that. For certain kinds of mathematician, it might be so. However, for those accustomed to constructive mathematics, such as is the case in formal specifications, the formal equivalence is adequate. As will become evident later, I have chosen to focus attention on the special bag operators, rather than being concerned about particular delimiters. After all, in the case of the latter, there seems to be already a proliferation of special delimiter forms in the *Meta-IV*.

There is always the possibility of ‘canonizing’ the word ‘bag’ so that it becomes a domain constructor. For instance, given domains  $X$  and  $Y$ , say, then  $BAG(X)$  and  $BAG(Y)$  denote new domains of bags over  $X$  and  $Y$ , respectively. We may reinforce this notation by defining

$$BAG(\Sigma) \triangleq \Sigma \xrightarrow[m]{} \mathbf{N}_1$$

Such an emphasis on the name moves us in the direction of language and away from abstract form. That is to say we would tend to focus on  $BAG(\Sigma)$ , rather than  $\Sigma \xrightarrow[m]{} \mathbf{N}_1$ . To use  $BAG(\Sigma)$  is to recognize explicitly in our specification that it is indeed a bag that we have in mind. On the other hand, as will be demonstrated, one may use the abstract form  $\Sigma \xrightarrow[m]{} \mathbf{N}_1$  in specifications and perhaps not realize that it may be a bag.

Just because a bag is a map, then one employing the abstract form in a specification tends to use only the primitive set of map operators and thereby misses the opportunity of employing more expressive notation, one that is peculiar to bags. The material in the remainder of this section shows the evolution of such expressive

## THE METHOD OF THE IRISH SCHOOL

notation and demonstrates its effectiveness by drawing on sample applications in the literature.

### 4.1. Operations on Bags

Operations on a bag are immediately inherited from those of the underlying map domain. Consider, for example, the operation of adding an element to a bag:

$$add: \Sigma \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1)$$

$$\begin{aligned} add[[e]]\beta &\triangleq \\ \chi[[e]]\beta &\rightarrow \beta + [e \mapsto \beta(e) + 1] \\ &\rightarrow \beta \cup [e \mapsto 1] \end{aligned}$$

Clearly, this operation of addition encompasses both map override and map extend. Interestingly, map override is non-commutative; but map extend is commutative. I will demonstrate that from this definition a natural commutative operation on bags may be constructed, that of the addition of two bags. Now, let us apply the process of generalisation:

$$add: (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1)$$

$$\begin{aligned} add[[\theta]]\beta &\triangleq \beta \\ add[[e \mapsto n] \cup \beta_1]\beta_2 &\triangleq \\ \chi[[e]]\beta_2 &\rightarrow add[[\beta_1]](\beta_2 + [e \mapsto \beta_2(e) + n]) \\ &\rightarrow add[[\beta_1]](\beta_2 \cup [e \mapsto n]) \end{aligned}$$

Note that I rely on the definition of the map extend operator to express a non-empty map as the extension of two maps, one of which is the singleton map. We now have a definition for the addition of two bags, the properties of which are subsequently exhibited.

#### 4.1.1. Addition of Bags

Denoting addition by  $\oplus$  one may express the above in the form

$$\begin{aligned} \theta \oplus \beta &= \beta \\ ([e \mapsto n] \cup \beta_1) \oplus \beta_2 &= \\ \chi[[e]]\beta_2 &\rightarrow \beta_1 \oplus (\beta_2 + [e \mapsto \beta_2(e) + n]) \\ &\rightarrow \beta_1 \oplus (\beta_2 \cup [e \mapsto n]) \end{aligned}$$

I have chosen to use  $\oplus$  to denote addition rather than  $+$ , since the latter is already used to denote map override and I wish to avoid confusion and ambiguity at this

definitional stage. Knuth uses the operator symbol  $\uplus$  and remarks that “it would not be as desirable to use ‘ $A + B$ ’ for [addition of bags  $A$  and  $B$ ], since algebraists have found that  $A + B$  is a good notation for  $\{\alpha + \beta \mid \alpha \in A \text{ and } \beta \in B\}$ ” where  $A$  and  $B$  are bags of nonnegative integers (Knuth 1981, 2:636).

If we choose to rely on the basic map operations only, then adding an element to a bag is a two-line specification guarded by a boolean condition. On the other hand, with the new definition of the addition of bags, adding an element becomes a simple operation:

$$\begin{aligned} \text{add}: \Sigma &\longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1) \\ \text{add}[e]\beta &\triangleq [e \mapsto 1] \oplus \beta \end{aligned}$$

What is more, the properties of  $\oplus$  are inherited from the properties of addition of natural numbers. Addition of bags is commutative:  $\beta_1 \oplus \beta_2 = \beta_2 \oplus \beta_1$ , and associative:  $\beta_1 \oplus (\beta_2 \oplus \beta_3) = (\beta_1 \oplus \beta_2) \oplus \beta_3$ . Such properties may be laboriously demonstrated by rigorous application of map operators. However, a simple reflection on the properties of a bag in SCRABBLE and what one does with a fisful of tiles, is convincing enough evidence to proceed without caring to be sidetracked. We may sum all of this up by simply saying that a bag domain with binary operation  $\oplus$  is a commutative monoid isomorphic to the monoid of positive natural numbers under addition.

Since multiplication of natural numbers may be defined recursively in terms of addition, then we also have multiplication of bags by natural numbers:

$$m \otimes ([e \mapsto n] \cup \beta) = [e \mapsto m \times n] \cup (m \otimes \beta)$$

where I use  $\otimes$  for reasons similar to the use of  $\oplus$ . Such multiplication obeys the usual distributive law:  $n \otimes (\beta_1 \oplus \beta_2) = (n \otimes \beta_1) \oplus (n \otimes \beta_2)$ . Again this can be summed up by saying that a bag domain with binary operators  $\oplus$  and  $\otimes$  is a commutative semi-ring isomorphic to the semi-ring of natural numbers under addition and multiplication.

## THE METHOD OF THE IRISH SCHOOL

### 4.1.2. The Size of a Bag

Let us consider the computation of the number of elements in a bag. The size of a bag may be given by:

$$\text{size}: (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow \mathbf{N}$$

$$\text{size}(\theta) \triangleq 0$$

$$\text{size}([e \mapsto n] \cup \beta) \triangleq n + \text{size}(\beta)$$

Just as we may replace the word ‘add’ with the abstract symbol  $\oplus$ , so we may choose either  $\#\beta$  or  $|\beta|$  to denote the size of the bag  $\beta$ . Some obvious properties are  $|\beta_1 \oplus \beta_2| = |\beta_1| + |\beta_2|$  and  $|n \otimes \beta| = n \times |\beta|$ .

### 4.1.3. The Transfer Function *items*

There is an obvious generalisation of the *elems* operator on sequences. Recall that *elems* maps a sequence of elements into the set of elements in the sequence without repetition. Should we wish to retain information on the number of occurrences of an element in a sequence then the *items* operator, which maps a sequence of elements into the bag of elements in the sequence, is required:

$$\text{items}: \Sigma^* \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1)$$

$$\text{items} \Lambda \triangleq \theta$$

$$\text{items}(\langle e \rangle \wedge \tau) \triangleq [e \mapsto 1] \oplus \text{items} \tau$$

Let  $\sigma$  denote a sequence. Then the relationship between *elems* and *items* may be expressed in the form:  $\text{dom} \circ \text{items} \sigma = \text{elems} \sigma$ . The transfer operator *items* satisfies other relations. For example, the length of a sequence  $\sigma$  may be expressed in terms of the *size* operator of bags:  $\text{len} \sigma = |\text{items} \sigma|$ .

### 4.1.4. Union and Intersection

Since a bag is a generalisation of a set, thus the name multiset, one may define union,  $\cup$ , and intersection,  $\cap$ , operators, using maximum and minimum operators, respectively (Knuth 1981, 2:464; Manna and Waldinger 1985, 1:522-3). The VDM definitions are exactly analogous to those of bag addition given above. For union, one has

$$\text{union}: (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1)$$

$$\text{union}[\theta]\beta \triangleq \beta$$

$$\text{union}[[e \mapsto n] \cup \beta_1]\beta_2 \triangleq$$

$$\chi[[e]\beta_2 \rightarrow \text{union}[\beta_1](\beta_2 + [e \mapsto \beta_2(e) \uparrow n])$$

$$\rightarrow \text{union}[\beta_1](\beta_2 \cup [e \mapsto n])$$

and for intersection:

$$\text{intersect}: (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1)$$

$$\text{intersect}[\theta]\beta \triangleq \beta$$

$$\text{intersect}[[e \mapsto n] \cup \beta_1]\beta_2 \triangleq$$

$$\chi[[e]\beta_2 \rightarrow \text{intersect}[\beta_1](\beta_2 + [e \mapsto \beta_2(e) \downarrow n])$$

$$\rightarrow \text{intersect}[\beta_1](\beta_2 \cup [e \mapsto n])$$

#### 4.1.5. Bag summation

If we consider bags  $\beta_1$  and  $\beta_2$  over the nonnegative integers then the summation of bags, denoted  $\beta_1 + \beta_2$ , referred to above may be formally specified by

$$\text{sum}: (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1) \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1)$$

$$\text{sum}[\theta]\beta \triangleq \beta$$

$$\text{sum}[[e \mapsto n] \cup \beta_1]\beta_2 \triangleq \text{sum}[\beta_1](+[e], \times[n])\beta_2$$

where  $(+[e], \times[n])\beta$  denotes the *application* of curried primitive recursive functions plus $[e]$  and times $[n]$  over the map  $\beta$ .

There is no need to speculate on the possibility of other bag-specific operations. Whatever operations are available to natural numbers are immediately applicable to bags. In addition, one can always apply the basic map operations to bags where needed. Consider, for instance, the removal of an element from a bag. I can specify this in terms of subtraction of natural numbers or, alternatively, choose to use the map removal operation. Further, there is sufficient material in Chapters 3 and 4 to build up the theory of bags and their homomorphisms. For example, I have already mentioned that the transfer function *items* is a  $\Sigma^*$ -homomorphism which may also be expressed as a reduction.

# THE METHOD OF THE IRISH SCHOOL

## 4.2. Applications

Two applications are discussed very briefly in this subsection, both of which illustrate the use of bags to solve problems. Other applications are to be found in the body of the thesis. Knuth also cites a few interesting applications (1983, 2:636). The first example, that of the bill of material, is taken from the work of Dines Bjørner (1988, 2: 456–466) and is a classic in the field of formal specification. Only that portion which relates the parts-explosion algorithm to the invariant is discussed in this Chapter. A more complete treatment in the style of the Irish School is presented in Chapter 6 where the domain of material requirements planning is introduced. The second application is drawn from the field of Trace Theory.

### 4.2.1. Bill of Materials

A bill of materials, *BOM*, is a structure used to represent the assembly of a product in terms of other subassemblies and basic parts. It may be presented in a variety of forms, one of which is the *indented* bill of material. For example, consider a hypothetical snow shovel, the assembly of which may be denoted by the following (Vollmann et al. 1984, 32):

```
1605 snow shovel
    13122 top handle assembly (1)
        457 top handle (1)
        082 nail (2)
        11495 bracket assembly (1)
            129 top handle bracket (1)
            1118 top handle coupling (1)

    048 scoop shaft connector (1)
    118 shaft (1)
    082 nail (2)
    14127 rivet (4)
    314 scoop assembly (1)
        2142 scoop (1)
        019 blade (1)
        14127 rivet (6)
```

In the *Meta-IV* this structure may be modelled by the domain equations

$$\gamma \in BOM = Pn \xrightarrow{m} P\_Rec$$

$$Pn = \dots$$

$$P\_Rec = Pn \xrightarrow{m} \mathbf{N}_1$$

where  $Pn$ ,  $P\_Rec$ , denote part names and part records, respectively. The significant point to note here is that these domain equations are similar to those of the abstract model of the file system,  $FS_0$ , which we have just considered. The domain equations for the bill of material are subject to the invariant:

$$\begin{aligned} inv-BOM(\gamma) &\triangleq \\ &(\forall pr \in rng \gamma)(dom pr \subseteq dom \gamma) \quad -- \text{all recorded} \\ &\wedge (\forall p \in dom \gamma)(p \notin Parts(p, \gamma)) \quad -- \text{no cycles} \end{aligned}$$

where  $Parts$  determines *all* the parts (subassemblies and basic parts) of which  $p$  is composed:

$$\begin{aligned} Parts: Pn \times BOM &\longrightarrow \mathcal{P}Pn \\ Parts(p, \gamma) &\triangleq \\ &(\text{let } ps = \{p' \mid p' \in dom \gamma: p' \in dom \gamma(p) \\ &\quad \vee (\exists p'' \in ps)(p' \in dom \gamma(p''))\} \\ &\text{in } ps) \end{aligned}$$

For the purpose of analysing the invariant, it is convenient to present a more abstract model of a bill of material in which the *number* of parts is ignored. I denote this model by  $BOM_0$  where the appropriate specification is

$$BOM_0 = Pn \xrightarrow{m} \mathcal{P}Pn$$

In the abstract, this form corresponds to that of the dictionary,  $DICT_6$ , given in Appendix A. Given this model, the invariant now takes the form

$$\begin{aligned} inv-BOM_0(\gamma) &\triangleq \\ &(\forall ps \in rng \gamma)(ps \subseteq dom \gamma) \quad -- \text{all recorded} \\ &\wedge (\forall p \in dom \gamma)(p \notin Parts(p, \gamma)) \quad -- \text{no cycles} \end{aligned}$$

where  $Parts$  is defined by the rather complex expression

## THE METHOD OF THE IRISH SCHOOL

$$\begin{aligned}
 & \text{Parts: } Pn \times BOM_0 \longrightarrow \mathcal{P}Pn \\
 & \text{Parts}(p, \gamma) \triangleq \\
 & \quad (\text{let } ps = \{p' \mid p' \in \text{dom } \gamma: p' \in \gamma(p) \\
 & \quad \quad \vee (\exists p'' \in ps)(p' \in \gamma(p''))\} \\
 & \quad \text{in } ps)
 \end{aligned}$$

It is difficult ‘to see’ what the *Parts* operation actually specifies. I call such a specification *existential*, in the sense that it states that the solution exists as a fixed-point. Indeed, the very occurrence of the logical expression,  $\exists p'' \in ps$ , signposts the existential nature. As it stands, the form of the specification does not exhibit constructability and it seems to me that it would very difficult to employ it in any formal proof that a particular operation on a bill of material,  $\gamma$ , conserves the invariant. A second objection may be raised. Whatever about the correctness of the invariant, it is equally problematic to explain how the form is derived or obtained in the first place.

On account of these aforesaid difficulties, I reject such existential forms and prefer those that are overtly constructive. Indeed, the above *Parts* operation is nothing other than a specialization of the fundamental ‘parts-explosion’ operation which is discussed below. Let us split the invariant into two parts, that which asserts that all are recorded—denoted *allrec* and that which asserts that there are no cycles—denoted *nonrec*. The first part may be given immediately in the Irish *Meta-IV* operator calculus

$$\text{allrec}(\gamma) \triangleq \cup / (\text{rng } \gamma) \subset \text{dom } \gamma$$

Note carefully that the  $\subseteq$  operator has been transformed into the  $\subset$  operator. This is a crucial result. For suppose that we had retained the  $\subseteq$  operator, and asserted that

$$\text{allrec}(\gamma) \triangleq \cup / (\text{rng } \gamma) \subseteq \text{dom } \gamma$$

was a basic lemma with respect to the model  $BOM_0$ . Then, following Lakatos’ method we would focus on the  $\subseteq$  operator and ask the question: Can it ever be the case that equality holds? In other words, we consider the case

$$\text{allrec}(\gamma) \triangleq \cup / (\text{rng } \gamma) = \text{dom } \gamma$$

and try to obtain a counter-example. After a few moments, it becomes clear that equality can only hold precisely if the second part of the invariant, *nonrec* fails, i.e.,

if and only if there is a cycle. Thus we have exposed a simple error in the original invariant. There are two comments worth making here. First, one may suppose that the error is simply a transcription error. Second, and most important of all, any mechanical technique which might be used for verification that the original invariant holds would always be successful if the invariant holds under the  $\subset$  operator. It is unlikely that it would ever detect that the *allrec* invariant ‘interferes’ with the *nonerec* part. It is only by approaching stated lemmas and theorems with suspicion, in the spirit of Lakatosian criticism, that one will ever expose such errors. Now we can formulate our first part of the invariant as a lemma which is independent of the second part of the invariant:

LEMMA 5.3. *A map  $\gamma \in BOM_0$  models a bill of material if the range of the map  $\gamma$  is a strict subset of its domain.*

$$\cup / (\text{rng } \gamma) \subset \text{dom } \gamma$$

This lemma is necessary but not sufficient. We must also consider the second part of the invariant. Again, in the Irish VDM, one does not rush on to the next part. It is critical to explore other ways of expressing the *allrec* invariant. An obvious next stage is to consider employing the use of the inverse image operator to give

$$\text{allrec}(\gamma) \triangleq \cup / (\text{rng } \gamma) \subset \gamma^{-1}(\text{rng } \gamma)$$

where,  $\gamma^{-1}(S)$ , is the short form for  $\cup / (\mathcal{P}\gamma^{-1})S$ , and thus eliminating the explicit reference to the domain. This formulation has an æsthetic appeal. But, the very use of the primitive application of the inverse image operator,  $\mathcal{P}\gamma^{-1}$ , leads us to identify further intrinsic structure. Specifically, it produces the set of partitions of the domain. One distinguished partition, which we shall denote by  $\varpi_\emptyset = \{p \mid p \in \text{dom } \gamma, \gamma(p) = \emptyset\}$ , contains all the basic parts. What of the others? They can only be the singleton sets  $\varpi_q, q \in \text{dom } \gamma \setminus \varpi_\emptyset$ . These are (1) the partition containing the product or top level assembly, denoted  $\varpi_\gamma$ , and (2) the subassembly partitions  $\varpi_q$ . We can further conclude that if a partition was formed which was neither of the two types mentioned, then we have a problem of part name aliasing. This latter observation is of a practical nature in an implementation, where we must take care that part names are unique. Our model implicitly defines them to be unique. There is one other possibility that might arise in practice and that, only

## THE METHOD OF THE IRISH SCHOOL

at the reification stage. It is possible that a pair of distinct subassemblies consists of different numbers of the same subassemblies and/or basic parts and are distinct therefore only with respect to number and not with respect to constituents. In that case the abstract model  $BOM_0$  must be deemed to be inadequate. For the present, we shall assume that such is not the case and, hence, we have a second basic lemma

LEMMA 5.4. *A map  $\gamma \in BOM_0$  models a bill of material if the inverse image of the range results in a set of partitions of the domain of  $\gamma$  such that each partition is either a singleton set or the set of basic parts.*

A further transformation of the form of the *allrec* invariant is possible. One may eliminate the subset operator and replace it with the primitive application of the characteristic function to give

$$allrec(\gamma) \triangleq \wedge / \circ \chi \llbracket rng \gamma \rrbracket \circ \gamma^{-1}(rng \gamma)$$

where, of course, we must explicitly state that equality is ruled out:

$$\cup / (rng \gamma) \neq \gamma^{-1}(rng \gamma)$$

Let us leave the second part of the invariant for the moment and consider instead the parts-explosion algorithm which is fundamental to the very concept of a bill of material:

$$\begin{aligned} Parts\_Explosion: Pn \times BOM &\longrightarrow TBL \\ Parts\_Explosion(p, \gamma) &\triangleq \\ Explosion(\gamma(p), 1, \theta)(\gamma) \end{aligned}$$

where

$$\begin{aligned} Explosion: (Pn \xrightarrow[m]{} \mathbf{N}_1) \times \mathbf{N}_1 \times TBL &\longrightarrow (BOM \longrightarrow TBL) \\ Explosion(trees, n, \tau)(\gamma) &\triangleq \\ \text{if } trees = \theta & \\ \text{then } \tau & \\ \text{else (let } p \in dom\ trees \text{ in} & \\ \text{let } \tau' = ((\gamma(p) = \theta) \rightarrow & \\ ((p \in dom \tau) \rightarrow \tau + [p \mapsto \tau(p) + n \times trees(p)], & \\ \mathbf{T} \rightarrow \tau \cup [p \mapsto n \times trees(p)], & \\ \mathbf{T} \rightarrow Explosion(\gamma(p), n \times trees(p), \tau)(\gamma)) & \\ \text{in } Explosion(trees \setminus \{p\}, n, \tau')(\gamma)) & \end{aligned}$$

I adapted this specification from the original text with only slight modifications. It computes the table ( $\tau \in TBL = Pn \xrightarrow{m} \mathbf{N}_1$ ) of basic parts which make up an assembly (Bjørner 1988, 465). I considered this form of the algorithm to be too complex for the purposes I had in mind. In trying to understand its operation with a view to explaining it to others, I soon realised that the algorithm might be simplified through the use of the bag domain. The table is, in fact, a bag and moreover, a part record is also a bag. Using bag arithmetic and a little rearranging, which is the result of considerable experimentation with different representational forms from the perspective of æsthetics, I obtained the following specification, where  $\mathcal{E}$  denotes ‘parts explosion’:

$$\begin{aligned} \mathcal{E}: Pn \times BOM &\longrightarrow (BOM \times TBL) \\ \mathcal{E}(p, \gamma) &\triangleq \mathcal{E}_1[\![\gamma(p)]\!](\gamma, \theta) \end{aligned}$$

where  $\mathcal{E}_n, \forall n \in \mathbf{N}_1$ , is the tail-recursive form

$$\begin{aligned} \mathcal{E}_n: (Pn \xrightarrow{m} \mathbf{N}_1) &\longrightarrow (BOM \times TBL) \longrightarrow (BOM \times TBL) \\ \mathcal{E}_n[\![\theta]\!](\gamma, \tau) &\triangleq (\gamma, \tau) \\ \mathcal{E}_n[\![p \mapsto m] \cup t]\!](\gamma, \tau) &\triangleq \\ \gamma(p) = \theta & \\ &\rightarrow \mathcal{E}_n[\![t]\!](\gamma, \tau \oplus n \otimes [p \mapsto m]) \\ &\rightarrow \mathcal{E}_n[\![t]\!] \circ \mathcal{E}_{m \times n}[\![\gamma(p)]\!](\gamma, \tau) \end{aligned}$$

The function of the algorithm is now clearer. The  $\gamma$  is a read-only structure;  $\tau$  is the result. In the case that a basic part is being processed,  $\gamma(p) = \theta$ , then the table is updated to reflect that part’s contribution to the overall total; otherwise, the sub-assembly is expanded and the algorithm recurses, with the appropriate modification to the multiplier factor  $m \times n$ .

Let us now generalize the algorithm to compute an ‘all’ parts explosion, i.e., construct a table that records the contribution of subassemblies as well as basic parts to the structure of an assembly:

$$\begin{aligned} \mathcal{E}_n[\![\theta]\!](\gamma, \tau) &\triangleq (\gamma, \tau) \\ \mathcal{E}_n[\![p \mapsto m] \cup t]\!](\gamma, \tau) &\triangleq \mathcal{E}_n[\![t]\!] \circ \mathcal{E}_{m \times n}[\![\gamma(p)]\!](\gamma, \tau \oplus n \otimes [p \mapsto m]) \end{aligned}$$

The interpretation of this form is indeed very straightforward. Using the language of graph theory, for the bill of material is nothing other than an instance of a directed acyclic graph, one may describe the explosion algorithm as a ‘note-taking’ of a node

## THE METHOD OF THE IRISH SCHOOL

as one passes through it, such ‘notes’ being recorded in the table  $\tau$ . Moreover, as written, the algorithm may also be regarded as a depth-first traversal of the bill of material,  $\gamma$ . Finally, it is clear that commuting the functions  $\mathcal{E}_n$  and  $\mathcal{E}_{m \times n}$  suggests breadth-first traversal. I wish to draw attention here to the careful choice of language that I use here: ‘may be regarded’ and ‘suggests’. The specification, as written, is completely general in so far that  $\mathcal{E}_n[[t]]$  only indicates a ‘lateral’ development, whereas  $\mathcal{E}_{m \times n}[[\gamma(p)]]$  signifies a ‘downward’ development. The argument in question is of the form of a map which does not have inherent order!

One specialization of the algorithm will now lead to the original parts explosion algorithm:

$$Parts\_Explosion(p)(\gamma) \triangleq \text{let } (\gamma, \tau) = \mathcal{E}_1[[p]](\gamma, \theta) \text{ in } \gamma^{-1}(\theta) \triangleleft \tau$$

where the inverse image  $\gamma^{-1}(\theta)$  denotes the set of basic parts. Initially, I thought that the following second specialization would give me a constructive definition of *Parts* for the invariant:

$$Parts(p, \gamma) \triangleq \text{let } (\gamma, \tau) = \mathcal{E}_1[[p]](\gamma, \theta) \text{ in } \text{dom } \tau$$

Indeed it does; but only if the structure,  $\gamma$ , is really a bill of material. Consider the counter-example

$$\gamma = [p \mapsto [p \mapsto 1]]$$

Using this to test the parts-explosion algorithm results in non-termination. I have almost obtained the solution I am seeking. Recalling that the parts-explosion algorithm is really a graph-traversal algorithm in disguise, then I may resolve the problem by using the concept of recording the nodes that I have visited. I leave the presentation of this solution to the next Chapter.

Now, of course, there is another interesting question that one would like to ask about the parts-explosion algorithm: Is it correct for a well-formed bill of material? If one were to follow the method of the English School of the *VDM*, then it would be necessary to formulate appropriate pre- and post-conditions and then prove formally that the post-condition follows from the pre-condition. I have argued that it is indeed correct, for how could it be otherwise? My intuition tells me that it must be so. The form tells me that it must be so. Before I demonstrate by another way that it is indeed correct, I quote at length a passage from Kant that nicely characterises the freedom allowed in the choice of method of proof in the Irish School of the *VDM*.

I do not agree with all the sentiments expressed, for within this quotation may be identified precisely that flaw in Kantian philosophy which was exposed by the ‘discovery’ of non-Euclidean geometries alluded to earlier. The passage is remarkable also to the extent that it reflects something of the philosophy behind the Irish School of the *VDM*:

“An apodeictic proof can be called a demonstration, only in so far as it is intuitive. Experience teaches us what is, but does not teach us that it could not be other than what it is. Consequently, no empirical grounds of proof can ever amount to apodeictic proof . . . Mathematics alone, therefore, contains demonstrations, since it derives its knowledge not from concepts but from the construction of them, that is, from intuition, which can be given *a priori* in accordance with the concepts. Even the method of algebra with its equations, from which the correct answer, together with its proof, is deduced by reduction, is not indeed geometrical in nature, but is still constructive in a way characteristic of the science”. [*charakteristische Konstruktion*. The meaning in which Kant uses this phrase is doubtful. It might also be translated ‘construction by means of symbols’—Trans.] (Kant [1781, 1787] 1929, 590).

I now propose to translate the problem into an algebraic one. Consider a typical entry in the bill of material of the form

$$[p \mapsto [x_1 \mapsto a_1, x_2 \mapsto a_2, \dots, x_k \mapsto a_k, \dots, x_n \mapsto a_n]]$$

This may be mapped onto the polynomial

$$p = a_1x_1 + a_2x_2 + \dots + a_kx_k + \dots a_nx_n$$

where the part names are  $x_1, \dots, x_n$ , and the number of parts are the coefficients  $a_1, \dots, a_n$ . Let the basic parts be the indeterminates  $y_j$ . Then the entire bill of material may be mapped onto a system of equations where the left-hand-side is some assembly and the right-hand-side an expression of the immediate constituent parts. Suppose, without loss of generality, that  $x_k$  is a composite part. Then it may be expressed in the form  $x_k = b_1x'_1 + \dots + b_ix'_i + \dots + b_rx'_r$ . Elimination of  $x_k$  from any other equation (or polynomial expression) gives

$$p = \dots + a_k(b_1x'_1 + \dots + b_ix'_i + \dots + b_rx'_r) + \dots$$

## THE METHOD OF THE IRISH SCHOOL

In other words  $x_k$  contributes  $a_k b_i x_i'$  to the result. Computation of the parts explosion is nothing more than the reduction of the system of equations to a single equation of the form

$$q = c_1 y_1 + \dots + c_j y_j + \dots + c_m y_m$$

which may be represented by the bag

$$[y_1 \mapsto c_1, \dots, y_j \mapsto c_j, \dots, y_m \mapsto c_m]$$

A simple example of a bill of material mapped into a system of polynomial equations will convince one of the correctness of the algorithm. What could be more clearer? I might add that this simple translation mechanism used in the above demonstration is very similar to the use of generating functions for multisets given in (Knuth 1981, 2:636).

The bill of material is but one of the essential components in the problem domain of material requirements planning in manufacturing industry. I still have much to say about it in the next Chapter. I would like to conclude this introduction to the problem domain by remarking how a physical entity such as a bag in SCRABBLE may be employed very effectively in reducing algorithmic complexity is a seemingly unrelated problem area.

### 4.2.2. Trace Theory

In this final illustration of the significance of the bag domain in formal specifications, I would like to stress the different interpretation that will be given to the concept of bag, one that initially seems to bear no relation to the bag of SCRABBLE. I mean to say that a bag may be interpreted as a collection of counters.

The notion of ‘occurrence’ of an element in a string, which is just a sequence of elements, is an important one in Trace Theory (Diekert 1990). The following definition is taken from Mazurkiewicz (1987, 282): “With each string an ordered set can be associated, namely that of symbol occurrences. Let  $w = a_1 a_2 \dots a_n$  be a string,  $n \geq 0$ ; each ordered pair  $(a_i, n_i)$  with  $n_i = \text{card} \{j \mid j \leq i, a_i = a_j\}$  is called an occurrence of  $a_i$  in  $w$ ”. Let  $occ$  denote the transformation that maps a string into a sequence of occurrences. Thus, for example, the string  $abacba$  is transformed into  $\langle (a, 1), (b, 1), (a, 2), (c, 1), (b, 2), (a, 3) \rangle$ . Now it is clear that such a sequence bears a strong resemblance to a bag. Therefore, it is hypothesised that the construction or computation of such an ordered set may readily be obtained by generalising the

items operation. Indeed, the use of a bag as a sort of temporary ‘counter’ mechanism is the key. For æsthetic purposes I define *occ* as:

$$\begin{aligned} \text{occ}: \Sigma^* &\longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1) \times (\Sigma \times \mathbf{N}_1)^* \longrightarrow (\Sigma \xrightarrow{m} \mathbf{N}_1) \times (\Sigma \times \mathbf{N}_1)^* \\ \text{occ} \llbracket \Lambda \rrbracket (\beta, \omega) &\triangleq (\beta, \omega) \\ \text{occ} \llbracket \langle e \rangle \wedge \tau \rrbracket (\beta, \omega) &\triangleq \\ &\text{occ} \llbracket \tau \rrbracket (\beta \oplus [e \mapsto 1], \omega \wedge \langle (e, (\beta \oplus [e \mapsto 1])(e)) \rangle) \end{aligned}$$

The correctness of the definition is obvious. The bag  $\beta$  acts as a set of counters for the string. At each occurrence of  $e$  in the string, the corresponding counter is incremented  $\beta \oplus [e \mapsto 1]$  and the pair  $(e, (\beta \oplus [e \mapsto 1])(e))$  is the result. The ordering of occurrences follows naturally from the manner of traversal of the string. For the purposes of constructing an imperative form, and say comparing it with similar algorithms in the literature (Goldwurm 1990, 74), one rewrites the above definition as a sequence of let constructs:

$$\begin{aligned} \text{occ} \llbracket \sigma \rrbracket (\beta, \omega) &\triangleq \\ &\text{if } \sigma = \Lambda \text{ then } (\beta, \omega) \\ &\text{else let } e = \text{hd } \sigma \text{ in} \\ &\quad \text{let } \beta' = \beta \oplus [e \mapsto 1] \text{ in} \\ &\quad \text{let } \omega' = \omega \wedge \langle (e, \beta'(e)) \rangle \text{ in} \\ &\quad \text{let } \sigma' = \text{tl } \sigma \text{ in} \\ &\quad \text{occ} \llbracket \sigma' \rrbracket (\beta', \omega') \end{aligned}$$

from which a skeletal while loop may be produced:

```

β ← θ;
ω ← Λ;
while σ ≠ Λ do
  e ← hd σ;
  β ← β ⊕ [e ↦ 1];
  ω ← ω ∧ ⟨ (e, β(e)) ⟩;
  σ ← tl σ;
end while
return ω

```

I have only considered that aspect of the bag which conceptually models a set of counters. There is yet another aspect which is conceptually ‘set-like’ with respect to operations such as membership, cardinality, union and intersection. For this

## THE METHOD OF THE IRISH SCHOOL

particular conceptual model I reserve the word ‘multiset’. I have not had any need to adopt this perspective of a bag, i.e., multiset, in practice. The concept of a bag as a semi-ring which is isomorphic to the natural numbers has served adequately to date.

One more aspect of the *Meta-IV* notation used in the Irish School still needs to be addressed—the domain of relations. The following section gives a very brief introduction, using the bill of material as a motivating example.

### 5. Domain of Relations

A (binary) relation is frequently considered to be a set of ordered pairs and defined as such in many formalised treatments of set theory (Royden 1968, 22). But Royden points out that “there is a difficulty with this approach in that  $=$ ,  $\in$ , and  $\subset$  are no longer relations”. Although, the relation domain is fundamental to  $\mathcal{Z}$ , it is not so in the *VDM Meta-IV*, in the sense that there is no special notation or set of operators peculiar to relations. Just as in the case of the bag, so it is also possible to incorporate the concept of relation into the *Meta-IV*. Indeed, all the usual set operators can readily be transferred to relations. Then why not do so and continue on to other topics? In the Irish School, as has been already noted earlier, the relation is just a function or map of the form

$$MODEL = X \xrightarrow{m} \mathcal{P}Y$$

Consider an abstract model of a ‘real’ dictionary which is considered to associate definitions with words. In the *Meta-IV* this is *adequately* and *faithfully* represented by

$$DICT = WORD \xrightarrow{m} \mathcal{P}DEF$$

Typical entries in a dictionary  $\delta$  are

- $[w \mapsto \{d_1, d_2, d_3\}]$ : which states that the word  $w$  has three definitions  $d_1$ ,  $d_2$ , and  $d_3$ ;
- $[w' \mapsto \emptyset]$ : which states that the word  $w'$  does not currently have any definitions.

It is not difficult to convince oneself that this model is a natural one. That it might also be modelled as a relation is, of course, true. But somehow, the notion of using a relation to model the dictionary is counter-intuitive. But in saying this I am stating a particular bias in my own conceptual model of the world. I can envisage the existence of people for whom the converse is the case—that the relation is more natural than the map, conceptually speaking.

In the theory of finite state machines, there is a standard way of constructing a nondeterministic recogniser for the language generated by a regular grammar. The transition function for such a recognizer belongs to the domain

$$DELTA = Q \times \Sigma \xrightarrow{m} \mathcal{P}Q$$

where  $Q$  denotes the set of states and  $\Sigma$  the input alphabet. Typical entries in a given  $\delta$  are of the form

- $[(q, i) \mapsto \{q_1, q_2\}]$ : which states that the machine in state  $q$  upon input  $i$  may transition nondeterministically to either state  $q_1$  or  $q_2$ ;
- $[(q, i) \mapsto \{q'\}]$ : which states that the machine will deterministically transition to state  $q'$ ;
- $[(q, i) \mapsto \emptyset]$ : which states that there is no such transition.

Again, like the dictionary, one could have modelled the transition diagram as a relation. But that is not the usual way of doing things.

Technically, in Chapter 3, I defined a relation  $\mathcal{R}$  from set  $X$  to set  $Y$  to be a map  $\mathcal{R}$  from  $X$  to the power set of  $Y$ :

$$\mathcal{R} = X \xrightarrow{m} \mathcal{P}Y$$

which is a well-established result in automata theory (Eilenberg 1974, 1:2).

There are, of course, applications where the converse may be true—that it is more natural to use the relation rather than the map as the appropriate structure. For example, in PROLOG, it is customary to declare facts relationally. But in the practice of software engineering with PROLOG, it is also customary to program relations as elements of the abstract domain  $X \xrightarrow{m} Y^*$ , which may be considered a reification of  $X \xrightarrow{m} \mathcal{P}Y$ . If one really wishes to model a domain using relations as sets of ordered pairs, then the appropriate abstract domain is obviously of the form

$$MODEL = \mathcal{P}(X \times Y)$$

## THE METHOD OF THE IRISH SCHOOL

Irrespective as to whether one believes that the relation or the map is the more appropriate structure to use in a given situation, it is clearly important to express the relationship between the two types of model,  $\mathcal{P}(X \times Y)$  and  $X \xrightarrow{m} \mathcal{P}Y$ , respectively. This is now addressed. Here we are not so much interested in the end-result, as in the manner in which the transformations are expressed.

### 5.1. Transfer Operations

To present the discussion of the relationship between maps and relations, it will be convenient to refer again to a concrete example from the *VDM Meta-IV* literature—the bill of material (Bjørner 1988, 2:456), which has already been introduced earlier in this Chapter. Ignoring all details as to the number of parts that are required for a particular assembly, then the bill of materials  $BOM_0$  may adequately be represented by the map

$$\gamma_m \in BOM_0 = X \rightarrow \mathcal{P}X$$

where the very abstract domain  $X$  has been chosen to denote ‘part name’. A typical instance of a bill of material is given by

$$\gamma_m = \begin{array}{l} [x_1 \mapsto \{x_2, x_3\}, \quad x_2 \mapsto \{x_4, x_5\}, \quad x_3 \mapsto \{x_4, x_6\} \\ x_4 \mapsto \emptyset, \quad x_5 \mapsto \emptyset, \quad x_6 \mapsto \emptyset] \end{array}$$

Note carefully that a distinction has been drawn between a basic part, such as  $x_4 \mapsto \emptyset$ , and a composite part, such as  $x_2 \mapsto \{x_4, x_5\}$ . Technically, it is not necessary to record the basic parts, if and only if the universe of discourse is clearly understood; they can always be computed.

Alternatively, the bill of material may be represented by a relation:

$$\gamma_r \in BOM_0 = \mathcal{P}X^2$$

with the same instance represented by

$$\begin{aligned} \gamma_r = & \{(x_1, x_2), \quad (x_1, x_3), \\ & (x_2, x_4), \quad (x_2, x_5), \\ & (x_3, x_4), \quad (x_3, x_6)\} \end{aligned}$$

Observe that there is no mechanism for explicitly distinguishing the basic parts. This may be resolved by introducing an auxilliary unary relation for such parts, or by computing those elements,  $x$ , which are not in the ‘domain’ of the relation. I will define below precisely what I mean by ‘domain’ of a relation.

5.1.1. Conversion from a Map to a Relation

The transfer function, `maptorel`, which converts a map,  $\mu \in (X \xrightarrow{m} \mathcal{P}Y)$ , to a relation,  $\rho \in \mathcal{P}(X \times Y)$ , is given as a tail-recursive definition:

$$\begin{aligned} \text{maptorel}: (X \xrightarrow{m} \mathcal{P}Y) &\longrightarrow (\mathcal{P}(X \times Y) \longrightarrow \mathcal{P}(X \times Y)) \\ \text{maptorel}[\emptyset]\rho &\triangleq \rho \\ \text{maptorel}[\mu]\rho &\triangleq \\ \text{let } x \in \text{dom } \mu \text{ in} & \\ \text{let } S = \mu(x) \text{ in} & \\ \text{cases } S: & \\ \emptyset \rightarrow \text{maptorel}[\{x\} \leftarrow \mu]\rho & \\ \rightarrow \text{let } y \in S \text{ in} & \\ \text{maptorel}[\mu + [x \mapsto S - \{y\}]](\rho \uplus \{(x, y)\}) & \end{aligned}$$

A few brief remarks are probably in order. First, a basic part contributes nothing to the relation; this is denoted by the expression  $\emptyset \rightarrow \text{maptorel}[\{x\} \leftarrow \mu]\rho$ . Thus, for the simple example

$$\mu = [x \mapsto \{y\}, y \mapsto \emptyset]$$

the resulting relation is  $\rho = \{(x, y)\}$ . Second, in the case of a composite part, the relation is built one piece at a time. Note also that I have chosen to employ the  $\uplus$  operator to signal the fact that the element  $(x, y)$  is uniquely added to the relation  $\rho$ .

5.1.2. Conversion from a Relation to a Map

Turning now to the *inverse* transfer operation, the tail-recursive definition is given simply by

$$\begin{aligned} \text{reitmap}: \mathcal{P}(X \times Y) &\longrightarrow (X \xrightarrow{m} \mathcal{P}Y) \longrightarrow (X \xrightarrow{m} \mathcal{P}Y) \\ \text{reitmap}[\emptyset]\mu &\triangleq \mu \\ \text{reitmap}[\rho]\mu &\triangleq \\ \text{let } (x, y) \in \rho \text{ in} & \\ \text{reitmap}[\rho - \{(x, y)\}](\mu \oplus [x \mapsto \{y\}]) & \end{aligned}$$

## THE METHOD OF THE IRISH SCHOOL

where the  $\oplus$  operator is defined by

$$\begin{aligned} \mu \oplus \theta &\triangleq \mu \\ \theta \oplus \mu &\triangleq \mu \\ \mu \oplus [x \mapsto S] &\triangleq \begin{cases} \mu + [x \mapsto \mu(x) \cup S], & \text{if } x \in \text{dom } \mu; \\ \mu \cup [x \mapsto S], & \text{otherwise.} \end{cases} \end{aligned}$$

with signature

$$- \oplus -: (X \xrightarrow{m} \mathcal{P}Y) \times (X \xrightarrow{m} \mathcal{P}Y) \longrightarrow (X \xrightarrow{m} \mathcal{P}Y)$$

This is a direct analogue of the addition of bags given in the previous section. Properties of this operator are inherited directly from those of set union, just as the bag operation inherited properties from addition of natural numbers. There is now enough evidence to suggest that operations, which are specified in terms of guarded map extend and map override operators, are themselves operators of some importance. Looking back at the put command of the file system, I hypothesise that with a little more thought and analysis, it can probably be turned into an operator, a problem I will leave for future work.

Using the same example that was employed for the maptorel transfer function, we note that  $\rho = \{(x, y)\}$  is transformed into  $\mu = [x \mapsto \{y\}]$ . Clearly, the function *reldom* is not strictly the inverse of *map*. To solve this small problem, let us introduce an injection function  $j: y \mapsto [y \mapsto \emptyset]$ . Given  $\mu = \text{reldom}[\rho]\theta$ , then the set of elements which must be added to  $\mu$  is given by  $S = \text{rng } \mu - \text{dom } \mu$ . The expression  $\cup / \circ (\mathcal{P}j)S$  computes the required ‘missing’ information.

### 5.1.3. Theorem and Proof

This Chapter has been more concerned with the concept of proof, as used in the Irish School, rather than forms of notation. Let us look at the *reldom* function and identify an invariant that must be conserved. Since it is tail-recursive, one obvious invariant would appear to be  $k = \text{dom } \rho \cup \text{dom } \mu$ , where I assume an intuitive understanding of what it means to apply a domain operator to a relation. To prove that it is indeed invariant we must demonstrate that

$$\text{dom}(\rho - \{(x, y)\}) \cup \text{dom}(\mu \oplus [x \mapsto \{y\}]) = \text{dom } \rho \cup \text{dom } \mu$$

Using the operator calculus, it is clearly sufficient if we can establish that

$$\text{dom}(\rho - \{(x, y)\}) \cup \text{dom}(\mu \oplus [x \mapsto \{y\}]) = (\{x\} \triangleleft \text{dom } \rho) \cup (\text{dom } \mu \cup \{x\})$$

and thus I must verify that  $\text{dom}(\rho - \{(x, y)\}) = \{x\} \triangleleft \text{dom } \rho$  and  $\text{dom}(\mu \oplus [x \mapsto \{y\}]) = \text{dom } \mu \cup \{x\}$ . The appropriate definition of a domain operator applied to a relation,  $\rho \in \mathcal{P}(X \times Y)$ , is deduced by means of a simple illustration

$$\begin{aligned} \rho &= \{(x_1, y_1), (x_2, y_2), \dots, (x_j, y_j), \dots, (x_n, y_n)\} \\ \implies (\mathcal{P}\pi_1)\rho &= \{\pi_1(x_1, y_1), \pi_1(x_2, y_2), \dots, \pi_1(x_j, y_j), \dots, \pi_1(x_n, y_n)\} \\ &= \{x_1, x_2, \dots, x_j, \dots, x_n\} \end{aligned}$$

Therefore,  $\text{dom } \rho \triangleq (\mathcal{P}\pi_1)\rho$ . Similarly, the *rng* operator is defined by  $\text{rng } \rho \triangleq (\mathcal{P}\pi_2)\rho$ . From these definitions, one may derive the usual sorts of laws. For example,

$$\begin{aligned} \text{dom}(\rho_1 \cup \rho_2) &= \mathcal{P}\pi_1(\rho_1 \cup \rho_2) \\ &= \mathcal{P}\pi_1(\rho_1) \cup \mathcal{P}\pi_1(\rho_2) \\ &= \text{dom } \rho_1 \cup \text{dom } \rho_2 \end{aligned}$$

and, for the problem in hand,

$$\begin{aligned} \text{dom}(\rho - \{(x, y)\}) &= \text{dom}(\{(x, y)\} \triangleleft \rho) \\ &= \mathcal{P}\pi_1(\{(x, y)\} \triangleleft \rho) \\ &= \mathcal{P}\pi_1\{(x, y)\} \triangleleft \mathcal{P}\pi_1\rho \\ &= \{x\} \triangleleft \text{dom } \rho \end{aligned}$$

The proof of the other required result,  $\text{dom}(\mu \oplus [x \mapsto y]) = \text{dom } \mu \cup \{x\}$ , follows directly from the properties of the domain operator on maps.

The significance of the result is more a matter of identifying the need to determine the definition of relational operators such as *dom* and their properties arising from a particular problem to solve, rather than simply trying to find and prove an invariant. In other words, in the Irish School of the *VDM*, it is expected that significant lemmas and theorems arise as a natural consequence of using the operator calculus. The next stage is then to investigate the corresponding algebraic structures that provide a natural setting for the results that arise.

# THE METHOD OF THE IRISH SCHOOL

## 6. Summary

In the introduction to Chapter 4, I stated that proving theorems does not seem to be the essence of doing mathematics but that, nevertheless, one of the most important facets of doing formal specifications *is* theorem-proving. The statements seemed to be paradoxical. I have emphasised in this Chapter that theorem-proving has more to do with the process of discovering theorems rather than proving existing theorems. That there are such theorems for the *VDM* has already been adequately demonstrated; whether they would be discovered as readily using a purely formalist approach is open to question.

I have taken advantage of the availability of existing *VDM* specifications to demonstrate the method of the Irish School in the belief that they are as valuable for serious applications work in formal methods as, say, geometry and algebra problems are for mathematics. In particular, there is an urgent need to prove existing theorems in formal specifications *by different techniques*, as is the customary practice in mathematics.

I have freely taken for granted that the concept of a retrieve function, which guarantees that one model adequately and faithfully realises a more abstract model, is understood by practitioners of the Irish School of the *VDM*. It is a concept, fundamental to any School of the *VDM*. However, I would like to draw attention to the fact, which does not seem to be made in the published *VDM* literature, that the concept is already well understood in automata theory, under the name of *a covering* (Eilenberg 1976, 2: 9). Indeed, the hidden complexities of the file system case study which were identified in this Chapter would seem to bear some relation to the notion of *wreath product* of transformation semigroups (Eilenberg 1976, 2:26). The specific application of the latter to formal specifications is a goal for future research.

The next step in the presentation of the method of the Irish School of the *VDM* is to show by what means one produces specifications *ab initio*. I do not mean to say that we should consider the invention of specifications but rather the discovery of specifications. There is the construction of a specification from requirements; the reverse-engineering of a specification from a concrete realisation, and requirements from specifications; and the reuse of existing specifications to develop others in either a related or different problem domain. These matters are now touched upon.

# Chapter 6

## Discovering Specifications

### 1. Introduction

The analysis of an existing specification, whether written in the *VDM Meta-IV*, or some other formal specification language, is a relatively simple matter compared to discovering the specification in the first place. If we consider the case of Euclidean Geometry, for example, specification analysis is analogous to the study and proof of existing theorems. It is entirely a different matter to consider some concrete problem domain and attempt to apply our knowledge of Geometry. Beginning with this Chapter, I wish to indicate how one may approach the problem of applying the *VDM* to concrete problem domains.

In the introduction to Chapter 1, I stated that “formal software development consists of”, among other things, “a formal specification derived from requirements”. Conventionally, a requirement is the expression of a basic need, as distinct from a ‘want’. There is an enormous body of literature on the capture of requirements, a discipline which, in the domain of computing, used to be called ‘systems analysis’. A generic name for said discipline is ‘conceptual analysis’ (Sowa 1984: 294), a name which automatically triggers the notion of ‘conceptual model’, the subject matter of this thesis. Where human beings are to be part of the system to be, then it is customary to use the term ‘task analysis’. Requirements are usually expressed in the natural language of the client. The supplier/analyst, whose job it is to re-express these requirements with a view to building a computing system to meet the client’s needs, produces a ‘requirements specification’, which may consist of one or more documents. The ideal properties that such a requirements specification ought to possess are enumerated in the IEEE Guide to Software Requirements Specifications which is an ANSI standard (ANSI/IEEE Std 830-1984). Although it specifically addresses software requirements specifications, the guide is also valid for systems

## DISCOVERING SPECIFICATIONS

requirements specification. Other more rigorous guidelines also exist—the STARTS Purchasers’ Handbook (STARTS 1986). Many different formalisms have been proposed for the expression of requirements specifications. Typical samples may be found in (Birrell and Ould 1985; Ceri 1986; Yadav et al., 1988). Unfortunately, actual requirements specifications for real projects are usually proprietary documents and all that one is likely to find in the literature are either anodyne or fragmentary examples—the library problem (Wing 1988; Bjørner 1988, 2: 661–79; Diller 1990, 173–93), image processing system (Birrell and Ould 1985). It seems pointless to me to add to this collection. Therefore, for the purpose of illustration, I have chosen to apply the *VDM* at the meta-level of specification of a systems requirements specification which I present in the next section.

It would be incorrect to suppose that all development **must** begin with requirements. In practice it is rarely the case that a system must be specified *ab initio*. On the contrary, ‘new’ systems are frequently variational designs or extensions of existing systems. This is entirely analogous to the situation in manufacturing. In such cases, constraints often have more weight than requirements. The Bill of Material, introduced in the previous Chapter, provides the setting for a study of the problem domain of Material Requirements Planning. In subsequent sections, I will demonstrate how one extends an existing specification, that of the bill of material, to embrace a more complex realistic system in manufacturing.

## 2. Requirements Model

It is possible to build up a model of requirements specifications without committing oneself to a definition of a requirement. In choosing a starting point, an analyst will often begin at that point which he/she finds most interesting or which is considered to be most important. In my own case, the concept of requirements traceability proved to be an intellectually stimulating domain:

“An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. Two types of traceability are recommended: (1) Backward traceability (that is, to previous stages of development ) depends upon each requirement explicitly referencing its source in previous documents.

(2) Forward traceability (that is, to all documents spawned by the SRS) depends upon each requirement having a unique name or reference number.

When a requirement in the SRS represents an apportionment or a derivative of another requirement, both forward and backward traceability should be provided ” (ANSI/IEEE Std 830–1984, 13).

Let  $REQ$  denote the domain of requirements, which for the present is not further detailed:

$$REQ = \dots$$

Let  $Rid$  denote the domain of requirements identifiers, again not further specified:

$$Rid = \dots$$

For the purpose of obtaining traceability it is essential that one specify a mapping between requirements identifiers and requirements:

$$\tau \in TRACE = Rid \xrightarrow{m} REQ$$

We wish to eliminate the possibility (for the moment) that the same requirement can be identified by more than one requirement identifier, i.e., the map must be 1–1:

$$\begin{aligned} inv-TRACE: TRACE &\longrightarrow \mathbf{B} \\ inv-TRACE(\tau) &\triangleq (\forall r \in rng \tau)(\tau^{-1}(r) = \{i\}) \end{aligned}$$

where I have chosen to use the inverse image  $\tau^{-1}(r) = \{i \mid i \in dom \tau, \tau(i) = r\}$ .

## DISCOVERING SPECIFICATIONS

It is customary to group related requirements together according to some criteria. Consequently, I introduce the notion of requirements sets:

$$S \in REQ\_SET = \mathcal{PREQ}$$

where I am willing to consider that the same requirement may belong to more than one requirement set, i.e., for some  $S_j$  and  $S_k$  in  $REQ\_SET$ , it may be the case that  $S_j \cap S_k \neq \emptyset$ . It is now necessary to consider in what manner such requirements sets may be uniquely identified. Let us posit the existence of a domain of requirements set identifiers,  $RSid$ , where, of course, it is imperative that  $Rid \cap RSid = \emptyset$ . Now at the back of my mind I have an intuition that the actual structure of a requirements set identifier ought to bear some relationship to the identifiers of the requirements contained therein, a relationship that is complicated due to the possibility of a requirement belonging to more than one requirements set. This is a classic sort of problem, the obvious solution to which is to insist on disjoint requirements sets. In developing the current model, such an issue is flagged but not permitted to interfere with the overall development. The trace map is now augmented to accommodate the new concepts:

$$TRACE = (Rid \xrightarrow{m} REQ) \mid (RSid \xrightarrow{m} REQ\_SET)$$

and we will also extend the corresponding invariant to ensure that the ‘new’ portion is also 1-1:

$$\begin{aligned} & inv-TRACE: TRACE \longrightarrow \mathbf{B} \\ & inv-TRACE(\tau) \triangleq \\ & (\forall r \in rng(Rid \triangleleft \tau))(\tau^{-1}(r) = \{i\}) \\ & \wedge (\forall S \in rng(RSid \triangleleft \tau))(\tau^{-1}(S) = \{j\}) \\ & \wedge \dots \end{aligned}$$

Note the use of the ellipsis which indicates that there is probably more to the invariant than is already stated. For example, one will also want to ensure that that every requirement in a requirement set has a unique identifier associated with it:

$$(\forall r \in S)(\exists! i \in dom(Rid \triangleleft \tau)(\tau(i) = r))$$

Note that I do not insist that all requirements listed in the trace must be members of some requirements set and, conversely, I do not require that the requirements in a requirements set should be listed independently in the trace. I am satisfied to

terminate the development of the model at this point. There is sufficient specification material, the semantic domain, *TRACE*, for discussion and elaboration. In particular, it is easy to posit the existence of syntactic domains which are a *necessary* consequence of the *Meta-IV* set and map operators. The appropriate semantic functions then follow naturally. From the perspective of the Irish School of the *VDM* there is also the matter of the elimination of the universal and existential quantifiers which I have used. I do not deem it necessary to present the details here.

A different (conventional) view of requirements may be obtained by supposing that they are presented in documents:

$$\begin{aligned} d \in \text{DOC} &= \text{CHAPTER}^* \\ c \in \text{CHAPTER} &= \text{PAGE}^* \\ p \in \text{PAGE} &= \text{REQ}^* \end{aligned}$$

where I have specified that a requirement is not ‘split’ across page boundaries. Nor do I assume that requirements are identified other than by their unique location within a document— $d[i, j, k]$  denotes the  $k$ th requirement on the  $i$ th page of the  $i$ th chapter. It behoves me to state explicitly that requirements within a chapter are unique. I do permit the possibility that the same requirement may occur in different chapters. Such an arbitrary decision may be justified by the particular conceptual view that I wish to adopt with respect to a requirements document. Such arbitrariness recalls Saussurean linguistics. The proposed invariant, is, therefore:

$$\begin{aligned} \text{inv-CHAPTER: CHAPTER} &\longrightarrow \mathbf{B} \\ \text{inv-CHAPTER}(c) &\triangleq \text{unique}^*(c) \wedge \text{unique}(\wedge / c) \end{aligned}$$

where the predicate *unique*, which we have met before, is defined over a page:

$$\begin{aligned} \text{unique: PAGE} &\longrightarrow \mathbf{B} \\ \text{unique}(p) &\triangleq \\ &(p = \Lambda) \\ &\rightarrow \text{true} \\ &\rightarrow \text{let } p = \langle r \rangle \wedge \tau \text{ in} \\ &\quad \neg \chi_\tau(p) \wedge \text{unique}(\tau) \end{aligned}$$

Note that  $\text{unique}^*(c)$  extends the *unique* predicate to all pages in a chapter and  $\text{unique}(\wedge / c)$  covers an entire chapter. The latter predicate subsumes, of course, the former. To be absolutely rigorous one should ensure that there are no duplicate chapters.

## DISCOVERING SPECIFICATIONS

We now have two different traceability or tagging schemes for requirements—one based on some sort of requirements identifiers and the second based on location within a document structure. I am of the opinion that both ought to be preserved in the model. Clearly, the next issue to be addressed is to establish the relationship between the two traceability schemes:

$$\delta \in \text{DOC\_TRACE} = \text{Rid} \xrightarrow{m} \text{DOC\_IDX}$$

$$di \in \text{DOC\_IDX} = \text{CHPT\_IDX} \times \text{PAGE\_IDX} \times \text{REQ\_IDX}$$

where I will avail of the clause, for  $mk\text{-DOC\_IDX}(i, j, k)$  use  $[i, j, k]$ . The idea here is that the domain of requirements identifiers, *Rid*, is of paramount importance in indexing the requirements documents. In other words, I have effectively given primacy of place to the first traceability scheme that I introduced.

Now we are going to need to provide an invariant for this map. In particular, as it stands one has the possibility of two different requirements identifiers mapping to the same document index (and by implication to the same requirement which we have ruled out). Thus, I will want the map to be 1–1. On the other hand, I wish to retain the possibility that the same requirement may occur in different chapters. The ‘problem’ here is that we no longer would have a map but a relation, a problem which we have met already in the previous Chapter. Consequently, the ‘natural’ model which we need is

$$\text{DOC\_TRACE} = \text{Rid} \xrightarrow{m} \mathcal{P}\text{DOC\_IDX}$$

which is structurally isomorphic to a dictionary. What more is there to say? In a complete development of a requirements model we shall need to extend this traceability relationship to collections of requirements documents and indeed to other ‘documents’ such as formal specifications documents, design documents, programme texts, test scripts, etc.

Let us return now to the issue of the invariant for *DOC\_TRACE*. Considering an entry of the form

$$[i \mapsto \{[i_1, j_1, k_1], [i_2, j_2, k_2], \dots, [i_n, j_n, k_n]\}]$$

we certainly want to rule out the possibility of a chapter index *i* occurring more than once, for then we would have the situation where the same requirement occurred more than once in a given chapter, something which has already been prohibited by

an earlier invariant. Thus

$$\begin{aligned}
 \text{inv-DOC\_TRACE}: \text{DOC\_TRACE} &\longrightarrow \mathbf{B} \\
 \text{inv-DOC\_TRACE}(\delta) &\triangleq \\
 &(\forall i \in \text{dom } \delta)(\text{let } S = \delta(i) \text{ in} \\
 &\quad \text{let } \text{cil} = \hat{\ } / \circ \mathcal{P}j \circ \mathcal{P}\pi_1(S) \text{ in} \\
 &\quad \text{unique}(\text{cil}))
 \end{aligned}$$

where  $\pi_1$  is the projection operator of cartesian products and  $j$  is the usual injection operator that injects elements into singleton sequences. Rather than elaborating further on the overall system architecture of a requirements model, it is time to focus on some of the details that were omitted. The next section is dedicated to a consideration of the structure of a requirement  $r \in \text{REQ}$ .

### 3. A Requirement

Requirements are usually stated in the natural language of the client or customer. In real projects, a requirements document forms the baseline for a contract between the client and the supplier and consequently, very precise language must often be employed. As a very first rough approximation a requirement may be considered to be a sequence of words:

$$\text{REQ} = \text{WORD}^*$$

where a word is considered to be a sequence of characters taken from some underlying alphabet  $\Sigma$ . If we wish to be precise about what we mean by a word, then we may introduce a lookup dictionary

$$\delta \in \text{DICT} = \mathcal{P}\text{WORD}$$

and declare that a ‘word’  $w$  is in fact a word in the normal sense of that term if and only if it is contained in some dictionary  $\delta$ , acceptable to both parties. Indeed, such a dictionary may very well be dynamic in nature where both parties agree to add or delete certain words. Naturally, one may continue in this vein to distinguish various

## DISCOVERING SPECIFICATIONS

categories of words: nouns, verbs, etc., and make corresponding modifications to the dictionary.

I would now like to take a slightly different approach to the use of the *VDM* to model a requirement. Specifically, I propose to demonstrate that we consider requirements as nothing more than pieces of syntax and use the *VDM* in its traditional rôle for the specification of denotational semantics. Personally I consider a requirement to be simply a statement of a particular form.

**DEFINITION 6.1.** *A requirement is a statement about some action that must be performed. There must be an agent which is responsible for the action and there must be an object to be acted upon.*

Consequently, I model a requirement as a piece of abstract syntax

$$REQ :: ACT \times [AGENT] \times OBJ$$

where I deliberately emphasise the primary rôle of action over both agent and object.

For example, the requirement

*Subsystem X must support authentication.*

may be represented by the tree

$$mk-REQ((mk-ACT(\text{support}), mk-AGENT(\text{Subsystem } X), \\ mk-OBJ(\text{authentication})))$$

In addition, I allow for the use of the passive voice where an agent need not be explicitly mentioned. Thus, the requirement

*Authentication must be supported.*

is given by the abstract syntax tree

$$mk-REQ(mk-ACT(\text{support}), mk-AGENT(\text{nil}), mk-OBJ(\text{authentication}))$$

Earlier in the thesis I stated that the *VDM* trees are exactly equivalent to facts in PROLOG. In fact, the above abstract syntax trees may just as readily be expressed as the PROLOG facts

$$\text{req}(\text{act}(\text{support}), \text{agent}(\text{"Subsystem X"}), \text{obj}(\text{authentication})). \\ \text{req}(\text{act}(\text{support}), \text{agent}(\text{Unknown}), \text{obj}(\text{authentication})).$$

an identification which may be more forcefully expressed by using functors such as ‘mk\_Req’ in place of ‘req’, etc.

In practice, I prefer to use the conceptual graph notation of Sowa (1984) to represent such requirements, without adhering strictly to his conventions. Thus, the first requirement may also be represented by the form

[**act**: support]–  
     (*agnt*) → [**actor**: Subsystem *X*]  
     (*obj*) → [**act**: authentication]

where I have introduced the new category of ‘actor’. The corresponding abstract syntax in the *VDM* is simply

$$AGENT :: ACTOR \mid \dots$$

where I propose to allow for the possibility that there may be other entities which are agents but are not actors. In addition, I have further qualified ‘authentication’ as an act. Now it ought not be supposed that one is to write requirements in any of the forms given above. What I have simply suggested is that requirements may be cast into some normal form as a result of parsing. Moreover, I have demonstrated the equivalence of the three forms of representation.

There is currently emphasis being placed on ‘object-oriented’ requirements engineering in analogy to object-oriented programming, an emphasis which I feel is misplaced. Consider the term ‘authentication’. It is the object of the verb/act ‘to support’. However, it happens to be a substantive, derived from a verb ‘to authenticate’. Such substantives play an important rôle in requirements reification, a process by which one is led inexorably towards the discovery of other related requirements. Indeed, such is their importance that I have introduced the notion of a specific requirements test which I call the *deus ex machina* test, to be discussed below.

Let us consider what it ‘means’ to say that the object *authentication* may be interpreted as an act, from the perspective of the *VDM* abstract syntax. Since an act is the primary term in a requirement, then it is clear that an object may be specified by

$$OBJ :: ENTITY \mid REQ \mid \dots$$

and this is a satisfying result in the sense that abstract syntax simply mirrors classical grammars and it is not possible to obtain any interesting language from a (formal) grammar unless there are recursive rules.

## DISCOVERING SPECIFICATIONS

From the point of view of natural language processing, I have basically sketched a proposal that conforms to the Lexical-Functional Grammar approach (Winograd 1983, 1:328–43). It is lexical in the sense that a lexicon of domain terminology is an essential feature of the approach. However, I would like to emphasise most strongly that I am not interested here in issues of tool support for requirements capture and analysis; rather my goal is to exhibit the validity of application of the *VDM* to the requirements phase of system development as much as to requirements specification *per se*. Before turning to consider more complex detailed requirements that arise in practice, I now present two simple requirements tests that I have employed on real industrial projects: the *deus ex machina* requirements test and the communications requirements test.

### 3.1. The Deus ex machina Requirements Test

The origin of the term *deus ex machina*, i.e., ‘god out of the machinery’ derives from early Greek theatre where it was used to describe the device by which playwrights extricated their heroes from convoluted problems of plot by forcing the sudden appearance of a god (in the air) to resolve the issue. Aristotle, in his *Poetics*, Chapter 15, specifically refers to it as a “contrivance, a *deus ex machina*, as in the *Medea* . . .” (Ackrill 1987, 553). The literal text is “*απο μηχανης*”, i.e., ‘from the machine’ which means “‘divine intervention’, since in the Greek theatre gods were suspended over the scene by means of the *mêchanê* or crane” (Hutton 1982, 97).

I have chosen this name of a classical concept to denote the importance of the fact that in requirements documents, all of the players or actors must (eventually) be revealed; and that all said actors have been part of the main action or plot from the very beginning and are not brought in as an afterthought. From the point of view of conceptual models, the name of the test has been deliberately chosen to trigger associations both in the domains of theatre and in philosophy. I might have called the test the ‘look for missing information’ requirements test. But the corresponding impact would not be the same. *As an aside, I would like to emphasise another critical rôle that the term plays with respect to conceptual models in formal specifications.* Pólya employs the term to denote those solutions (to mathematical problems) that are presented by some trick “out of the blue” ([1962] 1981, 2:120). The same may be said of much of the formal specifications that are published in the

literature. With this foremost in my mind, I have taken pains to exhibit how and in what manner such specifications (and theorems and proofs) are discovered, not only in this Chapter which might have been entitled ‘Deus ex machina is anathema’, but also throughout the thesis. I have not done so in all cases lest the thesis become uncomfortably large. In mentioning Aristotle in conjunction with *deus ex machina*, I also wish to draw an association between the conceptual analysis being performed at the requirements level and Aristotle’s work on categories, the forerunner of all Artificial Intelligence and philosophical work (Sowa 1984).

Consider again the following typical ‘high-level’ requirement:

*Subsystem X will support authentication of subscribers*

The main action is that of ‘support’; the actor is ‘Subsystem X’ and the object of support is the ‘authentication’ of subscribers. We may reformulate this in conceptual graph notation as:

[**act**: support]–  
     (*agnt*) → [**actor**: Subsystem X]  
     (*obj*) → [**act**: authentication]

Now I replace the substantive ‘authentication’ with the present participle ‘authenticating’ in its place, to give

*Subsystem X will support the authenticating of subscribers*

One might quibble whether or not this is ‘good’ English style. I have noted that it is a form frequently employed by non-native English speakers which conveys an interesting sense of the action intended. Then it is moot to ask who or what is responsible for authenticating subscribers. It could be, of course, Subsystem X; or it could be some other actor. The important point to note is that the actor in question is not explicitly known at this point—a potential *deus ex machina*:

[**act**: support]–  
     (*agnt*) → [**actor**: Subsystem X]  
     (*obj*) → [**act**: authenticate *y*]  
 [**act**: authenticate *y*]–  
     (*agnt*) → [**actor**: ?]  
     (*obj*) → [**entity**: subscribers]

Referring to my theme that system development is analogous to the writing of a play, then the high-level requirements document should be concerned with the main

## DISCOVERING SPECIFICATIONS

actions of the plot. Major actors must be identified. This is the subject matter of the *deus ex machina* test. But it would be a strange sort of play if the actors all acted in isolation, i.e., the script consisted solely of stage directions and soliloquies. In practice, one expects to find some sort of interaction, dialogue, or communication. As an aside I wish to point out that this approach to requirements was also a major influence in developing the unified conceptual model of

structure  $\rightarrow$  communication  $\rightarrow$  behaviour

within the Irish School of the *VDM* which is the subject of Chapter 8.

### 3.2. The Communications Requirements Test

Frequently, one comes across subsystem requirements that speak of sending, transmitting, reporting, etc., information to some other subsystem or entity, such as the display screen of an MMI. Conversely, one finds requirements that speak about accepting, receiving, etc., information or messages from some source. Occurrence of either form of requirement necessarily implies that the other should occur. Consider the following requirement taken from the ANSI EIA/TIA Standard *Mobile Station–Land Station Compatibility Specification* for cellular mobile telecommunications systems (ANSI/EIA/TIA-553-1989, 2-13):

*Whenever a mobile station receives an overhead message train, the mobile station must compare  $SID_s$  with  $SID_r$ .*

Recasting this in the form of a conceptual graph gives us

[**act:** receive]–  
    (*rcpt*)  $\rightarrow$  [**actor:** mobile]  
    (*srce*)  $\rightarrow$  [**actor:** ?]

Quite obviously, this is a *deus ex machina* situation. However, it is distinguished precisely because one expects to find a requirement of the form

[**act:** send]–  
    (*agnt*)  $\rightarrow$  [**actor:** ?]  
    (*dest*)  $\rightarrow$  [**actor:** mobile]

somewhere else in the document. At the highest level of specification, it is clear that the unidentified ‘actor’ is the land station. However, in subsequent reification, some specific piece(s) of software will be responsible for sending overhead messages to the

mobile and consequently, the rôle of the major player—the land station—will be played by some other actor. The conceptual analysis is, of course, incomplete. No mention has been made of the object being received—the overhead message train. We extend the conceptual graph in the appropriate manner:

[**act**: receive]–  
 (*rcpt*) → [**actor**: mobile]  
 (*srce*) → [**actor**: ?]  
 (*obj*) → [**entity**: message]

and then expand this to conform to Sowa's template by including an indication of the means (the instrument) by which the message is received (1984, 412):

[**act**: receive]–  
 (*rcpt*) → [**actor**: mobile]  
 (*srce*) → [**actor**: ?]  
 (*obj*) → [**entity**: message]  
 (*instr*) → [**entity**: channel FOCC]

where FOCC denotes a forward control channel which is used by land stations to support system access by mobile stations (EIA/TIA-553-1989, 3-5). From the point of view of abstract syntax, it is clear how one may specify this particular type of communications requirement. The pattern is exactly analogous to that given before. However, it is easy to see that the *VDM* abstract syntax domains for a requirement will become very unwieldy. Therefore, now is the time to structure such domains in a manner which reflects the type of requirements themselves. Introducing the domain name *ReceiveREQ*, then I would write

$$\textit{ReceiveREQ} :: \textit{ACT} \times \textit{RECPT} \times \textit{SRCE} \times \textit{INSTR}$$

where verbs/acts such as 'recieve', 'accept', etc., all denote 'receive type' requirements. By so doing I am literally constructing an aristotelian type hierarchy as expounded in Sowa (1984). Finally, I wish to address the issue of low-level requirements and constraints, a topic that will take us directly back into the bill of material domain.

## DISCOVERING SPECIFICATIONS

### 3.3. Constraints, Context Conditions, and Invariants

A requirement without a constraint is like a *VDM Meta-IV* domain without an invariant. Consider the ‘authentication of subscribers’ example. With a little imagination we may apply this requirement to my own bank where ‘Subsystem *X*’ becomes *Bank teller X* and ‘subscriber’ translates to *account holder*. In the scenario that an account holder is cashing a personal cheque, it may very well be the case that the teller must authenticate the account holder. There is a natural time constraint on the act of authentication. Violation of the constraint is likely to lead to loss of custom. From a conceptual point of view I am of the opinion that it is best to keep constraints separate from, but of course linked to, the corresponding requirements. For the example in question, an appropriate piece of abstract syntax might be

$$CON :: ACT \times TIME$$

where the form  $mk\text{-}CON(mk\text{-}ACT(\text{authenticate}), mk\text{-}TIME(1))$  is to be interpreted as ‘the act of authentication is constrained to take place within 1 minute maximum’. In practice, it is quite feasible to link the constraint to an operating environment where the time may be dependent on customer service queues. Now the important point to note is that the constraint is given by a piece of abstract syntax which is structurally similar to that of a requirement, not the details or adequacy of the constraint itself.

In the *VDM* the term ‘context condition’ or ‘well-formedness constraint’ is frequently employed to denote the constraints on the validity of use of a piece of abstract syntax. For example, we may wish to state explicitly that a requirement is well-formed if and only if both the action and object are recorded in some lexicon:

$$\begin{aligned} is\text{-}wf\text{-}REQ: REQ &\longrightarrow DICT \longrightarrow \mathbf{B} \\ is\text{-}wf\text{-}REQ(mk\text{-}REQ(act, agnt, obj))\delta &\triangleq \\ \text{let } mk\text{-}ACT(x) = act, mk\text{-}OBJ(y) = obj &\text{ in} \\ Lkp[x]\delta \wedge Lkp[y]\delta & \end{aligned}$$

This is a constraint at the meta-level. The exact same approach may be employed for ordinary constraints on requirements. For example, suppose that the authentication requirement is subsequently refined, with respect to an automated teller machine, into

*Subsystem X must authenticate the PIN of the customer.*

There is also a ‘hidden’ communications requirement implied. Let us suppose that the ‘real’ customer requirement is a request for service:

$$RequestService \ :: \ CustId \times \ PIN \times \dots$$

then, we might wish to specify the constraint on this requirement in a classical *VDM* form of a context condition or well-formedness constraint:

$$\begin{aligned} is\text{-}wf\text{-}RequestService: RequestService &\longrightarrow \mathbf{B} \\ is\text{-}wf\text{-}RequestService(mk\text{-}RequestService(c, n, \dots)) &\stackrel{\triangle}{=} \\ is\text{-}wf\text{-}CustId(c) \wedge is\text{-}wf\text{-}PIN(n) \wedge \dots & \end{aligned}$$

where I still need to specify what it means for both a customer id,  $c$ , and a PIN,  $n$ , to be well-formed. Alternatively, I may prefer to use the notion of invariant on the domains  $CustId$  and  $PIN$  to convey the same information. An invariant constrains the domain.

I have adequately demonstrated the applicability of the *VDM* to the domain of requirements, both at the ‘usual’ level and at the meta-level. While I consider this to be rather obvious, it does need to be put in print. As recently as the *VDM’90* symposium in Kiel, FRG, April 1990, it was proposed by eminent researchers in  $\mathcal{Z}$  that  $\mathcal{Z}$  was more appropriate for (requirements) specification than the *VDM*, which was better suited for design. I believe the proposal was more political than scientific in nature. Requirements statements, *qua* restricted natural language forms, may be represented by parse trees, which in turn, though generated by recursive rules, are restricted forms of cycle-free directed graphs and the *prototype* concept, in the classical sense of that term, which I have chosen for such graphs, because of its immediate conceptual value, is the bill of material, to which I now return.

# DISCOVERING SPECIFICATIONS

## 4. Bill of Materials

Recall from Chapter 5 that a bill of material, at the most abstract level, may be specified by the domain model:

$$BOM_0 = Pn \xrightarrow[m]{} \mathcal{P}Pn$$

subject to an invariant which shall now be presented. There are two parts to the invariant: that all parts are recorded, *allrec*, and that there are no cycles, *nonrec*. The first (part of the) invariant has already been expressed in the operator calculus as

$$allrec(\gamma) \triangleq \cup / (rng \gamma) \subset dom \gamma$$

To deal with the second (part of the) invariant, the appropriate starting point is the generalised parts-explosion algorithm:

$$\begin{aligned} \mathcal{E}_n[\theta](\gamma, \tau) &\triangleq (\gamma, \tau) \\ \mathcal{E}_n[[p \mapsto m] \cup t](\gamma, \tau) &\triangleq \mathcal{E}_n[t] \circ \mathcal{E}_{m \times n}[\gamma(p)](\gamma, \tau \oplus n \otimes [p \mapsto m]) \end{aligned}$$

which was interpreted as a directed acyclic graph traversal algorithm with ‘note taking’. Let us translate the bill of material directly into the terminology of graphs:

$$GRAPH = NODE \xrightarrow[m]{} \mathcal{P}NODE$$

and, in place of the table, I will introduce the concept of a marking:

$$MARKING = NODE \xrightarrow[m]{} \{0, 1\}$$

In doing the graph traversal, we will want to mark all nodes initially as ‘unvisited’. Then as we proceed through the graph, each node which has not been visited will be marked as ‘visited’. The corresponding algorithm, which is a reuse of the parts-explosion algorithm, is simply:

$$\begin{aligned} \mathcal{E}: NODE &\longrightarrow GRAPH \longrightarrow GRAPH \times MARKING \\ \mathcal{E}[b]\gamma &\triangleq \\ \text{let } \mu &= [n_j \mapsto 0 \mid n_j \in dom \gamma] \text{ in} \\ &\mathcal{E}[\gamma(n)](\gamma, \mu + [n \mapsto 1]) \end{aligned}$$

where  $\mathcal{E}[\gamma(n)](\gamma, \mu + [n \mapsto 1])$  is an invocation of the tail-recursive form

$$\begin{aligned}
\mathcal{E}: \mathcal{PNODE} &\longrightarrow (\text{GRAPH} \times \text{MARKING}) \longrightarrow (\text{GRAPH} \times \text{MARKING}) \\
\mathcal{E}[\emptyset](\gamma, \mu) &\triangleq (\gamma, \mu) \\
\mathcal{E}[\{n_j\} \uplus S](\gamma, \mu) &\triangleq \\
\mu(n_j) &= 0 \\
&\rightarrow \mathcal{E}[S] \circ \mathcal{E}[\gamma(n_j)](\gamma, \mu + [n_j \mapsto 1]) \\
&\rightarrow \mathcal{E}[S](\gamma, \mu)
\end{aligned}$$

If  $(\gamma, \mu) = \mathcal{E}[n]\gamma$ , then  $\mu^{-1}(1)$  is the set of nodes reachable from  $n$ . I have chosen to introduce a specific marking structure to signal the design possibility of using a bit in the node itself for the purposes of indicating whether or not a node has been visited. I could have used a more abstract specification based on sets which would simply record the name of the node visited. Let us use this algorithm to specify the second part of the invariant, *nonerec*. If indeed the bill of material is well-formed, then clearly we will obtain the desired result. Now suppose that there are cycles. Let  $pn$  denote the part name for the product and replace the initialisation,  $\mathcal{E}[\gamma(n)](\gamma, \mu + [n \mapsto 1])$ , with  $\mathcal{E}[\gamma(pn)](\gamma, \mu)$ , where I indicate that only constituent parts are to be marked as ‘visited’. Then  $\mathcal{E}[pn]\gamma$  will give the the pair  $(\gamma, \mu)$  and  $\mu^{-1}(1)$  is the set of part names visited. Consequently  $pn \notin \mu^{-1}(1)$  is the desired condition, or is it? There is another subtle point yet to be addressed, one which will come to light once we study the basic operations that one may perform on a bill of material.

#### 4.1. The Operations

The operations that one may perform on a bill of material modelled by  $\gamma \in BOM_0$  are adopted from (Bjørner 1988, 457–8). In presenting this material, I wish to point out some hidden assumptions that underly the specifications. The operations themselves may be taken to be requirements specifications in the sense of the material in the earlier part of this Chapter and I propose to demonstrate what is involved in analysing requirements.

## DISCOVERING SPECIFICATIONS

### 4.1.1. The Create Operation

The only initial comment that I need make here is that it corresponds exactly to its counterpart in  $DICT_6$ .

$$Create_0: \longrightarrow (Pn \xrightarrow{m} \mathcal{P}Pn)$$

$$Create_0 \triangleq \theta$$

An empty bill of material trivially satisfies the invariant.

### 4.1.2. The Enter Operation

This is used to extend a bill of material by entering a part name  $p$  and its corresponding set of constituent parts  $Q$ :

$$Enter_0: (Pn \times \mathcal{P}Pn) \longrightarrow (Pn \xrightarrow{m} \mathcal{P}Pn) \longrightarrow (Pn \xrightarrow{m} \mathcal{P}Pn)$$

$$Enter_0[[p, Q]]\gamma \triangleq \gamma \cup [p \mapsto Q]$$

subject to the pre-condition  $(p \notin dom \gamma) \wedge Q \subset dom \gamma$ . In the original text, the second part of the pre-condition read,  $Q \subseteq dom \gamma$ , thus repeating the error in the original *allrec* invariant. Having made this observation, it is now also immediately clear that the real use (or meaning) of the enter operation is to build a bill of material from ‘the ground up’. Consequently, we infer that from a conceptual model point of view, to use this operation one must know in advance the structure of the bill of material that one wishes to build and one builds upwards from basic parts to subassemblies.

One might suppose that one could use the enter command to build up a subassembly in stages. For example, assuming that the pre-condition is true, then  $Enter_0[[p, \emptyset]]\gamma$  extends the existing bill of material with  $[p \mapsto \emptyset]$ . A standard interpretation from the conceptual model point of view is that  $p$  is a basic part. Now let us suppose for the sake of argument that we really want to consider  $p$  as a subassembly and that we intend to build it up in stages. The only way in which this can be done is to use the add command whose specification is given below. However, the pre-condition of the  $Add_0$  command,  $\gamma(p) \neq \emptyset$ , specifically rules out this interpretation. The only way in which we can use the enter command to build a subassembly in stages is to give at least one constituent part. Contrast this interpretation with its analogue  $Ent_6$  in the  $DICT_6$  model.

These observations lead to the ‘surprising discovery’ that a bill of material,  $\gamma \in BOM_0$ , may consist of a set of basic parts where there is no structural relationship between those parts, or it may consist of a set of subassemblies which are conceptually independent, or, ultimately, it may consist of a set of products. Moreover, mixtures of all these types are admissible! In other words, there is an immense richness of structures implied by the domain equation

$$BOM_0 = Pn \xrightarrow{m} \mathcal{P}Pn$$

subject to the original invariant of the text. Now, referring to my operator calculus form of the *nonerec* part of the invariant, the reason that I raised the query with respect to its validity is clear—it is possible to construct a  $\gamma$  such that for parts  $p_j, p_k \in \gamma$ , the restricted bill of material

$$((\pi_2 \circ \mathcal{E}[[p_j]]\gamma)^{-1}(1) \cup \{p_j\}) \triangleleft \gamma$$

satisfies the invariant and  $((\pi_2 \circ \mathcal{E}[[p_k]]\gamma)^{-1}(1) \cup \{p_k\}) \triangleleft \gamma$  does not! Of course, the operator form of *nonerec* is as valid as the original ‘fixed-point’ recursive form of Bjørner. The significant point to note is that the original domain model for a bill of material needs to be extended, the most obvious extension being a dictionary or database of part names and their properties which is beyond the scope of the present work. Let us now consider some proofs.

Given an invariant and the specification of the enter command, we now have problems to solve in the sense of Pólya. Specifically, we must verify that both parts of the invariant are conserved under the enter operation. One immediately notes that we must strengthen the pre-condition by adding the obviously necessary clause,  $p \notin Q$ , for without it we would invalidate the invariant—another oversight in the original text. Let us consider each problem in turn. First we look at the *allrec* invariant.

**HYPOTHESIS 6.1.**  $\cup/(rng \gamma) \subset dom \gamma, p \notin dom \gamma, Q \subset dom \gamma, p \notin Q.$

**CONCLUSION 6.1.**  $\cup/(rng(\gamma \cup [p \mapsto Q])) \subset dom(\gamma \cup [p \mapsto Q]).$

*Proof:* First we observe that  $\cup/(rng(\gamma \cup [p \mapsto Q])) \subseteq \cup/(rng \gamma) \cup Q$ . Since  $\cup/(rng \gamma) \subset dom \gamma$  and  $Q \subset dom \gamma$ , by hypothesis, it follows that

$$\cup/(rng(\gamma \cup [p \mapsto Q])) \subset dom(\gamma)$$

## DISCOVERING SPECIFICATIONS

But, on the other hand, since  $p \notin \text{dom } \gamma$ , then  $\text{dom } \gamma \subset \text{dom}(\gamma \cup [p \mapsto Q]) = \text{dom } \gamma \cup \{p\}$  and the conclusion follows.

For the second part of the invariant, it is sufficient to state the hypothesis in the form

HYPOTHESIS 6.2.  $\neg \chi[[p]]((\pi_2 \circ \mathcal{E}[[p]]\gamma)^{-1}(1)), p \notin \text{dom } \gamma, Q \subset \text{dom } \gamma, p \notin Q.$

and we are required to establish

CONCLUSION 6.2.  $\neg \chi[[p]]((\pi_2 \circ \mathcal{E}[[p]](\gamma \cup [p \mapsto Q]))^{-1}(1)).$

The proof is obvious. By hypothesis, for all  $q \in Q$ ,  $q$  does not give rise to any cycles and  $p$  was not recorded in the original  $\gamma$ , which was also cycle-free. Therefore, the marking algorithm starting with  $p$  can not give cycles.

A remark on the ‘style’ of the proof is in order. It is not formal in the sense of logic; it is, rather, a proof along classical mathematical lines. That I can so argue for correctness is due entirely to the operator form of the invariant.

### 4.1.3. The Delete Operation

The intention of the delete operation, as specified in the original text, is to undo the effect of the enter command:

$$\begin{aligned} \text{Delete}_0: Pn &\longrightarrow (Pn \xrightarrow{m} \mathcal{P}Pn) \longrightarrow (Pn \xrightarrow{m} \mathcal{P}Pn) \\ \text{Delete}_0[[p]]\gamma &\triangleq \{p\} \leftarrow \gamma \end{aligned}$$

subject to Bjørner’s pre-condition  $p \in \text{dom } \gamma \wedge \text{inv-BOM}_0(\{p\} \leftarrow \gamma)$ . Let us prove the problem with respect to the *allrec* invariant.

HYPOTHESIS 6.3.  $\cup / (\text{rng } \gamma) \subset \text{dom } \gamma, p \notin \text{dom } \gamma.$

CONCLUSION 6.3.  $\cup / (\text{rng}(\{p\} \leftarrow \gamma)) \subset \text{dom}(\{p\} \leftarrow \gamma).$

*Proof:* The first observation is that Bjørner’s use of the *allrec* invariant as part of the pre-condition is *foul*. The very thing which we must prove is stated as proven. This sort of situation often arises in published VDM specifications with respect to invariants and pre-conditions and even more so with respect to retrieve functions. Without the aid of an operator calculus, proofs involving invariants and retrieve functions must appear tedious.

Let us attempt a proof beginning with the basic property of the *rng* operator with respect to the removal operator:

$$\cup/(rng(\{p\} \leftarrow \gamma)) \subseteq \cup/(rng \gamma \subset dom \gamma)$$

and since  $p \notin dom \gamma$ , by hypothesis, then  $dom(\{p\} \leftarrow \gamma) = dom \gamma \setminus \{p\}$ . But we can not complete the proof . . .! This leads us to a reconsideration of the nature of a bill of material  $\gamma \in BOM_0$  and its invariant in the light of the semantics of the delete operation. The result is surprising.

The delete operation is the exact inverse of the enter operation and thus works from the top down. Consequently, if one has a bill of material  $\gamma$  whose product name is  $p$ , then this must be the first part name to be deleted! After such a deletion, the resulting bill of material is effectively a set of bill of materials! Referring back to the enter operation, it then becomes clear that conceptually the enter operation is supposed to be used to join together independent subassemblies, each of which may be regarded as a bill of material in its own right. Thus, we have again ‘inadvertently discovered’ the intrinsic recursive substructure which is not immediately reflected in the domain equation  $BOM_0 = Pn \xrightarrow{m} \mathcal{P}Pn$  and such substructure is only exposed when we consider the specification of the operations. Alternatively, and more importantly, we have exposed a conceptual mismatch between the problem domain and the *VDM* model and the reason for our failure to carry through the proof with respect to the delete operation is only now too clear.

To continue with the existing model, we must strengthen the pre-condition to include a clause which stipulates that  $p$ , the part to be deleted, is not contained in any other part, i.e.,  $\{p\} \not\subseteq rng \gamma$  or, equivalently,  $p \notin \cup/(rng \gamma)$ . Hence, we have

$$\begin{aligned} \cup/(\{p\} \leftarrow \gamma) &\subseteq \cup/(rng \gamma) = \cup/(rng \gamma) \setminus \{p\} \\ &\subseteq dom \gamma \setminus \{p\} \\ &= dom(\{p\} \leftarrow \gamma) \end{aligned}$$

as required. It is not necessary to give the details of the proof for the *nonrec* part of the invariant—they are really too trivial.

## DISCOVERING SPECIFICATIONS

### 4.1.4. The Add Operation

I have already remarked above that this operation can only be used to extend an existing (sub)assembly. It can not be used to create or enter a subassembly. Only the enter operation is capable of doing that. Conceptually, the add operation is only applicable to substructure.

$$Add_0: (Pn \times Pn) \longrightarrow (Pn \xrightarrow{m} \mathcal{P}Pn) \longrightarrow (Pn \xrightarrow{m} \mathcal{P}Pn)$$

$$Add_0[[p, q]]\gamma \triangleq \gamma + [p \mapsto \gamma(p) \cup \{q\}]$$

subject to the pre-condition  $(p \in dom \gamma) \wedge (q \in dom \gamma) \wedge (\gamma(p) \neq \theta) \wedge (q \notin \gamma(p))$ . Bjørner explicitly uses the invariant  $inv\text{-}BOM_0(\gamma + [p \mapsto \gamma(p) \cup \{q\}])$  to guard the semantics of the add operation in the sense that it is valid if and only if “the resulting [bill of material] remains well-formed”. The proof that the add operator preserves the first part of the invariant causes no difficulty:

HYPOTHESIS 6.4.  $\cup/(rng \gamma) \subset dom \gamma, p \in dom \gamma, q \in dom \gamma, \gamma(p) \neq \theta, q \notin \gamma(p)$ .

CONCLUSION 6.4.  $\cup/(rng(\gamma + [p \mapsto \gamma(p) \cup \{q\}])) \subset dom(\gamma + [p \mapsto \gamma(p) \cup \{q\}])$ .

*Proof:* First, we have,  $\cup/(rng(\gamma + [p \mapsto \gamma(p) \cup \{q\}])) \subseteq \cup/(rng \gamma) \cup \{q\}$ . Also  $dom(\gamma + [p \mapsto \gamma(p) \cup \{q\}]) = dom \gamma$ . But, by hypothesis,  $q \in dom \gamma$ . Hence, the conclusion follows.

With respect to the second part of the invariant, it seems clear that nothing new is being added to the bill of material, i.e., only a rearrangement of structure is being performed. Therefore, the add operation can not violate the invariant. In fact, Bjørner’s use of the invariant as a guard implies that the add operation could conceivably violate the operation. I do not believe this to be the case and, therefore, the very inclusion of the guard is misleading.

### 4.1.5. The Erase Operation

The erase operation is intended to be the inverse of the preceding add operation and thus operates exclusively on structure:

$$Erase_0: (Pn \times Pn) \longrightarrow (Pn \xrightarrow{m} \mathcal{P}Pn) \longrightarrow (Pn \xrightarrow{m} \mathcal{P}Pn)$$

$$Erase_0[[p, q]]\gamma \triangleq \gamma + [p \mapsto \gamma(p) \setminus \{q\}]$$

subject to a pre-condition which is similar to that for the add operation:  $(\{p, q\} \subset dom \gamma) \wedge (q \in \gamma(p) \neq \theta)$ . Needless to remark, the erroneous use of  $\subseteq$  was employed

in the original text. The preservation of the *allrec* part of the invariant with respect to the erase command is readily established.

HYPOTHESIS 6.5.  $\cup / (\text{rng } \gamma) \subset \text{dom } \gamma, \{p, q\} \subset \text{dom } \gamma, \gamma(p) \neq \emptyset, q \in \gamma(p)$ .

CONCLUSION 6.5.  $\cup / (\text{rng}(\gamma + [p \mapsto \gamma(p) \setminus \{q\}])) \subset \text{dom}(\gamma + [p \mapsto \gamma(p) \setminus \{q\}])$ .

*Proof:* The proof is straightforward:

$$\begin{aligned} \cup / (\text{rng}(\gamma + [p \mapsto \gamma(p) \setminus \{q\}])) &\subseteq \cup / (\text{rng } \gamma) \\ &\subset \text{dom } \gamma \\ &= \text{dom}(\gamma + [p \mapsto \gamma(p) \setminus \{q\}]) \end{aligned}$$

It is not necessary to provide any details for the preservation of the *nonerec* part of the invariant.

#### 4.2. Bill of Material as Product

There are, of course, other operations that one might perform on a bill of material. In addition, the specifications need to be reworked when we consider the process of reification. However, I would like to focus on the issue of requirements specification. In other words, a bill of material is, in reality, the bill of material of some product, a product which has a particular realisation—consider the snow shovel. The part name for a product is a distinguished part name, different in nature conceptually from other part names. Thus, a feasible model for a product is simply given by:

$$\text{PRODUCT} :: Pn \times BOM_1$$

where the model of the bill of material is the reified model

$$\gamma \in BOM_1 = Pn \xrightarrow{m} P\_Rec$$

$$Pn = \dots$$

$$P\_Rec = Pn \xrightarrow{m} \mathbf{N}_1$$

and subject to some invariant(s). We will want to supply a retrieve function which maps a bill of material  $\gamma_1 \in BOM_1$  to the more abstract  $\gamma_0 \in BOM_0$ . This is readily obtained by considering the transformation

$$[p \mapsto [q_1 \mapsto n_1, \dots, q_k \mapsto n_k]] \implies [p \mapsto \{q_1, \dots, q_k\}]$$

which suggests iterating over the range of  $\gamma_1$  with the domain operator, *dom*. The required retrieve function is, therefore,

## DISCOVERING SPECIFICATIONS

$$\begin{aligned} \text{retr-}BOM_0: BOM_1 &\longrightarrow BOM_0 \\ \text{retr-}BOM_0(\gamma_0) &\triangleq (- \xrightarrow[m]{} \text{dom})\gamma_0 \end{aligned}$$

Now, looking at the invariants for the product, it must be the case that the bill of material part satisfies both the *allrec* and the *nonerec* invariants:

$$\begin{aligned} \text{inv}_1\text{-}PRODUCT: PRODUCT &\longrightarrow \mathbf{B} \\ \text{inv}_1\text{-}PRODUCT(\text{mk-}PRODUCT(p, \gamma)) &\triangleq \\ \text{let } \gamma_0 = \text{retr-}BOM_0(\gamma) \text{ in} & \\ \cup / (\text{rng } \gamma_0) \subset \text{dom } \gamma_0 & \end{aligned}$$

and

$$\begin{aligned} \text{inv}_2\text{-}PRODUCT: PRODUCT &\longrightarrow \mathbf{B} \\ \text{inv}_2\text{-}PRODUCT(\text{mk-}PRODUCT(p, \gamma)) &\triangleq \\ \text{let } \gamma_0 = \text{retr-}BOM_0(\gamma) \text{ in} & \\ (\forall q \in \text{dom } \gamma_0)(\neg \chi \llbracket q \rrbracket ((\pi_2 \circ \mathcal{E} \llbracket q \rrbracket \gamma_0)^{-1}(1))) & \end{aligned}$$

where I have retained the use of the  $\forall$  logical operator for clarity. But from the previous section, we know that this is not sufficient. The bill of material  $\gamma_1 \in BOM_1$  *must* denote a finished product, i.e., the domain of the bill of material must be exactly the set of parts reachable from the part name of the product:

$$\begin{aligned} \text{inv}_3\text{-}PRODUCT: PRODUCT &\longrightarrow \mathbf{B} \\ \text{inv}_3\text{-}PRODUCT(\text{mk-}PRODUCT(p, \gamma)) &\triangleq \\ \text{let } \gamma_0 = \text{retr-}BOM_0(\gamma) \text{ in} & \\ \text{let } S = (\pi_2 \circ \mathcal{E} \llbracket p \rrbracket \gamma_0)^{-1}(1) \text{ in} & \\ S = \text{dom } \gamma_0 & \end{aligned}$$

This is an additional constraint on the domain  $BOM_1$  which distinguishes a ‘bill of material as product’ from a ‘bill of material in the making’. Clearly, we now have two distinct views of a bill of material at the conceptual level. Although the same domain equation may be used for both, it is the use of the invariant that provides the *differentia*. Moreover, it is clear that the operations given in the previous section are inapplicable to the bill of material as product. A useful exercise is the exploration of the applicability of said operations to *PRODUCT* in the presence of the third invariant and to note the conditions under which it is violated. This helps to reinforce the conceptual model of a bill of material in the making.

Real products are built out of real parts, some of which must be acquired from suppliers: nails, screws, bolts, etc., and others which are manufactured in-house.

The bill of material essentially gives just the structure of the product. There are other requirements issues and corresponding constraints to be addressed. In the remainder of the Chapter I will demonstrate just such issues. However, from the point of view of the Irish School of the *VDM*, it is important to relate known concepts and techniques from one problem domain to problem solving in other (frequently unrelated) domains. I have already shown the interconnections between the bill of material and the bag by re-presenting an existing parts-explosion algorithm, which in turn was modified to give a directed acyclic graph traversal algorithm. I would like to bring these ideas together in the presentation of a simple algorithm related to the problem of raising a number to a power.

#### 4.3. Addition Chains

I used the exponential function  $exp(x, n) = x^n$  in Chapter 4 to illustrate the concept of tail-recursion and remarked that there was an interesting body of material of both a theoretical as well as of a practical nature in (Knuth 1981, 2:441 *et seq.*). The  $exp$  function is a homomorphism from natural numbers under multiplication to natural numbers under addition. Instead of studying powers directly, one may focus on their indices. Thus, the product  $x^{11}$  may be expressed in the form

$$\begin{aligned} x^{11} &= x^1 \cdot x^{10} \\ &= x^1 \cdot (x^5)^2 \\ &= x^1 \cdot (x^2 \cdot x^3)^2 \\ &= x^1 \cdot (x^2 \cdot (x^1 \cdot x^2))^2 \end{aligned}$$

which demonstrates how  $x^{11}$  may be computed from  $x^1$  by squaring and multiplication. This computational schema may be organised as an addition chain which is defined by

DEFINITION 6.2. *An addition chain for  $n \in \mathbf{N}_1$  is a sequence of integers*

$$1 = a_0, a_1, a_2, \dots, a_r = n$$

*with the property that  $a_i = a_j + a_k$ , for some  $k \leq j < i$ .*

## DISCOVERING SPECIFICATIONS

For example,  $n = 11$  gives rise to the sequence

$$\begin{aligned} a_0 &= 1 \\ a_1 &= a_0 + a_0 = 2 \\ a_2 &= a_1 + a_0 = 3 \\ a_3 &= a_2 + a_1 = 5 \\ a_4 &= a_3 + a_2 = 10 \\ a_5 &= a_4 + a_0 = 11 \end{aligned}$$

It may be assumed, without loss of generality, that an addition chain is monotonically increasing:

$$1 = a_0 < a_1 < a_2 < \dots < a_r = n$$

Knuth introduces the terms ‘doubling step’ and ‘star step’ defined in the following manner. Let  $a_i = a_j + a_k$  denote step  $i$  of an addition chain. Then step  $i$  is a doubling if  $j = k = i - 1$  and a star step if  $j = i - 1$ . The addition chain for  $n = 11$  involves only doublings and star steps.

Every addition chain can be represented by a directed acyclic graph (DAG). I would now like to present an algorithm that converts an addition chain, represented as a non-empty sequence over an alphabet  $\Sigma$ , into a directed acyclic graph. I choose  $\Sigma$  rather than  $\mathbf{N}_1$  to bring out the correspondence between the structures to be introduced and those that we have already seen before. Consider the addition chain for  $n = 11$ . It will have the representation

$$\langle p_1, p_2, p_3, p_5, p_{10}, p_{11} \rangle$$

where, in place of a number  $k$ , I have chosen to write  $p_k$  to suggest a part name.

The corresponding DAG will have the form

$$\begin{aligned} [p_1 \mapsto [p_2 \mapsto 2, p_3 \mapsto 1, p_{11} \mapsto 1], \\ p_2 \mapsto [p_3 \mapsto 1, p_5 \mapsto 1], \\ p_3 \mapsto [p_5 \mapsto 1], \\ p_5 \mapsto [p_{10} \mapsto 2], \\ p_{10} \mapsto [p_{11} \mapsto 1], \\ p_{11} \mapsto \theta] \end{aligned}$$

which is just a bill of material! Consider the entry  $[p_1 \mapsto [p_2 \mapsto 2, p_3 \mapsto 1, p_{11} \mapsto 1]]$ . This is interpreted to read that the ‘product’  $p_{11} \equiv x^{11}$ , and the subassemblies

$p_2 \equiv x^2$  and  $p_3 \equiv x^3$ , are computed using  $p_1 \equiv x^1$ . The ‘numbers of parts’ denote doublings or star steps. I now introduce the domain equations

$$\begin{aligned}\sigma &\in ADD\_CHAIN = \Sigma^+ \\ \Sigma &= \mathbf{N}_1 \\ \gamma &\in DAG = \Sigma \xrightarrow{m} (\Sigma \xrightarrow{m} \mathbf{N}_1)\end{aligned}$$

and ignore the invariant that ought to constrain the latter. Then, the transformation,  $T$ , from an addition chain to a DAG may be expressed simply by

$$\begin{aligned}T: \Sigma^+ &\longrightarrow (\Sigma \xrightarrow{m} (\Sigma \xrightarrow{m} \mathbf{N}_1)) \\ T(\sigma) &\triangleq T[\sigma]\theta\end{aligned}$$

where the tail-recursive form is

$$\begin{aligned}T: \Sigma^+ &\longrightarrow (\Sigma \xrightarrow{m} (\Sigma \xrightarrow{m} \mathbf{N}_1)) \longrightarrow (\Sigma \xrightarrow{m} (\Sigma \xrightarrow{m} \mathbf{N}_1)) \\ T[\langle a, b \rangle \wedge \sigma]\gamma &\triangleq \\ &T[\langle b \rangle \wedge \sigma](\gamma \cup [a \mapsto [b \mapsto 1]] + [(b - a) \mapsto \gamma(b - a) \oplus [b \mapsto 1]]) \\ T[\langle e \rangle]\gamma &\triangleq \gamma \cup [e \mapsto \theta]\end{aligned}$$

Note that for a doubling,  $b = 2a$ , we have

$$T[\langle a, b \rangle \wedge \sigma]\gamma = T[\langle b \rangle \wedge \sigma](\gamma \cup [a \mapsto [b \mapsto 2]])$$

In presenting the algorithm I have freely made use of the fact that the ‘part record’ of the ‘bill of material’ is a bag (see Chapter 5) and used the appropriate bag operator of addition,  $\oplus$ . An alternative version of the algorithm may be obtained by initialising the required DAG to give

$$T(\sigma) \triangleq \text{let } \gamma = [a_i \mapsto \theta \mid a_i \in \text{elems } \sigma] \text{ in } T[\sigma]\gamma$$

The tail-recursive part then takes the form:

$$\begin{aligned}T[\langle a, b \rangle \wedge \sigma]\gamma &\triangleq \\ &T[\langle b \rangle \wedge \sigma](\gamma + [a \mapsto \gamma(a) \oplus [b \mapsto 1]] + [b - a \mapsto \gamma(b - a) \oplus [b \mapsto 1]])\end{aligned}$$

and in the case of a doubling

$$T[\langle a, b \rangle \wedge \sigma]\gamma = T[\langle b \rangle \wedge \sigma](\gamma + [a \mapsto \gamma(a) \oplus [b \mapsto 2]])$$

The override is essential due to the initialization  $\gamma = [a_i \mapsto \theta \mid a_i \in \text{elems } \sigma]$ .

## DISCOVERING SPECIFICATIONS

If one sketches the resulting DAG of the addition chain of  $n$  one notes immediately that it corresponds to an inverted ‘bill of material’ where the ‘product’ is  $n$ . A DAG which gives the conventional ‘product’ structure is readily devised:

$$T: \Sigma^+ \longrightarrow (\Sigma \xrightarrow{m} (\Sigma \xrightarrow{m} \mathbf{N}_1))$$

$$T(\sigma) \triangleq \text{let } a = \text{hd } \sigma \text{ in } T[\sigma][a \mapsto \theta]$$

The first element in the addition chain  $\sigma$  is a ‘basic part’, the only one! The tail-recursive form is

$$T: \Sigma^+ \longrightarrow (\Sigma \xrightarrow{m} (\Sigma \xrightarrow{m} \mathbf{N}_1)) \longrightarrow (\Sigma \xrightarrow{m} (\Sigma \xrightarrow{m} \mathbf{N}_1))$$

$$T[\langle a, b \rangle \wedge \sigma] \gamma \triangleq$$

$$T[\langle b \rangle \wedge \sigma](\gamma \cup [b \mapsto [a \mapsto 1] \oplus [(b - a) \mapsto 1]])$$

$$T[\langle e \rangle] \gamma \triangleq \gamma$$

where every ‘subassembly’  $p_b$  is composed of  $p_a$  and  $p_{(b-a)}$ . This observation leads us to ask what the relationship there may be between an addition chain and ‘standard’ bill of material operations, such as create, enter, etc. Consider the example  $\langle 1, 2, 3, 5, 10, 11 \rangle$  once again. Recording the composition of  $p_5$  with respect to some  $\gamma$  gives

$$\gamma \cup [p_5 \mapsto [p_3 \mapsto 1] \oplus [p_2 \mapsto 1]]$$

where both  $p_3$  and  $p_2$  have already been recorded in  $\gamma$ . Although I did not give the specification for the  $Enter_1$  command on the  $BOM_1$  model, it is clear that we may denote the recording of  $p_5$  by

$$Enter_1[p_5, [p_3 \mapsto 1] \oplus [p_2 \mapsto 1]] \gamma$$

Consequently, we may re-present the algorithm that constructs the ‘product’ structure of an integer  $n$  in the alternative form

$$T(\sigma) \triangleq$$

$$\text{let } a = \text{hd } \sigma \text{ in}$$

$$\text{let } \gamma = Create_1 \text{ in}$$

$$T[\sigma] \circ Enter_1[p_a, \theta] \gamma$$

$$T[\langle a, b \rangle \wedge \sigma] \gamma \triangleq$$

$$T[\langle b \rangle \wedge \sigma] \circ Enter_1[p_b, [p_a \mapsto 1] \oplus [p_{(b-a)} \mapsto 1]] \gamma$$

$$T[\langle e \rangle] \gamma \triangleq \gamma$$

With respect to a bill of material model, an addition chain is just the trace of a sequence of  $Enter_1$  commands. Let us look at just one more aspect of the addition chain.

Every addition chain may be represented by a reduced DAG “that contains one ‘source’ vertex (labelled 1) and one ‘sink’ vertex (labelled n); every vertex but the source has in-degree  $\geq 2$  and every vertex but the sink has out-degree  $\geq 2$ ”, and two addition chains are *equivalent* if they have the same reduced graph (Knuth 1981, 2:461). For example, the addition chain  $\langle 1, 2, 3, 5, 10, 11 \rangle$  has the reduced form

$$\begin{aligned} & [p_1 \mapsto [p_2 \mapsto 2, p_5 \mapsto 1, p_{11} \mapsto 1], \\ & \quad p_2 \mapsto [p_5 \mapsto 2], \\ & \quad p_5 \mapsto [p_{11} \mapsto 2], \\ & \quad p_{11} \mapsto \theta] \end{aligned}$$

The reduction rule is “delete any vertex whose out-degree is 1 and attach the arcs from its predecessors to its successor”. Our goal is to formalise this specification of a reduced DAG.

Basically, one considers those elements  $a_j \mapsto [a_i \mapsto 1]$  and eliminates them. Let  $a_k \mapsto [\dots, a_j \mapsto v, \dots]$  be a predecessor of  $a_j$ . Then the  $v$  arcs from  $a_k$  to  $a_j$  must be attached to  $a_i$ . But one must also consider the possibility that  $a_k$  may already be a predecessor of  $a_i$ . In developing the specification, I would sketch a ‘computational sequence’ of the form

$$\begin{aligned} \gamma(a_k) &= [\dots, a_j \mapsto v, \dots] \\ (\gamma(a_k))(a_j) &= v \\ \{a_j\} \Leftarrow \gamma(a_k) &= [\dots] \neq \theta \\ (\{a_j\} \Leftarrow \gamma(a_k)) \oplus [a_i \mapsto v] &= [\dots, a_i \mapsto v', \dots] \end{aligned}$$

in order to match formal *Meta-IV* map operations with the natural language concepts of the original ‘specification’. Without worrying about locating either the  $a_j$  to be deleted or the  $a_k$  to be modified, I proceed to specify an algorithm (*cut&paste*) that does the appropriate ‘surgery’ on a DAG, which I denote by  $\gamma_a$ , to distinguish it from that which gives ‘product’ structure, denoted  $\gamma_b$ . Given  $a_j$ , where  $\gamma_a(a_j) = [a_i \mapsto 1]$ , is the ‘part’ to be deleted, and  $a_k$ , where  $(\gamma_a(a_k))(a_j) = v$ , is the part to be updated, then the surgical operation is

## DISCOVERING SPECIFICATIONS

$$\begin{aligned}
& cut\&paste[[a_j, a_k]]\gamma_a \triangleq \\
& \text{let } \{a_i\} = dom \gamma_a(a_j) \text{ in} \\
& \text{let } \gamma'_a = \{a_j\} \Leftarrow \gamma_a \text{ in} \quad \text{-- cut!} \\
& \text{let } \mu = \gamma'_a(a_k) \text{ in} \\
& \text{let } v = \mu(a_j) \text{ in} \\
& \text{let } \mu' = \{a_j\} \Leftarrow \mu \text{ in} \quad \text{-- I prefer to use the bag operation } \mu \ominus [a_j \mapsto v] \\
& \text{let } \mu'' = \mu' \oplus [a_i \mapsto v] \text{ in} \quad \text{-- paste!} \\
& \gamma'_a + [a_k \mapsto \mu'']
\end{aligned}$$

Note that I have written out the specification in great detail in order that I may check it step by step. Once I am satisfied that I have expressed precisely what I want, I then eliminate many of the ‘let clauses’ by back substitution to give

$$\begin{aligned}
& cut\&paste[[a_j, a_k]]\gamma_a \triangleq \\
& \text{let } \{a_i\} = dom \gamma_a(a_j) \text{ in} \\
& \text{let } \gamma'_a = \{a_j\} \Leftarrow \gamma_a \text{ in} \quad \text{-- cut!} \\
& \gamma'_a + [a_k \mapsto \gamma'_a(a_k) \ominus [a_j \mapsto (\gamma'_a(a_k))(a_j)] \oplus [a_i \mapsto (\gamma'_a(a_k))(a_j)]]
\end{aligned}$$

Referring to the addition chain for 11, it is clear that both vertices 10 and 3 should be cut. In cutting vertex 10, one only needs to paste in the appropriate arcs for vertex 5. But in cutting vertex 3, both vertices 1 and 2 are affected. Consequently, the *cut&paste* algorithm should be extended to cater for a set of predecessors:

$$\begin{aligned}
& cut\&paste[[a_j, S]]\gamma_a \triangleq \\
& S = \emptyset \\
& \rightarrow \gamma_a \\
& \rightarrow \text{let } a_k \in S \text{ in } cut\&paste[[a_j, S - \{a_k\}]] \circ cut\&paste[[a_j, a_k]]\gamma_a
\end{aligned}$$

Now, all that needs to be done is to locate all the appropriate  $(a_j, a_k)$  pairs in  $\gamma_a$ . This is surely to be accomplished by traversing the ‘product’ structure DAG,  $\gamma_b$ , which is analogous to the parts-explosion algorithm for a bill of material! Using the latter algorithm as a template, I produce the following ‘reduction’ algorithm

$$\begin{aligned}
 & \text{reduce}(n, \gamma_b, \gamma_a) \stackrel{\Delta}{=} \text{reduce}[\gamma_b(n)](\gamma_b, \gamma_a) \\
 & \text{reduce}[[m \mapsto v] \cup \mu](\gamma_b, \gamma_a) \stackrel{\Delta}{=} \\
 & \quad |\gamma_a(m)| = 1 \\
 & \quad - - \text{I am using the bag size operator} \\
 & \quad - - \text{to determine if vertex } m \text{ should be cut} \\
 & \quad \rightarrow \text{let } S = \text{dom } \gamma_b(m) \text{ in} \\
 & \quad \quad \text{reduce}[\mu] \circ \text{reduce}[\gamma_b(m)](\gamma_b, \text{cut\&paste}[m, S]\gamma_a) \\
 & \quad \rightarrow \text{reduce}[\mu] \circ \text{reduce}[\gamma_b(m)](\gamma_g, \gamma_a) \\
 & \text{reduce}[\theta](\gamma_b, \gamma_a) \stackrel{\Delta}{=} (\gamma_b, \gamma_a)
 \end{aligned}$$

Note how the ‘product’ structure,  $\gamma_b$ , is used in read-only mode and modifications are made to  $\gamma_a$  in much the same way as was the case for the table,  $\tau$ , in the parts-explosion algorithm for the bill of material.

Upon reflection, it is clear that the notion of ‘product’ structure plays a central rôle in developing specifications such as that of the graph reduction algorithm. Of course, one might argue that there is nothing new in this—graph theory is well-established. However, I hasten to point out the difference between a theory and the use of a particular notation. Whereas I have argued that the relationship between formal specifications and algebra needs to be strengthened, in particular with a view to discovery of theorems and their proofs, it is also crucial to keep in mind that, from a practical point of view, the discovery of formal specifications takes place essentially within a conceptual framework of applying mathematics. Returning to the real world of manufacturing, I will now conclude this Chapter by demonstrating how ‘product’ structure is used to compute the gross-to-net requirements for a product in a material requirements planning system.

## DISCOVERING SPECIFICATIONS

### 5. Material Requirements Planning

The material in this section on material requirements planning (MRP) is primarily based on *Manufacturing and Planning Control Systems* by Vollmann *et al.* (1984). Of necessity, I can give but a brief overview of the problem domain in order to provide a setting for the specification to follow. In a manufacturing facility, products are made to satisfy consumer demand. Such demand is either known in advance through fixed customer orders or forecasted according to some model. The total demand may be expressed in terms of units of product over a specific time period. Consider the snow shovel example, introduced in Chapter 5. In planning a production schedule, one must know, in advance, the demand for snow shovels. Such a demand is the top level gross requirements,  $\rho \in GROSS\_REQ$ . Since the top handle assembly is an immediate constituent component of the snow shovel, then it too is subject to the same gross requirements. Consider the MRP record for such an assembly:

13122 TOP HANDLE ASSEMBLY		1	2	3	4	5	6	7	8	9	10
Gross requirements			20		10		20	5		35	10
Scheduled receipts											
Projected available balance	25	25	5	5	0	0	0	0	0	0	0
Planned order releases			5		20	5		35	10		

Lead time = 2

The demand is expressed as 20 units in week 2, 20 units in week 4, etc., where it is assumed that the *planning horizon* is 10 weeks. In the second line of the MRP record, the number of units scheduled to be received by the manufacturing facility from some other source is recorded according to the same convention. ‘Scheduled receipts’ is denoted by  $\sigma \in SCHED\_REC$ . In this example, it is clearly assumed that all top handle assemblies are manufactured in-house. For each part, it is customary to provide some sort of buffering mechanism, the inventory, that ensures that adequate quantities are available to meet demand. Should production outstrip consumption, then the inventory increases adding to the cost of the product. On the other hand, if we can not produce enough to meet demand then customer orders may be lost. The projected available balance,  $\beta \in PA\_BALANCE$ , is the predicted level of inventory based on the demand (gross requirements) less any scheduled receipts. To ensure

that the inventory never falls below a certain level, 0 in this case, it is necessary to issue release orders,  $\varrho \in PO\_RELEASE$ , such that the constituent components of the top handle assembly may be available in time. Such constituent components have a ‘time-to-make’ or ‘time-to-deliver’ delay factor built in. This is indicated as the *lead time*, denoted  $d \in LEAD\_TIME$ .

The first, and most obvious, algorithm that must be specified is that which computes the projected available balance and the planned order releases for each part. Let us introduce some intuitive domain models. An MRP record, *MRP\_REC*, is essentially a tree:

$$MRP\_REC :: GROSS\_REQ \times SCHED\_REC \times PA\_BALANCE \times PO\_RELEASE \\ \times LOT\_SIZE \times SAFETY\_STOCK \times LEAD\_TIME$$

where in addition to the concepts already introduced there are the two additional domains: *lot size*,  $l \in LOT\_SIZE$ , and *safety stock*,  $s \in SAFETY\_STOCK$ . Some parts, for example, nails and bolts, are supplied in sizes of multiple units. Nails might be supplied in lots of 100. There may be a policy in the manufacturing facility to maintain a basic level of inventory, the *safety stock* level, as a hedge against interruptions in the production process.

For simplicity, I have chosen to model the main concepts using the *Meta-IV* sequence domain:

$$\begin{aligned} \rho &\in GROSS\_REQ = \mathbf{N}^* \\ \sigma &\in SCHED\_REC = \mathbf{N}^* \\ \beta &\in PA\_BALANCE = \mathbf{N}^* \\ \varrho &\in PO\_RELEASE = \mathbf{N}^* \end{aligned}$$

without bothering to be precise at this stage, for example, to give the corresponding domain invariants. One may always return later to the semantic domains to constrain the models. Similarly, the other domains may be specified loosely by

$$\begin{aligned} d &\in LEAD\_TIME = \mathbf{N} \\ l &\in LOT\_SIZE = \mathbf{N}_1 \\ s &\in SAFETY\_STOCK = \mathbf{N} \end{aligned}$$

## DISCOVERING SPECIFICATIONS

Then the computation of the projected available balance and planned order releases,  $Upd$ , is given by

$$\begin{aligned}
 Upd: MRP\_REC &\longrightarrow MRP\_REC \\
 Upd(\mu) &= \\
 &\text{let } mk\text{-}MRP\_REC(\rho, \sigma, \beta, \varrho, l, s, d) = \mu \text{ in} \\
 &\text{let } (l, s, \beta', \varrho') \triangleq Comp[\rho, \sigma](l, s, \beta, \varrho) \text{ in} \\
 &\text{let } \varrho'' = \text{shift}[d]\varrho' \text{ in} \\
 &\quad mk\text{-}MRP\_REC(\rho, \sigma, \beta, \varrho'', l, s, d)
 \end{aligned}$$

where the actual details are specified by the tail-recursive form  $Comp$  and then the resulting planned order releases *shifted* by the appropriate lead-time. The tail-recursive  $Comp$  algorithm is specified by

$$\begin{aligned}
 Comp[\langle r \rangle \wedge \rho, \langle s \rangle \wedge \sigma](l, s, \beta \wedge \langle b \rangle, \varrho) &\triangleq \\
 \text{let } b' = b + s - r \text{ in} \\
 &b' < s \\
 &\rightarrow \text{let } r = POR(l, s, b', 1) \text{ in} \\
 &\quad \text{let } b'' = b' + r \text{ in} \\
 &\quad\quad Comp[\rho, \sigma](l, s, \beta \wedge \langle b \rangle \wedge \langle b'' \rangle, \varrho \wedge \langle r \rangle) \\
 &\rightarrow Comp[\rho, \sigma](l, s, \beta \wedge \langle b \rangle \wedge \langle b' \rangle, \varrho \wedge \langle 0 \rangle)
 \end{aligned}$$

and the obvious termination is given by

$$Comp[\Lambda, \Lambda](l, s, \beta, \varrho) \triangleq (l, s, \beta, \varrho)$$

where again I have postponed some of the details of the specification—the exact nature of the computation of the planned order releases,  $POR$ . Focusing on the structure of the  $Comp$  algorithm, I would like to make some simple observations. Although specified recursively, it is clear that this is an iterative algorithm and the introduction of sequence slices will lead immediately to a conventional imperative form. The projected available balance at any time period is  $\beta \wedge \langle b \rangle$  and not simply  $\beta$ . This is due to the fact that the projected available balance is always ‘dated’ one time period earlier than the current time period. Where there are no planned order releases in a given time period, then these are indicated as orders of quantity zero. The computation of the planned order release is specified by

$$\begin{aligned}
 POR(l, s, b, n) &\triangleq \\
 \text{let } r = n \times l \text{ in} \\
 &(r + b \geq s \rightarrow r, POR(l, s, b, n + 1))
 \end{aligned}$$

## Material Requirements Planning

where I use a McCarthy conditional. The shift due to the actual lead time,  $d$ , is given by

$$\begin{aligned} \text{shift}[[d]](\langle r \rangle \wedge \varrho) &\triangleq \text{shift}[[d-1]]\varrho \\ \text{shift}[[0]]\varrho &\triangleq \varrho \end{aligned}$$

Each part has an associated MRP record which must be updated in a similar way. Whereas the gross requirements for a snow shovel are used to update the MRP record of immediate constituent parts, it is the planned order releases of the latter which are used as ‘gross requirements’ for subsequent updating of their constituent parts. For example, the top handle is a constituent part of the top handle assembly. Therefore, its MRP record must have the form

457 TOP HANDLE		1	2	3	4	5	6	7	8	9	10
Gross requirements			5		20	5		35	10		
Scheduled receipts				25							
Projected available balance	22	22	17	42	22	17	17	0	0	0	0
Planned order releases						18	10				

Lead time = 2

But 2 nails are also used in building the top handle assembly. Therefore, the MRP record for nails *as far as this assembly is concerned* has the form

082 NAIL (2 REQUIRED)		1	2	3	4	5	6	7	8	9	10
Gross requirements			10		40	10		70	20		
Scheduled receipts		50									
Projected available balance	4	54	44	44	4	44	44	24	4	4	4
Planned order releases					50		50				

Lead time = 1

Lot size = 50

Note the emphasis that this MRP record is only valid for this particular subassembly. Nails are also used in building the snow shovel itself and, therefore, from the perspective of the snow shovel, the MRP record for nails is different. In updating the MRP records for all parts in a product, one must use the product structure, the bill of material, to determine the exact requirements. In addition, note that the planned order releases of the top handle assembly have been multiplied by a factor of 2 in determining the gross requirements for the MRP record for nails. That factor is precisely given by the corresponding bill of material.

## DISCOVERING SPECIFICATIONS

In determining the real material requirements for a given product, one combines the parts-explosion algorithm of a bill of material with the updating of MRP records for each part. The algorithm, called ‘gross-to-net requirements explosion’, denoted GrossNet, is specified by

$$\begin{aligned} \text{GrossNet}: Pn &\longrightarrow (BOM_1 \times MRP\_FILE) \longrightarrow (BOM_1 \times MRP\_FILE) \\ \text{GrossNet}[[pn]](\gamma, \phi) &\triangleq \\ \text{let } \phi' = \text{Upd}_0[[pn]]\phi &\text{ in} \\ \text{let } mk\text{-MRP\_REC}(-, -, -, \varrho, -, -, -) = \phi'(pn) &\text{ in} \\ \mathcal{E}_\varrho[[\gamma(pn)]](\gamma, \phi') & \end{aligned}$$

The algorithm  $\mathcal{E}$  is essentially that of the bill of material parts-explosion. However, note that in place of the subscript 1, one has the planned order releases,  $\varrho$ , of the product  $pn$ . The clause  $\text{Upd}_0[[pn]]\phi$  denotes the computation of the projected available balance and the planned order releases of the product  $pn$ . The subscript 0 denotes **null** ‘extra’ gross requirements input for the product since these are already part of the MRP record in the file. Given the domain equation

$$\phi \in MRP\_FILE = Pn \xrightarrow{m} MRP\_REC$$

then the update operation may be specified by

$$\begin{aligned} \text{Upd}_\rho: Pn &\longrightarrow MRP\_FILE \longrightarrow MRP\_FILE \\ \text{Upd}_\rho[[pn]]\phi &\triangleq \\ \text{let } \mu = \phi(pn) &\text{ in} \\ \text{let } mk\text{-MRP\_REC}(\rho_0, -, -, -, -, -, -) = \mu &\text{ in} \\ \text{let } \mu' = mk\text{-MRP\_REC}(\rho_0 + \rho, -, -, -, -, -, -) &\text{ in} \\ \text{let } \mu'' = \text{Upd}(\mu') &\text{ in} \\ \phi + [pn \mapsto \mu''] & \end{aligned}$$

where  $\text{Upd}(\mu)$  is the update operation for the MRP record given earlier. The extra gross requirements input,  $\rho$ , is added to those already in the MRP record. The addition is, of course, vector addition. Note that the ‘usual’ pre-condition for file update applies. Having computed the planned order releases for the product, we are now in a position to continue processing all of the constituent parts:

$$\mathcal{E}_\rho: (Pn \xrightarrow{m} \mathbf{N}_1) \longrightarrow BOM_1 \times MRP\_FILE \longrightarrow BOM_1 \times MRP\_FILE$$

$$\begin{aligned} \mathcal{E}_\rho[[pn \mapsto m] \cup t](\gamma, \phi) &\triangleq \\ \text{let } \phi' = Upd_{m \times \rho}[[pn]]\phi &\text{ in} \\ \text{let } mk\text{-}MRP\_REC(-, -, -, \varrho, -, -, -) = \phi'(pn) &\text{ in} \\ \mathcal{E}_\rho[t] \circ \mathcal{E}_\varrho[[\gamma(pn)]](\gamma, \phi') & \end{aligned}$$

$$\mathcal{E}_\rho[[\theta]](\gamma, \phi) \triangleq (\gamma, \phi)$$

Note that in the ‘lateral traversal’ of the product structure, the subscript is the current gross requirements,  $\rho$ , whereas in the ‘downward traversal’, the subscript is the planned order releases,  $\varrho$ , i.e., the new gross requirements input for constituent parts which is computed from  $(m \times \rho) + \rho_0$ . Comparing this algorithm with the basic *all* parts-explosion algorithm, it is clear that they are almost structurally identical. The correspondence may be exhibited in an obvious manner by rewriting the tail-recursive part of the parts-explosion algorithm in the form

$$\begin{aligned} \mathcal{E}_n[[pn \mapsto m] \cup y](\gamma, \tau) &\triangleq \\ \text{let } \tau' = Upd_{m \times n}[[pn]]\tau &\text{ in} \\ \text{let } n' = \tau'(pn) &\text{ in} \\ \mathcal{E}_n[t] \circ \mathcal{E}_{m \times n}[[\gamma(pn)]](\gamma, \tau') & \end{aligned}$$

where  $Upd_v[[pn]]\tau = \tau \oplus [pn \mapsto v]$ . Indeed, it is this form that permits me to argue for the correctness of the gross-to-net requirements explosion algorithm.

But there is more to manufacturing than material requirements planning. The gross-to-net requirements explosion algorithm is a natural extension of the parts-explosion algorithm for a bill of material. Gross requirements arise from market demand (whether known customer demand or forecasted customer demand). At the other end of the ‘process’ there is the actual manufacture of product. The product consists of parts, many of which must be machined on a shop floor. The machining process itself, usually called a job, often involves passing a part through a *sequence* of multiple machines and the same *set* of machines is frequently used for different parts. Scheduling jobs on machines is a fundamental activity in manufacturing and it is to this that I now apply the *Meta-IV*.

# DISCOVERING SPECIFICATIONS

## 6. Scheduling

In dealing with requirements specification of systems which incorporate computing subsystems it is common to find a mix of computing-independent requirements and computing-dependent requirements. With respect to the latter, one may not ignore that body of knowledge which has been accumulated over the past fifty years, knowledge which is frequently encoded as program text.

“Indeed, I believe that virtually every important aspect of programming arises somewhere in the context of sorting or searching” (Knuth 1973, 3:v).

Searching and sorting are intimately related. However, in the case of formal specifications, one may frequently exhibit clearly the distinction between the two concepts. Before I delve into the problem domain of scheduling, I present some thoughts on the ‘standard’ example of the dictionary, details of which are in Appendix A. This may appear at first sight to be an unwarranted digression. Its relevance will become apparent.

In the case of the spelling-checker dictionary  $DICT_0 = \mathcal{P}WORD$ , looking up the spelling of a word  $w$  in the dictionary  $\delta$  is a trivial search problem at the specification level:

$$Lkp_0[[w]]\delta \triangleq \chi[[w]]\delta$$

Indeed, the concept of sorting is inappropriate here since there is no ordering concept for sets. Now, from a conceptual model point of view, looking up a word is the primary action that a user wants to perform with respect to such a dictionary and, consequently, primacy of place must be given to searching in this context. In the reification of  $DICT_0$  to the partitioned dictionary  $DICT_1 = (\mathcal{P}WORD)^*$ , there is still no concept of sorting. Rather, the concept of searching has been refined using the notion of hashing:

$$Lkp_1[[w]]\delta \triangleq \text{let } \delta = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \text{ in } \chi[[w]]\delta_{h(w)}$$

Even in the case of the reification of  $DICT_0$  to the dictionary as a sequence,  $DICT_2 = WORD^*$ , where one would expect to find a linear search, this was completely avoided in the specification

$$Lkp_2[[w]]\delta \triangleq \chi[[w]] \text{elems } \delta$$

by reverting back to the use of test for membership in a set. While still on the theme of looking up a word in a dictionary and the conceptual relation between ‘lookup’ and ‘search’, I would like to introduce a specific evolution of the spelling-checker dictionary, by incorporating the notion of partition based on the frequency of lookup requests. Essentially, I introduce the notion of a ‘fast dictionary’ containing the words whose spelling is checked most frequently and a ‘slow dictionary’. To illustrate the conceptual relationship between specification and problem domain, I have chosen to drop the name *DICT*, and replace it with the more abstract notion of state, denoted  $\Sigma$ . Although it is customary to use  $\Sigma$  to denote an alphabet, there should be no danger of confusion within the current context. Unfortunately, the number of ‘interesting’ Greek capital letters is small.

The fast dictionary will be called a *cache* and the slow dictionary will be given the name *memory*. Lest all correspondence be lost between the model and the domain of dictionaries, I have retained the name *WORD*. The corresponding domain equations for the problem in hand are, therefore,

$$\begin{aligned} \Sigma &:: \text{CACHE} \times \text{MEMORY} \\ \text{CACHE} &= \text{WORD} \xrightarrow{m} \text{FREQ} \quad \text{-- fast dictionary} \\ \text{MEMORY} &= \text{WORD} \xrightarrow{m} \text{FREQ} \quad \text{-- slow dictionary} \end{aligned}$$

where  $\text{FREQ} = \mathbf{N}_1$ . It should be clear from the above equations that the specification is basically about a cartesian product of bags. The lookup request will be interpreted in the following manner. First, check if the word  $w$  is in the cache,  $\chi[[w]]\kappa$ . If it is then return *true* and increment the reference count,  $\kappa \oplus [w \mapsto 1]$ . If the word is not in the cache, then check memory,  $\chi[[w]]\mu$ . If it is in the memory, then return *true*, increment its reference count and then consider the need to swop it into the cache. If the word is in neither the cache or memory, then enter it into memory and consider the need to swop it into the cache. This latter action will result in a swop only in the initial build-up of the system dictionary. This specification may be expressed formally by

## DISCOVERING SPECIFICATIONS

$$\begin{aligned}
Lkp: WORD &\longrightarrow \Sigma \longrightarrow \Sigma \times \mathbf{B} \\
Lkp[w]mk-\Sigma(\kappa, \mu) &\triangleq \\
&\chi[w]\kappa \\
&\quad \rightarrow (mk-\Sigma(\kappa \oplus [w \mapsto 1], \mu), true) \\
&\neg\chi[w]\kappa \wedge \chi[w]\mu \\
&\quad \rightarrow (Swop(mk-\Sigma(\kappa, \mu \oplus [w \mapsto 1])), true) \\
&\quad \rightarrow (Swop(mk-\Sigma(\kappa, \mu \oplus [w \mapsto 1])), false)
\end{aligned}$$

In the swopping algorithm, we will use the reference count to determine if a word in memory ought to be swopped into the cache. The principle is simple. First obtain those words in memory whose reference count is a maximum:

$$M = (\mu^{-1} \circ \uparrow / \circ rng \mu) \triangleleft \mu$$

and those words in the cache whose reference count is a minimum:

$$C = (\kappa^{-1} \circ \downarrow / \circ rng \kappa) \triangleleft \kappa$$

If the maximum reference count of the words in  $M$  is greater than the minimum reference count of the words in  $C$ , then a swop is performed:

$$\begin{aligned}
Swop: \Sigma &\longrightarrow \Sigma \\
Swop(mk-\Sigma(\kappa, \mu)) &\triangleq \\
\text{let } C &= (\kappa^{-1} \circ \downarrow / \circ rng \kappa) \triangleleft \kappa, \\
M &= (\mu^{-1} \circ \uparrow / \circ rng \mu) \triangleleft \mu \text{ in} \\
&mk-\Sigma(\kappa \ominus C \oplus M, \mu \ominus M \oplus C) \\
Swop(mk-\Sigma(\theta, \mu)) &\triangleq \\
\text{let } M &= (\mu^{-1} \circ \uparrow / \circ rng \mu) \triangleleft \mu \text{ in } mk-\Sigma(M, \mu \ominus M)
\end{aligned}$$

In practice, we would limit the size of the cache. The above specification permits the possibility of an unbounded cache. Now the interesting point in this example, from the point of view of searching, is that I have still retained the abstraction based on the characteristic function, i.e., there are no ‘search’ details given with respect to looking up a word. However, on the other hand, an explicit search mechanism has been used in the computation of the maximum and minimum reference counts in the memory and cache, respectively. Given this conceptual foundation, we are now in a position to examine what looks like, at first sight, a totally unrelated problem domain—that of job scheduling on the shop floor of a manufacturing facility.

## 6.1. Johnson's Scheduling Algorithm

Johnson's algorithm to minimise 'make span' in shop floor processing with two machines may be stated as follows (Vollmann *et al.*, 1984, 443):

1. Select the job with the minimum processing time on either machine 1 or machine 2. If this time is associated with machine 1, schedule this job first. If it is for machine 2, schedule this job last in the series of jobs to be run. Remove this job from further consideration.
2. Select the job with the next smallest processing time and process as above (if machine 1, schedule it second; if machine 2, next to last). Any ties can be broken randomly.
3. Continue this process until all of the jobs have been scheduled.

An alternative formulation may be found in (Eiselt and von Frajer 1977, 343) where the algorithm is given precisely in matrix algebraic terms, and uses the ubiquitous 'go to' construct. Although that of Vollmann *et al.*, given above may seem less precise, it is adequate for a formal specification.

Consideration of the problem leads to the following sequence of domain equations:

$$MACHINE\_LIST = JOB \xrightarrow{m} TIME$$

$$JOB = \dots$$

$$TIME = \mathbf{N}$$

$$SCHEDULE = JOB^*$$

Although I have given names to the domains of discourse, I do not actually use them in specifying the algorithms yet to be discussed. I prefer to use the corresponding unnamed structures. A machine list  $\mu$  is a set of ordered pairs  $(j_k, t_k)$  where job  $j_k$  requires  $t_k$  time units for processing on a given machine, represented as the map

$$\mu = [j_1 \mapsto t_1, j_2 \mapsto t_2, \dots, j_k \mapsto t_k, \dots, j_n \mapsto t_n]$$

A schedule  $s$  is the sequence of jobs  $j_k$  that are to be processed in a given order. Thus given the pair of machine lists for machines  $\mu_1$  and  $\mu_2$ :

$$\mu_1 = [j_1 \mapsto 5, j_2 \mapsto 8, j_3 \mapsto 4, j_4 \mapsto 5]$$

$$\mu_2 = [j_1 \mapsto 7, j_2 \mapsto 3, j_3 \mapsto 8, j_4 \mapsto 4]$$

we will construct the set of pairs  $J_1 = \{(j_1, t_1) \mid (j_1, t_1) \in \mu_1\}$ , where  $t_1$  is the minimum time for machine 1, and  $J_2 = \{(j_2, t_2) \mid (j_2, t_2) \in \mu_2\}$ , where  $t_2$  is the min-

## DISCOVERING SPECIFICATIONS

imum time for machine 2. Explicit operator calculus notation for such sets is,  $J_1 = (\mu_1^{-1} \circ \downarrow / \circ \text{rng } \mu_1) \triangleleft \mu_1$  and  $J_2 = (\mu_2^{-1} \circ \downarrow / \circ \text{rng } \mu_2) \triangleleft \mu_2$ . Thus, our first version of the algorithm is

$$\begin{aligned} \mathcal{J}_2: (JOB \xrightarrow[m]{} TIME)^2 &\longrightarrow JOB^* \\ \mathcal{J}_2(\mu_1, \mu_2) &\triangleq \\ \text{let } J_1 = (\mu_1^{-1} \circ \downarrow / \circ \text{rng } \mu_1) \triangleleft \mu_1, J_2 = (\mu_2^{-1} \circ \downarrow / \circ \text{rng } \mu_2) \triangleleft \mu_2 &\text{ in} \\ \text{let } (j_1, t_1) \in J_1, (j_2, t_2) \in J_2 &\text{ in} \\ t_1 \leq t_2 & \\ \rightarrow \langle j_1 \rangle \wedge \mathcal{J}_2(\{j_1\} \triangleleft \mu_1, \{j_1\} \triangleleft \mu_2) & \\ \rightarrow \mathcal{J}_2(\{j_2\} \triangleleft \mu_1, \{j_2\} \triangleleft \mu_2) \wedge \langle j_2 \rangle & \\ \mathcal{J}_2(\theta, \theta) &\triangleq \Lambda \end{aligned}$$

Since there may be multiple jobs with the same time in the set  $J = (\mu^{-1} \circ \downarrow / \circ \text{rng } \mu) \triangleleft \mu$ , then we may use more operator calculus notation:

$$\begin{aligned} \mathcal{J}_2(\mu_1, \mu_2) &\triangleq \\ \text{let } J_1 = (\mu_1^{-1} \circ \downarrow / \circ \text{rng } \mu_1) \triangleleft \mu_1, J_2 = (\mu_2^{-1} \circ \downarrow / \circ \text{rng } \mu_2) \triangleleft \mu_2 &\text{ in} \\ \text{let } (-, t_1) \in J_1, (-, t_2) \in J_2 &\text{ in} \\ t_1 \leq t_2 & \\ \rightarrow \wedge / \circ (\mathcal{P}j)J_1 \wedge \mathcal{J}_2(\text{dom } J_1 \triangleleft \mu_1, \text{dom } J_1 \triangleleft \mu_2) & \\ \rightarrow \mathcal{J}_2(\text{dom } J_2 \triangleleft \mu_1, \text{dom } J_2 \triangleleft \mu_2) \wedge \wedge / \circ (\mathcal{P}j)J_2 & \end{aligned}$$

where  $j$  is the usual injection operator for sequences,  $j: e \mapsto \langle e \rangle$ . An alternative form may be constructed by building a pair of sequences  $\sigma_e$  and  $\sigma_l$  denoting the earliest and latest sequences of jobs, respectively. Then the algorithm to be presented will construct  $\sigma_e = \langle j_3, j_1 \rangle$  and  $\sigma_l = \langle j_2, j_4 \rangle$ , giving the schedule  $\sigma_e \wedge \sigma_l = \langle j_3, j_1, j_2, j_4 \rangle$ . In the specification we have a choice as to whether to sort the jobs first on processing time and then pick off the corresponding least job or to determine those jobs with minimum processing time as per the algorithm described by Vollmann *et al.* and used in the above specification. The latter choice leads to the following specification. The use of a sort requires a different domain structure for *MACHINE\_LIST* since order is involved and the corresponding specification is presented in the subsequent subsection for comparison.

Johnson's algorithm, denoted  $\mathcal{J}_2$ , may then be defined as a function from a pair of machine lists to a schedule:

$$\mathcal{J}_2: (JOB \xrightarrow[m]{} TIME)^2 \longrightarrow JOB^*$$

$$\mathcal{J}_2(\mu_1, \mu_2) \triangleq \text{let } (\sigma_e, \sigma_l) = \mathcal{S}_2[\mu_1, \mu_2](\Lambda, \Lambda) \text{ in } \sigma_e \wedge \sigma_l$$

subject to the well-formedness constraint that the same jobs occur in both machine lists:

$$\mathcal{J}_2: (JOB \xrightarrow[m]{} TIME)^2 \longrightarrow \mathbf{B}$$

$$\mathcal{J}_2(\mu_1, \mu_2) \triangleq \text{dom } \mu_1 = \text{dom } \mu_2$$

I did not bother to give this constraint in the first specification, although it was, of course, essential. This is another simple illustration of stating explicitly the hidden assumptions. The  $\mathcal{S}_2$  algorithm is

$$\mathcal{S}_2: (JOB \xrightarrow[m]{} TIME)^2 \longrightarrow (JOB^*)^2 \longrightarrow (JOB^*)^2$$

$$\mathcal{S}_2[\theta, \theta](\sigma_e, \sigma_l) \triangleq (\sigma_e, \sigma_l)$$

$$\mathcal{S}_2[\mu_1, \mu_2](\sigma_e, \sigma_l) \triangleq$$

$$\text{let } (j_1, t_1) = \text{min}(\mu_1), (j_2, t_2) = \text{min}(\mu_2) \text{ in}$$

$$t_1 \leq t_2$$

$$\rightarrow \mathcal{S}_2[\{j_1\} \triangleleft \mu_1, \{j_1\} \triangleleft \mu_2](\sigma_e \wedge \langle j_1 \rangle, \sigma_l)$$

$$\rightarrow \mathcal{S}_2[\{j_2\} \triangleleft \mu_1, \{j_2\} \triangleleft \mu_2](\sigma_e, \langle j_2 \rangle \wedge \sigma_l)$$

For completeness I present here the complete specification of the *min* algorithm that I have chosen to employ in this case:

$$\text{min}: (JOB \xrightarrow[m]{} TIME) \longrightarrow (JOB \times TIME)$$

$$\text{min}(\mu) \triangleq$$

$$\text{let } (j, t) \in \mu \text{ in}$$

$$\text{let } (t_{\text{min}}, js) = \text{min}[\{j\} \triangleleft \mu](t, \{j\}) \text{ in}$$

$$\text{let } j' \in js \text{ in } (j', t_{\text{min}})$$

together with the well-formedness constraint (or context condition) which insists that a machine list  $\mu$  not be empty:

$$\text{is-wf-min}: (JOB \xrightarrow[m]{} TIME) \longrightarrow \mathbf{B}$$

$$\text{is-wf-min}(\mu) \triangleq \mu \neq \theta$$

The tail-recursive function *min* computes the set of  $(j, t)$  pairs for minimum time  $t$ :

## DISCOVERING SPECIFICATIONS

$$\min: (JOB \xrightarrow[m]{\rightarrow} TIME) \longrightarrow (TIME \times \mathcal{P}JOB) \longrightarrow (TIME \times \mathcal{P}JOB)$$

$$\begin{aligned} \min[[j \mapsto t] \cup \mu](t_{min}, js) &\triangleq \\ t < t_{min} &\rightarrow \min[[\mu]](t, \{j\}) \\ t = t_{min} &\rightarrow \min[[\mu]](t_{min}, js \cup \{j\}) \\ t > t_{min} &\rightarrow \min[[\mu]](t_{min}, js) \\ \min[[\theta]](t_{min}, js) &\triangleq (t_{min}, js) \end{aligned}$$

### 6.2. An Alternative Specification

Instead of using an algorithm to compute those jobs with minimum processing time, it is clear that if the machine lists were sorted in ascending order then one might just pick off the appropriate jobs. However sorting requires order, for which the sequence is the appropriate domain. Therefore, the domain of machine lists is now given as

$$MACHINE\_LIST = (JOB \times TIME)^*$$

I really ought to provide an appropriate invariant which states that a sequence of  $(j, t)$  pairs is a representation of a map. Such an invariant is exactly analogous to that discussed in Chapter 2 when we considered how a sequence might be used to represent a set. The essential property is, of course, that a machine list,  $\mu$ , does not contain pairs  $(j, t)$  and  $(j, t')$ , where  $t \neq t'$ . This may be captured by

$$\begin{aligned} \text{inv-}MACHINE\_LIST: (JOB \times TIME)^* &\longrightarrow \mathbf{B} \\ \text{inv-}MACHINE\_LIST(\mu) &\triangleq \text{card} \circ \text{elems} \circ \pi_1^* \mu = \text{len } \mu \end{aligned}$$

which states that if we take the cardinality of the set of all jobs in a machine list then it ought to be equal to the length of the machine list. The alternative version of the Johnson algorithm is, then:

$$\begin{aligned} \mathcal{J}_2: ((JOB \times TIME)^*)^2 &\longrightarrow JOB^* \\ \mathcal{J}_2(\mu_1, \mu_2) &\triangleq \\ \text{let } \mu'_1 = \text{sort}(\mu_1), \mu'_2 = \text{sort}(\mu_2) &\text{ in} \\ \text{let } (\sigma_e, \sigma_l) = \mathcal{S}_2[[\mu'_1, \mu'_2]](\Lambda, \Lambda) &\text{ in } \sigma_e \wedge \sigma_l \end{aligned}$$

with the well-formedness given by

$$\begin{aligned} \text{is-wf-}\mathcal{J}_2: ((JOB \times TIME)^*)^2 &\longrightarrow \mathbf{B} \\ \text{is-wf-}\mathcal{J}_2(\mu_1, \mu_2) &\triangleq (\text{len } \mu_1 = \text{len } \mu_2) \wedge (\text{elems} \circ \pi_1^* \mu_1 = \text{elems} \circ \pi_1^* \mu_2) \end{aligned}$$

Note that the algorithm is valid even in the case that  $\mu_1 = \mu_2 = \Lambda$ . The modified  $\mathcal{S}_2$  algorithm is

$$\begin{aligned} \mathcal{S}_2: & ((JOB \times TIME)^*)^2 \longrightarrow (JOB^*)^2 \longrightarrow (JOB^*)^2 \\ \mathcal{S}_2[\Lambda, \Lambda](\sigma_e, \sigma_l) & \triangleq (\sigma_e, \sigma_l) \\ \mathcal{S}_2[\langle (j_1, t_1) \rangle \wedge \mu_1, \langle (j_2, t_2) \rangle \wedge \mu_2](\sigma_e, \sigma_l) & \triangleq \\ & t_1 \leq t_2 \\ & \rightarrow \mathcal{S}_2[\mu_1, \{(j_1, -)\} \wp (\langle (j_2, t_2) \rangle \wedge \mu_2)](\sigma_e \wedge \langle j_1 \rangle, \sigma_l) \\ & \rightarrow \mathcal{S}_2[\{(j_2, -)\} \wp (\langle (j_1, t_1) \rangle \wedge \mu_1), \mu_2](\sigma_e, \langle j_2 \rangle \wedge \sigma_l) \end{aligned}$$

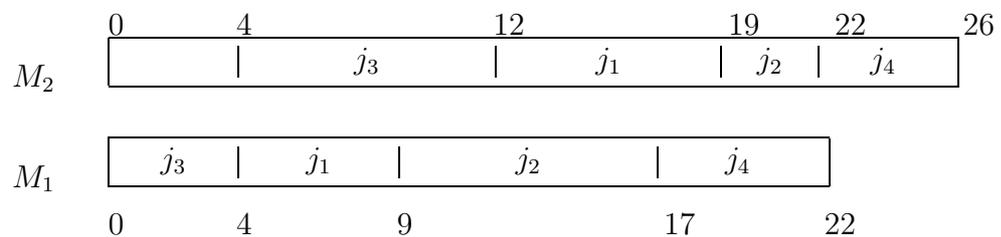
The null argument in  $\{(j_2, -)\} \wp (\langle (j_1, t_1) \rangle \wedge \mu_1)$  indicates that we do not know nor do we care what time is associated with  $j_2$ . Before turning to consider, a generalisation of the scheduling algorithm to  $n$  machines,  $n > 2$ , it would be appropriate, from the problem solving point of view, to consider how the results of the scheduling are employed. This is presented next.

### 6.3. The GANTT Diagram

The computed schedule is a sequence of jobs. In this subsection, my use of the word *sequence* is deliberately ambiguous. Using the machine lists, we then compute the start and stop times associated with each machine. However, these must be computed in the order, machine 1 followed by machine 2, since this is fundamental to the problem. Moreover, it is trivially clear that a job must be completed on a machine before another one starts. Using the simple example given above, the schedule  $\langle j_3, j_1, j_2, j_4 \rangle$  is converted into

$$[j_3 \mapsto (0, 4), j_1 \mapsto (4, 9), j_2 \mapsto (9, 17), j_4 \mapsto (17, 22)]$$

for the first machine and the *GANTT* diagram for the sample problem is



Note that the start time of a particular job on machine 1 is the stop time of the previous job, and the stop time is computed by adding the duration, obtained from the machine list, to the stop time. In the case of the second machine, the situation is slightly more complicated. It might happen that a job terminates on the second

## DISCOVERING SPECIFICATIONS

machine before the next job is finished on the first machine, in which case the start time of the new job on machine 2 is the maximum of the stop time of the previous job on machine 2 and the stop time of the new job on machine 1. Before, presenting the algorithm to display the GANTT chart, I give the relevant domain equations that are used:

$$\begin{aligned} GANTT &= SEQUENCE \times SEQUENCE \\ SEQUENCE &= JOB \xrightarrow{m} TIME^2 \end{aligned}$$

The GANTT chart is considered to be an ordered pair of sequences corresponding to machine 1 and machine 2, in that order. A sequence is a set of ordered pairs  $(j, (s, t))$  where  $s$  and  $t$  are the start and stop times of job  $j$  on a particular machine. Perhaps one would have supposed that a sequence of ordered pairs would have been the more appropriate model. This is unnecessary, since the ordering information is uniquely determined by the start and stop times. The specification is:

$$\begin{aligned} display: (JOB \xrightarrow{m} TIME)^2 \times JOB^* &\longrightarrow (JOB \xrightarrow{m} TIME^2)^2 \\ display(\mu_1, \mu_2, \sigma) &\triangleq \\ \text{let } \sigma = \langle j \rangle \wedge \tau, \bar{t}_1 = 0, \bar{t}_2 = 0 &\text{ in} \\ \text{let } s_1 = \bar{t}_1, t_1 = \bar{t}_1 + \mu_1(j) &\text{ in} \\ \text{let } s_2 = (t_1 \uparrow \bar{t}_2), t_2 = (t_1 \uparrow \bar{t}_2) + \mu_2(j) &\text{ in} \\ \text{let } \xi_1 = [j \mapsto (s_1, t_1)], \xi_2 = [j \mapsto (s_2, t_2)] &\text{ in} \\ display[\tau](\mu_1, t_1, \xi_1, \mu_2, t_2, \xi_2) & \end{aligned}$$

where I have decided to use the symbolic operators  $\downarrow$  and  $\uparrow$  for *min* and *max*, respectively. The tail recursive form is given by

$$\begin{aligned} display: (JOB \xrightarrow{m} TIME) \times TIME \times (JOB \xrightarrow{m} TIME^2) & \\ \times (JOB \xrightarrow{m} TIME) \times TIME \times (JOB \xrightarrow{m} TIME^2) & \\ \longrightarrow (JOB \xrightarrow{m} TIME) \times TIME \times (JOB \xrightarrow{m} TIME^2) & \\ \times (JOB \xrightarrow{m} TIME) \times TIME \times (JOB \xrightarrow{m} TIME^2) & \\ display[\langle j \rangle \wedge \tau](\mu_1, \bar{t}_1, \xi_1, \mu_2, \bar{t}_2, \xi_2) &\triangleq \\ \text{let } s_1 = \bar{t}_1, t_1 = \bar{t}_1 + \mu_1(j) &\text{ in} \\ \text{let } s_2 = (t_1 \uparrow \bar{t}_2), t_2 = (t_1 \uparrow \bar{t}_2) + \mu_2(j) &\text{ in} \\ display[\tau](\mu_1, t_1, \xi_1 \cup [j \mapsto (s_1, t_1)], \mu_2, t_2, \xi_2 \cup [j \mapsto (s_2, t_2)]) & \\ display[\Lambda](\mu_1, \bar{t}_1, \xi_1, \mu_2, \bar{t}_2, \xi_2) &\triangleq (\mu_1, \bar{t}_1, \xi_1, \mu_2, \bar{t}_2, \xi_2) \end{aligned}$$

A few brief remarks on the conceptual model behind the specification are in order. The *display* algorithm is written in the form of an initialisation section followed by a while loop. In an implementation, where the *GANTT* chart is to be displayed on a graphics device, then instead of the initialisation  $\xi_1 = [j \mapsto (s_1, t_1)]$ , the appropriate graphics code is executed. The same remark applies in the while loop. In other words,  $\xi_1$  and  $\xi_2$  may be interpreted as specified regions on a graphics device. Note that within the algorithm, the machine lists  $\mu_1$  and  $\mu_2$  are read-only structures. Finally, the only real transformations in the while loop are the changes to the stop time arguments  $t_1$  and  $t_2$ . In particular, upon termination,  $t_1$  and  $t_2$  give the total processing times on machine 1 and machine 2, respectively.

#### 6.4. Johnson's Algorithm for 3 Machines

Although an explicit dynamic-programming solution may be found to the two machine case, it can not be extended to the three machine case; it can be solved by the technique of 'permutation' programming (Walsh 1975, 181). However, there is a special three machine case which is reducible. "The basic principle is that if the maximum time requirement on machine 2 is less than or equal to the minimum time requirements on machines 1 and 3, then the problem may be reduced to the two machine case by adding the time requirements of machines 1 and 3 to form a new machine list and adding the time requirements of machines 2 and 3 to form the second machine list" (Eiselt and von Frajer 1977, 345). Formally, this gives the specification

$$\begin{aligned}
\mathcal{J}_3: (JOB \xrightarrow{m} TIME)^3 &\longrightarrow JOB^* \\
\mathcal{J}_3(\langle \mu_1, \mu_2, \mu_3 \rangle) &\triangleq \\
\text{let } J_1 = (\mu_1^{-1} \circ \downarrow / \circ \text{rng } \mu_1) &\triangleleft \mu_1, \\
J_2 = (\mu_2^{-1} \circ \uparrow / \circ \text{rng } \mu_2) &\triangleleft \mu_2, \\
J_2 = (\mu_3^{-1} \circ \downarrow / \circ \text{rng } \mu_3) &\triangleleft \mu_3 \text{ in} \\
\text{let } (-, t_1) \in J_1, (-, t_2) \in J_2, &(-, t_3) \in J_3 \text{ in} \\
(t_2 \leq t_1) \wedge (t_2 \leq t_3) & \\
\rightarrow \mathcal{J}_2(\mu_1 \oplus \mu_2, \mu_2 \oplus \mu_3) & \\
\rightarrow \perp &
\end{aligned}$$

where  $\oplus$  denotes bag addition, since  $\mu \in (JOB \xrightarrow{m} TIME)$  is structurally a bag. In the event, that the special case is not satisfied,  $\perp$ , then we would have to resort to

## DISCOVERING SPECIFICATIONS

the heuristic algorithm for  $n$  machines,  $n > 2$  in the next subsection. Rather than wait until I have discussed that algorithm before presenting the ‘completion’ of this one, I give the modifications immediately without comment:

$$\begin{aligned}
 & (t_2 \leq t_1) \wedge (t_2 \leq t_3) \\
 & \rightarrow \mathcal{J}_2(\mu_1 \oplus \mu_2, \mu_2 \oplus \mu_3) \\
 & \rightarrow |J_1| = 1 \\
 & \rightarrow \text{let } J_1 = \{(j_1, t_1)\} \text{ in} \\
 & \quad \langle j_1 \rangle \wedge \mathcal{J}_3(\{\{j_1\} \leftarrow -\}^* \langle \mu_1, \mu_2, \mu_3 \rangle) \\
 & \rightarrow \mathcal{J}_2(\mu_2, \mu_3)
 \end{aligned}$$

for which a special termination case must be considered:

$$\mathcal{J}_3(\langle \theta, \mu_2, \mu_3 \rangle) \triangleq \Lambda$$

Let us now finally consider the general heuristic algorithm.

### 6.5. A Heuristic Algorithm for $n$ Machines

In the general case, the best that we can do is to use heuristics. From Eiselt and von Frajer (1977, 347) “the order with the shortest working time requirement (on machine 1) comes first, the one with the second shortest comes second, etc. If this rule does not lead to a unique sequence, then the time requirement on the next machine determines which order has priority”. I choose, for simplicity, to base the specification on the alternative Johnson’s algorithm for the two machine case which uses a sort:

$$\begin{aligned}
 & \mathcal{J}_n: ((JOB \times TIME)^*)^n \longrightarrow JOB^* \\
 & \mathcal{J}_n(\langle \mu \rangle \wedge \tau) \triangleq \\
 & \quad \text{let } J = (\mu^{-1} \circ \downarrow / \circ \text{rng } \mu) \triangleleft \mu \text{ in} \\
 & \quad |J| = 1 \\
 & \quad \rightarrow \text{let } \{(j, -)\} = J \text{ in} \\
 & \quad \quad \langle j \rangle \wedge \mathcal{J}_n(\{\{j\} \leftarrow -\}^* (\langle \mu \rangle \wedge \tau)) \\
 & \quad \rightarrow \mathcal{J}_{n-1}(\tau) \\
 & \mathcal{J}_n(\langle \theta \rangle \wedge \tau) \triangleq \Lambda
 \end{aligned}$$

where, in the event that  $n = 3$ , then the special case is tried. Comparing the earlier dictionary problem with the scheduling problem, it is evident that there are fragments of operator calculus specification which occur in both. Specifications of both, moreover, avail of the concept of bag, which found a useful rôle in the bill

of materials problem. In discovering specifications, it is common to find the same operational calculus forms turning up in different problem domains, a phenomenon which is due largely to (1) the small well-defined set of basic *Meta-IV* domains and their operators, and (2) the symbolic nature of the notation as used in the Irish School. By emphasising the use of abstract symbolic operators and terms one is able to obtain a greater degree of reusability of specifications than would otherwise be the case were one to use ‘meaningful’ names.

## 7. Summary

I began this Chapter on the discovery of specifications at the ‘high-level’ of requirements and effectively ended up with specifications of very concrete algorithms. At each end of the spectrum the same *Meta-IV* operator calculus was employed. One might wonder whether or not there was room for directly executable specifications in the Irish School of the *VDM*. Should one not cater for every stage in the life-cycle? At one time I would have said ‘yes’, without hesitation. Now I would answer in the negative. I believe that there is a fortunate discontinuity between the world of specifications and the real world of programming. Effectively, from a specification it is immediately clear how one might proceed to an encoding; to go from encoding in one language to an encoding in another language is much more difficult.

Historically, most computing knowledge has been embedded within a programming framework. There are domains, such as numerical mathematics and formal language theory, which have managed to be programming-independent but at the expense of specialised notation. Where knowledge is embedded in encodings, then it is very likely that it will remain compartmentalised, of benefit only to those who ‘know’ the domain and its language. The method by which one extracts specifications from encodings, and specification *is* knowledge, is known as reverse engineering. I would like to continue to develop the theme of discovery of specifications, but this time from a reverse engineering viewpoint.

## DISCOVERING SPECIFICATIONS

Of all the problem domains that I have considered for this purpose, that of computer graphics and computer-aided design has proven to be the most fruitful, chiefly because computer graphics algorithms are viewed and presented in the context of imperative programming with visual side-effects. In addition, having reverse-engineered well-known graphics algorithms, the re-encoding of the resulting specifications gives a visual feedback which confirms the validity of the entire approach. I present a very brief summary of these results in the following Chapter which I have entitled *Graphics*.

# Chapter 7

## Graphics

### 1. Introduction

Curiously enough, although recursive definitions permeate the whole of CAD and computer graphics, especially where the treatment is mathematical, which is usually the case, it is somewhat astonishing to find that the algorithms are normally given in imperative form. Doubtless, there are many good reasons for such an approach, chiefly among which I would identify a particular author's concern to address as wide an audience as possible—especially those who program in FORTRAN, C, Pascal, Ada, etc. However, is it not strange to suppose that those who are able to understand the mathematics, must also be given algorithms which are specified in imperative programming language constructs? Of course, another putative reason might be the simple fact that, to a certain extent, matrices and vectors are the stuff of which CAD and computer graphics are made. Now, it has been traditional in computing, to process same with indices, and in modern structured programming languages to associate the *for loop* with such processing. Finally, it must be noted that much of the early development of the subject took place at a time when both hardware and programming languages were very primitive indeed.

I considered it to be of great conceptual interest to examine to what extent classical algorithms might be elegantly expressed recursively, and, if possible, without any reference to indices. Such an approach I term minimalist in the sense of abstract specification. That is to say, all of the relevant aspects must be captured and unnecessary detail pruned away. On the other hand, I firmly believe in the concept of *executable specifications*—even to the extent that they be efficient, such notion being defined within a suitable conceptual framework. That which I have chosen is algorithmic expressiveness within PROLOG. By efficient within such a framework I shall mean that (i) predicates are tail-recursive or (ii) (partial) solu-

# GRAPHICS

tions are directly obtained via unification on partial list structures. I reiterate that I do **not** mean that *Meta-IV* specifications should be **directly** executable. There is always that jump between the *Meta-IV* and the language to be used for executable specifications. For practical proof of concept, I chose to encode all of the material to be presented in this Chapter in HyperTalk, the script language of HyperCard. (Both HyperTalk and HyperCard are registered trademarks of the Apple Computer Corporation). Illustrations of graphics generated by these algorithms are included.

## 2. Line Drawing Algorithms

To illustrate the application of the *VDM Meta-IV* to the specification of graphics algorithms, I begin by presenting some well-known line drawing algorithms: the simple DDA algorithm and Bresenham's algorithm. Given a pair of points in device coordinates, both algorithms generate the 'in-between' points as illustrated in the figure

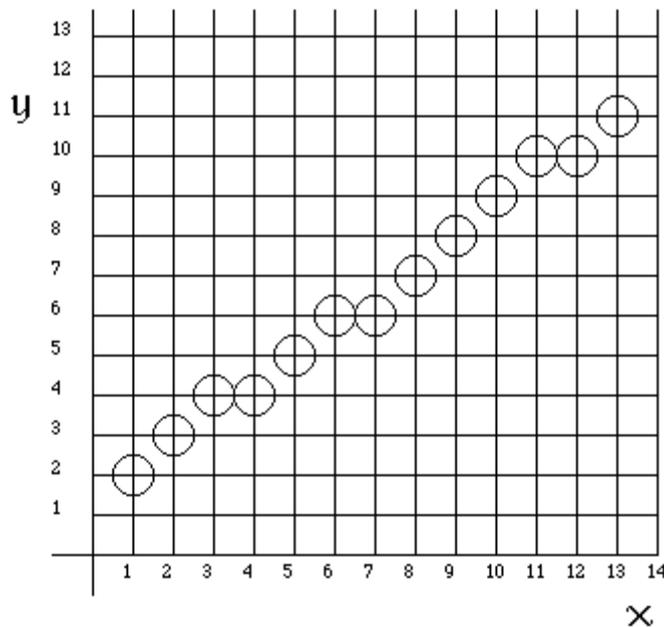


Figure: Pixels generated by line-drawing algorithms

For the purpose of clarity, the pixels are represented by circles centered on the computed points. It might be supposed that since the algorithms are already well-known and are, moreover, implemented in hardware, then the reverse-engineering exercise is purely of academic value. This would be a very short-sighted view. One immediate goal is to demonstrate that the two algorithms are structurally the same. Another is to exhibit what I mean by the specification of graphics algorithms through the use of simple examples. Indeed, the derivation of such line-drawing algorithms using program transformations has appeared in the literature (Sproull, 1982).

### 2.1. Simple DDA Algorithm

Consider the simple DDA algorithm, which is usually given in an imperative style (Newman and Sproull 1979; Hearn and Baker 1986; Salmon and Slater 1987). Mathematically, it is based on the general recurrence relation  $z_{i+1} = \mathcal{H}(z_i)$ , where  $\mathcal{H}(z) = (\mathcal{F} \times \mathcal{G})(x, y) = (\mathcal{F}(x), \mathcal{G}(y))$ . For the DDA algorithm

$$\begin{aligned} x_{i+1} &= \mathcal{F}(x_i) = x_i + 1 \\ y_{i+1} &= \mathcal{G}(y_i) = y_i + \frac{dy}{dx} \end{aligned}$$

where  $i = 0, 2, \dots, dx - 1$  and it is assumed that  $0 < \frac{dy}{dx} < 1$ . Since the  $z_i = (x_i, y_i)$  values are complex numbers which lie on the line, and the DDA algorithm must produce gaussian integers, then the  $i$ th gaussian integer has the value

$$(x_i, \text{round}(y_i + \frac{dy}{dx}))$$

#### 2.1.1. DDA Version 1

In this first version of the algorithm, a sequence of points (i.e., complex numbers), which are on the line, is computed. Then, the function *round* is mapped over this sequence to give the final result—a sequence of gaussian integers. The algorithm is

$$\begin{aligned} \text{dda: } DC\_LINE &\longrightarrow DC\_POINT^* \\ \text{dda}([z_1, z_2]) &\triangleq \\ &\text{let } dx = x_2 - x_1, dy = y_2 - y_1 \text{ in} \\ &(\mathcal{I} \times \text{round})^* \circ \text{points}([z_1, z_2], \frac{dy}{dx}) \end{aligned}$$

where  $\mathcal{I}$  denotes the identity function. The actual points are given by

## GRAPHICS

```

points: DC_LINE × R → POINT*
points([z, z],  $\frac{dy}{dx}$ ) ≙ ⟨ z ⟩
points([z1, z2],  $\frac{dy}{dx}$ ) ≙ ⟨ z1 ⟩ ^ points([(x1 + 1, y1 +  $\frac{dy}{dx}$ ), z2],  $\frac{dy}{dx}$ )

```

**Remark:** The points algorithm may easily be transformed into an efficient tail-recursive version:

```

points: DC_LINE × R → (POINT* → POINT*)
points([[z, z],  $\frac{dy}{dx}$ ]]p ≙ ⟨ z ⟩ ^ p
points([[z1, z2],  $\frac{dy}{dx}$ ]]p ≙ points([(x1 + 1, y1 +  $\frac{dy}{dx}$ ), z2],  $\frac{dy}{dx}$ ](⟨ z1 ⟩ ^ p)

```

I give the details here of the domains, *DC\_LINE*, *DC\_POINT* and *POINT*, which are all *Meta-IV* trees:

$$[z_j, z_k] \in DC\_LINE = DC\_POINT \times DC\_POINT$$

$$z = (x, y) \in DC\_POINT = \mathbf{N} \times \mathbf{N}$$

$$z = (x, y) \in POINT = \mathbf{R} \times \mathbf{R}$$

There are, of course, required invariants and well-formedness conditions which I do not give here. A brief note on some notational conventions which I have been using in the domain of computer graphics specifications are in order. Trees or cartesian products occur very frequently in graphics specifications. In my early work, I noted that the corresponding use of the *mk*-construct in *Meta-IV* led to very verbose notational forms that tended to obscure rather than clarify the underlying structure. For this very reason I introduced the ‘for ... use ...’ clause into the Irish *VDM*. In addition, I found it expedient to use a variety of bracket forms in order to avoid the ‘LISP phenomenon’. In the case in question, I adopted the notation  $[z_j, z_k]$ , to represent a finite line segment, from Knuth’s *METAFONT*, where it is automatically understood that every occurrence of  $z$  represents  $(x, y)$ .

### 2.1.2. DDA Version 2

In this second version, the goal is to compute the rounded  $y$  value within the body of points.

```

dda: DC_LINE → DC_POINT*
dda([z1, z2]) ≙
  let dx = x2 - x1, dy = y2 - y1 in
    pixels([z1, z2], y1,  $\frac{dy}{dx}$ )

```

where

$$\begin{aligned}
 &\text{pixels: } DC\_LINE \times \mathbf{R} \times \mathbf{R} \longrightarrow DC\_POINT^* \\
 &\text{pixels}([z, z], y_r, \frac{dy}{dx}) \triangleq \langle z \rangle \\
 &\text{pixels}([z_1, z_2], y_r, \frac{dy}{dx}) \triangleq \\
 &\quad \langle z_1 \rangle \wedge \text{pixels}([(x_1 + 1, \text{round}(y_r + \frac{dy}{dx})), z_2], y_r + \frac{dy}{dx}, \frac{dy}{dx})
 \end{aligned}$$

The derivation of a tail-recursive form is obvious. From the tail-recursive form we already know how to construct the equivalent *while loop* program.

## 2.2. The Bresenham Algorithm

A similar analysis of the Bresenham algorithm (Hearn and Baker 1986), which computes the *same* gaussian integers directly without doing any real arithmetic computations, gives the recurrence relations

$$\begin{aligned}
 x_{i+1} &= \mathcal{F}(x_i) = x_i + 1 \\
 y_{i+1} &= \mathcal{G}(y_i) = \begin{cases} y_i, & \text{if } e_i > 0 \\ y_i + 1 & \text{otherwise} \end{cases} \\
 e_0 &= 2dy - dx \\
 e_{i+1} &= \begin{cases} e_i + 2dy & \text{if } e_i > 0 \\ e_i + 2dy - 2dx & \text{otherwise} \end{cases}
 \end{aligned}$$

As in the case of the DDA algorithm we expect to have an initialisation section:

$$\begin{aligned}
 &bres: DC\_LINE \longrightarrow DC\_POINT^* \\
 &bres([z_1, z_2]) \triangleq \\
 &\quad \text{let } dx = x_2 - x_1, dy = y_2 - y_1 \text{ in} \\
 &\quad \text{let } e = 2dy - dx \text{ in} \\
 &\quad \text{pixels}([z_1, z_2], dx, dy, e)
 \end{aligned}$$

followed by the ‘while loop’:

$$\begin{aligned}
 &\text{pixels: } DC\_LINE \times \mathbf{Z} \times \mathbf{Z} \times \mathbf{Z} \longrightarrow DC\_POINT^* \\
 &\text{pixels}([z, z], dx, dy, e) \triangleq \langle z \rangle \\
 &\text{pixels}([z_1, z_2], dx, dy, e) \triangleq \\
 &\quad \text{if } e < 0 \\
 &\quad \quad \text{then } \langle z_1 \rangle \wedge \text{pixels}([(x_1 + 1, y_1), z_2], dx, dy, e + 2dy) \\
 &\quad \quad \text{else } \langle z_1 \rangle \wedge \text{pixels}([(x_1 + 1, y_1 + 1), z_2], dx, dy, e + 2dy - 2dx)
 \end{aligned}$$

Note the remarkable structural similarity between the refined DDA algorithm given above and this one. Such is the power of the specification method to reveal the essentials. I am still not entirely satisfied with the specifications from an æsthetic

## GRAPHICS

point of view. Although it is easy to distinguish the constants from the variables, it is not obvious how the different arguments ought to be grouped in order to select an intrinsic structure which I intuitively feel is present.

### 2.2. Implementations in PROLOG

Whereas I have stated that one may derive an imperative algorithm directly from the tail-recursive form, I prefer to demonstrate how the tail-recursive forms may be directly encoded in PROLOG. The syntax used here is the ‘standard’ Edinburgh syntax as used in (Clocksin and Mellish 1987). The algorithms given here, without comment, were executed on a VAX 11/780 using C-PROLOG (Pereira 1986). Comment should not be necessary. The direct equivalence between the specification and encoding is obvious, assuming that one understands the conceptual model of PROLOG.

#### 2.2.1. The Simple DDA Algorithm — Version 2

```
dda([X1, Y1], [X2, Y2], Points) :-  
    Dx is X2 - X1,  
    Dy is Y2 - Y1,  
    Slope is Dy/Dx,  
    pixels([X1, Y1], [X2, Y2], Y1, Slope, Points).  
  
pixels([X, Y], [X, Y], Yr, Slope, [[X, Y]]).  
pixels([X1, Y1], [X2, Y2], Yr, Slope, [[X1, Y1] | Rest]) :-  
    X11 is X1 + 1,  
    Yr1 is Yr + Slope,  
    round(Yr1, Y11),  
    pixels([X11, Y11], [X2, Y2], Yr1, Slope, Rest).  
  
round(Real, Int) :-  
    R is Real + 0.5,  
    Int is floor(R).
```

2.2.2. The Bresenham Algorithm

```

bres([X1, Y1], [X2, Y2], Points) :-
    Dx is X2 - X1,
    Dy is Y2 - Y1,
    E is 2 * Dx - Dy,
    pixels([X1, Y1], [X2, Y2], Dx, Dy, E, Points).

pixels([X, Y], [X, Y], Dx, Dy, E, [[X, Y]]).
pixels([X1, Y1], [X2, Y2], Dx, Dy, E, [[X1, Yone] | Rest]) :-
    E < 0,
    X11 is X1 + 1,
    E1 is E + 2 * Dy,
    pixels([X11, Y1], [X2, Y2], Dx, Dy, E1, Rest).
pixels([X1, Y1], [X2, Y2], Dx, Dy, E, [[X1, Yone] | Rest]) :-
    E >= 0,
    X11 is X1 + 1,
    Y11 is Y1 + 1,
    E1 is E + 2 * Dy - 2 * Dx,
    pixels([X11, Y11], [X2, Y2], Dx, Dy, E1, Rest).

```

These encodings do not produce any graphics—they are purely numerical computations. Attachment of graphics routines is straightforward—instead of ‘storing’ a computed point in a list, one may plot it. In fact, one may prefer to do both: store and plot. From the point of view of the conceptual model underlying computer graphics, this duality of store/plot is fundamental. One often refers to it as the data structure/procedure duality of graphics. The *VDM* specification permits this duality to be ‘seen at a glance’.

# GRAPHICS

## 3. Polygon Clipping

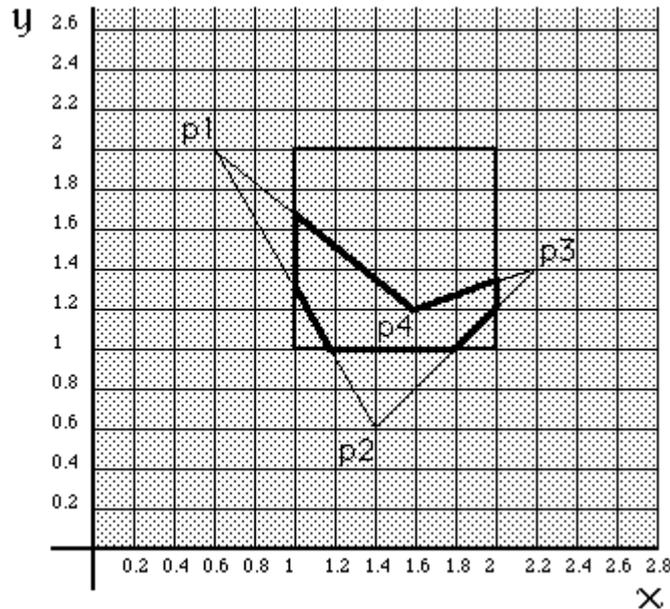


Figure: Sutherland-Hodgman Polygon Clipping

Newman and Sproull (1979) assert that the ‘ordinary’ line clipping algorithms can not handle polygon clipping. Certainly, in the case of the Cohen and Sutherland algorithm, a particular difficulty is raised by the point swapping behaviour. Assuming that some mechanism was introduced to overcome this problem, then it seems reasonable to suppose that line clipping algorithms would suffice, at least for convex polygons. Essentially, such a polygon clipping algorithm based on line clipping would be expressed simply in the form

$$\text{clip}[[w]]/p$$

Such an algorithm considers the window  $w$  to be an invariant against which the polygon  $p$  is processed. An entirely different view is adopted in the Sutherland and Hodgson algorithm, where it is not the window  $w$  which is invariant but each edge  $e$  of the window. From the reverse-engineered specification to be presented in this section, I discuss the corresponding imperative encoding which in turn was translated into HyperTalk. A sample application is shown above.

### 3.1. Sutherland-Hodgman Algorithm

The algorithm presented here has been reverse-engineered from the Pascal version given in the Foley and van Dam text book (1982). A  $2d$  window is considered to be a convex polygon, whose vertices are listed in anti-clockwise order:

$$WINDOW = POLYGON$$

$$POLYGON = POLYLINE$$

$$POLYLINE = VERTEX^+$$

$$VERTEX = POINT$$

where a window must be a convex polygon and a polygon is a polyline whose first and last points are identical:

$$\begin{aligned} \text{inv-WINDOW: } WINDOW &\longrightarrow \mathbf{B} \\ \text{inv-WINDOW}(w) &\triangleq \text{convex}(w) \wedge \dots \end{aligned}$$

$$\begin{aligned} \text{inv-POLYGON: } POLYGON &\longrightarrow \mathbf{B} \\ \text{inv-POLYGON}(p) &\triangleq \text{first}(p) = \text{last}(p) \wedge \text{inv-POLYLINE}(p) \end{aligned}$$

Again I refrain from giving all of the invariant details. Note that I infer that a window may be a polygon and not just a conventional rectangle. There appears to be nothing in the specification to rule out this generalisation. For example, a conventional  $2d$  window  $w$  will have the form

$$w = \langle w_1, w_2, w_3, w_4, w_1 \rangle$$

$$w_1 = (x_{min}, y_{min})$$

$$w_2 = (x_{max}, y_{min})$$

$$w_3 = (x_{max}, y_{max})$$

$$w_4 = (x_{min}, y_{max})$$

In the body of the clip algorithm the notion of edge is used:

$$EDGE = VERTEX \times VERTEX$$

The algorithm itself may be presented as a simple tail-recursive function:

$$\begin{aligned} \text{clip: } WINDOW &\longrightarrow POLYGON \longrightarrow POLYGON \\ \text{clip}[\langle w_i, w_{i+1} \rangle \wedge \tau]p &\triangleq \text{clip}[\langle w_{i+1} \rangle \wedge \tau] \circ \text{clip\_edge}[\langle w_i, w_{i+1} \rangle]p \\ \text{clip}[\langle w_{n-1}, w_n \rangle]p &\triangleq \text{clip\_edge}[\langle w_{n-1}, w_n \rangle]p \end{aligned}$$

This in itself is interesting. The sequence of window vertices is processed ‘2 at a time’, a feature common to many graphics algorithms. This is one of those special

## GRAPHICS

‘reduction’ paradigms mentioned in Chapter 4. In particular, I may express this specification succinctly in the form

$$\circ / (\text{clip\_edge}[\![ - , - ]\!] // w)^e p$$

where  $\text{clip\_edge}[\![ - , - ]\!] // w$  is used to construct the sequence of function operators

$$\langle \text{clip\_edge}[\![ w_1, w_2 ]\!], \dots, \text{clip\_edge}[\![ w_{n-1}, w_n ]\!] \rangle$$

which must then be reversed before reducing with respect to the functional composition operator. The  $\text{clip\_edge}$  function is given by

$$\begin{aligned} \text{clip\_edge}: \text{EDGE} &\longrightarrow \text{POLYGON} \longrightarrow \text{POLYGON} \\ \text{clip\_edge}[\![ e ]\!] \langle p, q \rangle &\triangleq \tau \\ \text{let } in\_first = \text{inside}[\![ e ]\!] p, in\_second = \text{inside}[\![ e ]\!] q &\text{ in} \\ in\_first \wedge in\_second &\rightarrow \langle q \rangle \wedge \text{clip\_edge}[\![ e ]\!] (\langle q \rangle \wedge \tau) \\ \neg in\_first \wedge in\_second &\rightarrow \text{let } r = \text{intersect}[\![ e ]\!] (p, q) \text{ in} \\ &\langle r, q \rangle \wedge \text{clip\_edge}[\![ e ]\!] (\langle q \rangle \wedge \tau) \\ in\_first \wedge \neg in\_second &\rightarrow \text{let } r = \text{intersect}[\![ e ]\!] (p, q) \text{ in} \\ &\langle r \rangle \wedge \text{clip\_edge}[\![ e ]\!] (\langle q \rangle \wedge \tau) \end{aligned}$$

where the base case is obviously,

$$\begin{aligned} \text{clip\_edge}[\![ e ]\!] \langle p, q \rangle &\triangleq \\ \text{let } in\_first = \text{inside}[\![ e ]\!] p, in\_second = \text{inside}[\![ e ]\!] q &\text{ in} \\ in\_first \wedge in\_second &\rightarrow \langle q \rangle \\ \neg in\_first \wedge in\_second &\rightarrow \text{let } r = \text{intersect}[\![ e ]\!] (p, q) \text{ in } \langle r, q \rangle \\ in\_first \wedge \neg in\_second &\rightarrow \text{let } r = \text{intersect}[\![ e ]\!] (p, q) \text{ in } \langle r \rangle \end{aligned}$$

The function ‘inside’ is determined by the sign of the  $z$ -component of the cross product of  $\overline{e_s e_e}$ , where  $e$  is the window edge  $e = (e_s, e_e)$ , and the vector  $\overline{e_s p}$ . I am using the classical convention that the directed vector  $\overline{ab}$  is equivalent to the difference of the two point vectors  $b$  and  $a$ :

$$\begin{aligned} \text{inside}: \text{EDGE} &\longrightarrow \text{VERTEX} \longrightarrow \mathbf{B} \\ \text{inside}[\![ e_s, e_e ]\!] p &\triangleq \text{let } (xi, yj, zk) = (e_e - e_s) \times (p - e_s) \text{ in } z \geq 0 \end{aligned}$$

The intersection point between the lines  $\mathcal{L}_t(p, q)$  and  $\mathcal{L}_u(e_s, e_e)$  may be obtained by solving a pair of simultaneous equations in  $t$  and  $u$ :

$$\mathcal{L}_t(p, q) = \mathcal{L}_u(e_s, e_e)$$

where I use the convention that  $\mathcal{L}_t(p, q) = (1-t)p + tq$ ,  $0 \leq t \leq 1$ . Using coordinates, gives

$$(1-t)x_p + tx_q = (1-u)x_{es} + ux_{ee}$$

$$(1-t)y_p + ty_q = (1-u)y_{es} + uy_{ee}$$

$$(1-t)z_p + tz_q = (1-u)z_{es} + uz_{ee}$$

Choosing the first pair of equations and rearranging terms appropriately gives:

$$t(x_q - x_p) - u(x_{ee} - x_{es}) = x_{es} - x_p$$

$$t(y_q - y_p) - u(y_{ee} - y_{es}) = y_{es} - y_p$$

For convenience, let us write

$$\Delta x_{pq} = x_q - x_p$$

$$\Delta y_{pq} = y_q - y_p$$

$$\Delta x_{se} = x_{ee} - x_{es}$$

$$\Delta y_{se} = y_{ee} - y_{es}$$

Then, the pair of simultaneous equations may be written as

$$t(\Delta x_{pq}) - u(\Delta x_{se}) = x_{es} - x_p$$

$$t(\Delta y_{pq}) - u(\Delta y_{se}) = y_{es} - y_p$$

Elimination of  $t$  gives

$$u = \frac{(x_{es} - x_p)\Delta y_{pq} - (y_{es} - y_p)\Delta x_{pq}}{\Delta y_{se}\Delta x_{pq} - \Delta x_{se}\Delta y_{pq}}$$

from which the intersection point may then be computed:

intersect: *EDGE*  $\longrightarrow$  *EDGE*  $\longrightarrow$  *VERTEX*

intersect $[[e_s, e_e]](p, q) \triangleq$

$$\text{let } u = \frac{(x_{es} - x_p)\Delta y_{pq} - (y_{es} - y_p)\Delta x_{pq}}{\Delta y_{se}\Delta x_{pq} - \Delta x_{se}\Delta y_{pq}} \text{ in } (\mathcal{L}_u(x_{es}, x_{ee}), \mathcal{L}_u(y_{es}, y_{ee}))$$

Considering the specification as a whole, it is obvious that only the high-level parts: clip and clip\_edge, are the most interesting. The rest is ‘routine’ computation dressed up in a particular style. Indeed the essence of the Sutherland and Hodgman algorithm is given by the expression

$$\text{clip}[[w]]p = \circ / (\text{clip\_edge}[[-, -]] // w) \circ p$$

The rest is mere detail.

## GRAPHICS

### 3.2. A Refinement

Introduction of slices for the window  $w[i \dots m]$ , the initial polygon  $p[j \dots n]$ , and the clipped polygon  $q[k \dots l]$ , leads to the obvious refinement:

$$\text{clip}[[w[i \dots m]]]p[1 \dots n] \triangleq \text{clip}[[w[i + 1 \dots m]]] \circ \text{clip\_edge}[[w[i], w[i + 1]]]p[1 \dots n]$$

Note that I do not include the termination form. This will be taken care of automatically when I move to the imperative refinement. The corresponding `clip_edge` algorithm is transformed into

$$\begin{aligned} & \text{clip\_edge}[[e, p[j \dots n]]]q[k \dots l] \triangleq \\ & \text{let } in\_first = \text{inside}[[e]]p[j], \quad in\_second = \text{inside}[[e]]p[j + 1] \text{ in} \\ & \quad in\_first \wedge in\_second \\ & \quad \rightarrow q[k \mapsto p[j + 1]] \cup \text{clip\_edge}[[e, p[j + 1 \dots n]]]q[k + 1 \dots l] \\ & \quad \neg in\_first \wedge in\_second \\ & \quad \rightarrow \text{let } r = \text{intersect}[[e]](p[j], p[j + 1]) \text{ in} \\ & \quad \quad q[k \mapsto r, k + 1 \mapsto p[j + 1]] \cup \text{clip\_edge}[[e, p[j + 1 \dots n]]]q[k + 2 \dots l] \\ & \quad in\_first \wedge \neg in\_second \\ & \quad \rightarrow \text{let } r = \text{intersect}[[e]](p[j], p[j + 1]) \text{ in} \\ & \quad \quad q[k \mapsto r] \cup \text{clip\_edge}[[e, p[j + 1 \dots n]]]q[k + 1 \dots l] \end{aligned}$$

The specification is still recursive. The introduction of the slice notation allows me to use a ‘for loop’ construct in the imperative encoding rather than the ‘while loop’ construct that one normally finds in the transformation of tail-recursive forms.

### 3.3. A Refinement — Skeletal Imperative Encoding

I will give a few brief remarks on the reasoning behind the transformation from specification to encoding. Consider the expression in the first ‘if then’ clause:

$$q[k \mapsto p[j + 1]] \cup \text{clip\_edge}[[e, p[j + 1 \dots n]]]q[k + 1 \dots l]$$

The ‘productive’ part,  $q[k \mapsto p[j + 1]]$ , gives the assignment, ‘ $q[k] \leftarrow p[j + 1]$ ’; and the  $q$  argument to the recursive call gives the index increment, ‘ $k \leftarrow k + 1$ ’. Since the window coordinates are processed in pairs, then assuming a slice of length  $m$ , the ‘for loop’ will range from 1 to  $m - 1$ . Similarly, the vertices of the polygon  $p$  are processed 2 at a time. A skeletal imperative encoding may be obtained directly:

```

for  $i = [1 \dots m - 1]$  do
   $e \leftarrow (w[i], w[i + 1]);$ 
   $k \leftarrow 1;$ 
  for  $j = [1 \dots n - 1]$  do
     $in\_first \leftarrow \text{inside}[e]p[j];$ 
     $in\_second \leftarrow \text{inside}[e]p[j + 1];$ 
    if  $in\_first \wedge in\_second$  then
       $q[k] \leftarrow p[j + 1];$ 
       $k \leftarrow k + 1;$ 
    end if
    if  $\neg in\_first \wedge in\_second$  then
       $r \leftarrow \text{intersect}[e](p[j], p[j + 1]);$ 
       $q[k] \leftarrow r;$ 
       $q[k + 1] \leftarrow p[j + 1];$ 
       $k \leftarrow k + 2;$ 
    end if
    if  $in\_first \wedge \neg in\_second$  then
       $r \leftarrow \text{intersect}[e](p[j], p[j + 1]);$ 
       $q[k] \leftarrow r;$ 
       $k \leftarrow k + 1;$ 
    end if
  end for
   $p \leftarrow q;$ 
end for
return  $q[1 \dots l]$ 

```

The only other significant feature is the independent indexing of the new polygon  $q$ . This is an essential part of the algorithm.

# GRAPHICS

## 3. Polygon Fill Algorithm

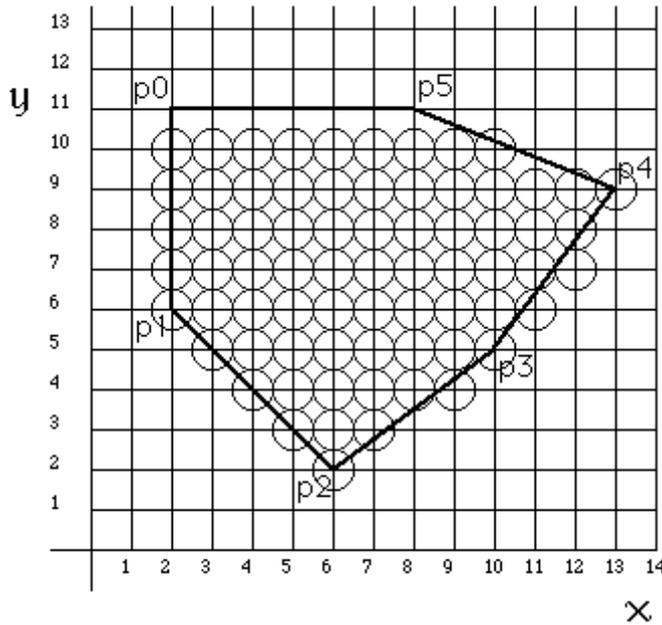


Figure: Polygon Fill algorithm

The graphics algorithms considered thus far have been *simple*, in the sense that only simple *Meta-IV* domain models have been used. I would now like to present another basic graphics algorithm, one which requires more complex domains, a polygon fill algorithm. From the illustration given above, it is absolutely clear what the algorithm produces and where it is ‘defective’. Quite apart from intrinsic interest, the example is ideally suited for the purpose of demonstrating how solutions to problems in the computer science domain involve the sort of data structures and algorithms that one is equally likely to need in other problem solving domains. In particular, the visual feedback may veritably be interpreted as specification animation and plays a rôle in computing in much the same way as graphs do in mathematical analysis. In other words, study of computer science structures and algorithms within the framework of computer graphics is pedagogically superior to more ‘conventional’ approaches. The polygon fill algorithm that I present has been reverse-engineered from a form of ML (Salmon and Slater 1987, 105–10).

### 3.1. The Build Phase

The scanline algorithm (Hearn and Baker 1986; Salmon and Slater 1987), based on the concept of **edge coherence**, makes use of a so-called edge table (*ET*) which may be specified by

$$ET = SCANLINE \xrightarrow{m} ET\_ENTRY^*$$

$$SCANLINE = \{0 \dots V\_DISP\_RES - 1\}$$

$$ET\_ENTRY :: y_{max}: SCANLINE \times x_{min}: \mathbf{R} \times \frac{dx}{dy}: \mathbf{R}$$

$$\text{for } mk-ET\_ENTRY(y_{max}, x_{min}, \frac{dx}{dy}) \text{ use } (y_{max}, x_{min}, \frac{dx}{dy})$$

where *V\_DISP\_RES* is the resolution of the display in the vertical direction (i.e., the number of scanlines). The algorithm consists of two phases—construction of the edge table (build) and generation of the appropriate scanlines (process). In this subsection I address the build phase.

First note, that in the abstract, the domain model has the form

$$MODEL = X \xrightarrow{m} Y^*$$

which may be interpreted as a reification of  $X \xrightarrow{m} \mathcal{P}Y$ , the classical model of relations. I do **not** mean, that from this, one is to infer that I might have been able to choose a more abstract model than the one presented. What I *do* wish to indicate is that operations on the model may be (re)used in any context where such a reification is invoked in other problem solving domains.

I present four versions of the algorithm for the build phase in order to illustrate some of the practical aspects of doing specifications in the Irish School of the VDM. To obtain *one* specification is not to be regarded as finding *the* solution. In general, an abstract naïve recursive form is obtained first—version 1. Then, from this, an efficient tail-recursive version is derived. The third version is a reification or evolution of the first version and it in turn is used to derive a tail-recursive form using version 2 as a guideline. The version 4 which is presented is the specification of the ‘real’ algorithm in tail-recursive form. Another practical aspect of the method is worth mentioning. I produced the algorithmic reifications and derivations first, satisfied myself as to their correctness, and only later considered the semantic domains. The amount of material generated would take a complete Chapter in itself. Perforce I am limited to present only a brief overview of the material.

# GRAPHICS

## 3.1.1. Build Version 1

In this simple naïve recursive version of the algorithm an edge table is constructed which contains only the information necessary to produce scanlines for the polygon in question.

```

build:  $DC\_POINT^* \longrightarrow ET$ 
build( $\langle z \rangle$ )  $\triangleq \theta$ 
build( $\langle z_1, z_2 \rangle \wedge \tau$ )  $\triangleq$ 
  let  $dx = x_2 - x_1, dy = y_2 - y_1, et = \text{build}(\langle z_2 \rangle \wedge \tau)$  in
     $y_1 < y_2 \rightarrow$ 
       $\chi[[y_1]]et$ 
       $\rightarrow et + [y_1 \mapsto \langle (y_2, x_1, \frac{dx}{dy}) \rangle \wedge et(y_1)]$ 
       $\rightarrow et \cup [y_1 \mapsto \langle (y_2, x_1, \frac{dx}{dy}) \rangle]$ 
     $y_1 = y_2 \rightarrow et$ 
     $y_1 > y_2 \rightarrow$ 
       $\chi[[y_2]]et$ 
       $\rightarrow et + [y_2 \mapsto \langle (y_1, x_2, \frac{dx}{dy}) \rangle \wedge et(y_2)]$ 
       $\rightarrow et \cup [y_2 \mapsto \langle (y_1, x_2, \frac{dx}{dy}) \rangle]$ 

```

Knowledge of the problem domain is necessary to understand the meaning of specification with respect to computer graphics. However, even at the purely formal level, much can be learned. Note that entry of information into the table is guarded by a boolean condition, the characteristic function for maps. I am using the abbreviation  $\chi[[y_1]]et \equiv \chi[[y_1]] \text{ dom } et$ . The guard is used to determine whether the override or the extend operator is to be employed. We recognise this as a standard way of writing map specifications: the file system, the bag, the dictionary, etc. In natural language terms, we are distinguishing between ‘enter’ (new information) and ‘update’ (existing information) operations.

## 3.1.2. Build Version 2

With experience of the method, I have found that the transformation from the naïve recursive form to tail-recursive form has gradually made it easier to produce tail-recursive specifications directly to solve problems. In fact, it is becoming more difficult to obtain the naïve recursive form first. I can not say whether this would be true in general for other users of the method. The efficient (i.e., tail-recursive) version derived from the previous specification is

```

build: DC_POINT* → (ET → ET)
build[[⟨z⟩]et] ≜ et
build[[⟨z1, z2⟩ ^ τ]et] ≜
  let dx = x2 - x1, dy = y2 - y1 in
    y1 < y2 →
      χ[[y1]et]
      → build[[⟨z2⟩ ^ τ](et + [y1 ↦ ⟨(y2, x1,  $\frac{dx}{dy}$ )⟩ ^ et(y1))]
      → build[[⟨z2⟩ ^ τ](et ∪ [y1 ↦ ⟨(y2, x1,  $\frac{dx}{dy}$ )⟩])
    y1 = y2 → build[[⟨z2⟩ ^ τ]et
    y1 > y2 →
      χ[[y2]et]
      → build[[⟨z2⟩ ^ τ](et + [y2 ↦ ⟨(y1, x2,  $\frac{dx}{dy}$ )⟩ ^ et(y2))]
      → build[[⟨z2⟩ ^ τ](et ∪ [y2 ↦ ⟨(y1, x2,  $\frac{dx}{dy}$ )⟩])

```

Note that the algorithm is invoked with an initially empty  $ET$ ,  $\text{build}[[p]\theta]$ . I would normally use a two-level specification to express this, as I have done frequently throughout the thesis. The first level gives the initialisation; the second level is the tail-recursive form. Before I present constraints on the ‘build’ specification, I would like to demonstrate that ‘build’ is essentially a bucket sort.

### 3.1.3. Bucket Sort

The bucket sort algorithm is the prototype for the edge table construction algorithm. In the version given in Aho, Hopcroft and Ullman (1974, 77), use is made of queues, called buckets, in which to store the result of distributing a sequence  $\sigma \in \mathbf{k}^* = \{0 \dots k - 1\}^*$ . Then the queues are concatenated to give the final result. The order in which such concatenation is performed is obviously essential. Let us model buckets as a map

$$\beta \in \text{BUCKETS} = \{0 \dots k - 1\} \xrightarrow{m} \mathbf{k}^*$$

Since maps do not have order, then the final part of the algorithm, in which buckets are concatenated, will seem to be quite complex. Note that the domain,  $\text{BUCKETS}$ , is isomorphic to that of the edge table. The sorting algorithm is given in the usual two-level specification form:

```

bucket_sort:  $\mathbf{k}^* \rightarrow \{0 \dots k - 1\}$ 
bucket_sort( $\sigma$ ) ≜ let  $\beta = \mathcal{B}[[\sigma]\theta]$  in concat[[ $\beta$ ]] $\Lambda$ 

```

## GRAPHICS

Note in particular the initialisation of the map. In practice, the form

$$\text{let } \beta = \mathcal{B}[\sigma][j \mapsto \Lambda \mid j \in \{0 \dots k - 1\}] \text{ in } \dots$$

is preferred, a form similar to which we will use in the next version of the edge table construction algorithm. Now let us consider the specification of  $\mathcal{B}$  which is given in tail-recursive form:

$$\begin{aligned} \mathcal{B}: \mathbf{k}^* &\longrightarrow (\text{BUCKETS} \longrightarrow \text{BUCKETS}) \\ \mathcal{B}[\langle a_j \rangle \wedge \tau] \beta &\triangleq \\ \chi[a_j] \beta & \\ \rightarrow \mathcal{B}[\tau](\beta + [a_j \mapsto \beta(a_j) \wedge \langle a_j \rangle]) & \\ \rightarrow \mathcal{B}[\tau](\beta \cup [a_j \mapsto \langle a_j \rangle]) & \end{aligned}$$

$$\mathcal{B}[\Lambda] \beta \triangleq \beta$$

Recall that this is structurally isomorphic to the addition of bags. Therefore, introducing the bucket concatenation operator,  $\otimes$ , defined in the usual way, the guard may be eliminated to give

$$\mathcal{B}[\langle a_j \rangle \wedge \tau] \beta \triangleq \mathcal{B}[\tau](\beta \otimes [a_j \mapsto \langle a_j \rangle])$$

I may use exactly the same operator to eliminate the characteristic function guards in the build algorithm, which is obviously a particular form of the  $\mathcal{B}$  algorithm just presented. For completeness, I now give the specification of the concatenation of the buckets:

$$\begin{aligned} \text{concat}: \text{BUCKETS} &\longrightarrow (\mathbf{k}^* \longrightarrow \mathbf{k}^*) \\ \text{concat}[\theta] \sigma &\triangleq \sigma \\ \text{concat}[\beta] \sigma &\triangleq \text{let } j \in \downarrow / \text{dom } \beta \text{ in } \text{concat}[\{j\} \triangleleft \beta](\sigma \wedge \beta(j)) \end{aligned}$$

Finally, it is worth observing that any reification which I might consider for the domain, *BUCKETS*, and the bucket sort algorithm, is applicable, *ceteris paribus*, to the edge table domain, to which I now return.

### 3.1.4. Build Version 3

A more faithful modelling of the scan-conversion process requires that there be edge table entries for every scanline in the range  $\{0 \dots V\_DISP\_RES - 1\}$ . Taking this factor into account, version 1 of the algorithm may be modified to give:

```

build: DC_POINT* → ET
build(⟨ z ⟩) ≜ [i ↦ Λ | 0 ≤ i ≤ V_DISP_RES - 1]
build(⟨ z1, z2 ⟩ ^ τ) ≜
  let dx = x2 - x1, dy = y2 - y1, et = build(⟨ z2 ⟩ ^ τ) in
    y1 < y2 → et + [y1 ↦ ⟨ (y2, x1, dx/dy) ⟩ ^ et(y1)]
    y1 = y2 → et
    y1 > y2 → et + [y2 ↦ ⟨ (y1, x2, dx/dy) ⟩ ^ et(y2)]
  
```

Note that the characteristic function guards have disappeared due to the initialisation. Such a modification brings the structure of the build algorithm more into line with that of the real bucket sort algorithm. The corresponding tail-recursive version may be obtained by an appropriate modification of version 2.

### 3.1.5. Build Version 4

In processing the edge table information, one is only interested in those entries generated by the polygon being considered. Thus, a further realistic improvement in the algorithm is the determination of the minimum and maximum scanline values needed. In other words, one must constrain the build algorithm. Introducing the auxiliary semantic domain:

$$SCANRANGE = \{SCANLINE \dots SCANLINE\}$$

with the obvious invariant

$$inv\text{-}SCANRANGE(sr) \triangleq sr \neq \emptyset$$

one then obtains what I shall consider as the final version, given in tail-recursive form:

## GRAPHICS

```

build: DC_POINT* → (ET × SCANRANGE → ET × SCANRANGE)
build[[⟨ z ⟩]](et, (y_min, y_max)) ≜ (et, (y_min, y_max))
build[[⟨ z1, z2 ⟩ ^ τ]](et, (y_min, y_max)) ≜
  let dx = x2 - x1, dy = y2 - y1 in
    y1 < y2 → build[[⟨ z2 ⟩ ^ τ]](et + [y1 ↦ ⟨ (y2, x1,  $\frac{dx}{dy}$ ) ⟩ ^ et(y1)],
      (y_min ↓ y1, y_max ↑ y2))
    y1 = y2 → build[[⟨ z2 ⟩ ^ τ]](et, (y_min, y_max))
    y1 > y2 → build[[⟨ z2 ⟩ ^ τ]](et + [y2 ↦ ⟨ (y1, x2,  $\frac{dx}{dy}$ ) ⟩ ^ et(y2)],
      (y_min ↓ y2, y_max ↑ y1))

```

where the build algorithm is invoked by:

```

build[[p]]([i ↦ Λ | 0 ≤ i ≤ V_DISP_RES - 1], (V_DISP_RES - 1, 0))

```

I could have produced this version directly from the reverse-engineering process without giving the other versions. Instead, I have chosen to exhibit the sort of choice that one normally makes in developing specifications. Each and every version given was a valid specification of the edge table construction. It is this final version of the algorithm, one which incorporates ‘design’ issues, which is used to generate the information required of the process algorithm in the next section.

### 3.2. The Process Phase

One may develop specification variants for the process phase, similar to those for the build phase. But since space is at a premium, I present a ‘final’ version which is directly obtained from the ML. Using the edge table one produces the fill scanlines with the assistance of an *active* edge table which is simply a sequence of edge table entries:

$$ACT\_ET = ET\_ENTRY^*$$

Then, the process algorithm is given by

```

process: ET × SCANRANGE → (ACT_ET → DC_LINE*)
process[[et, (y_max, y_max)]]aet ≜ Λ
process[[et, (i, y_max)]]aet ≜
  let aet' = sort((update[[i]]aet) ^ et(i)) in
    (joined_lines[[i]]aet') ^ process[[et, (i + 1, y_max)]]aet'

```

where *sort* arranges the edges in ascending order according to the *x*-coordinate, since filling takes place from left to right, i.e., from  $x_{min}$  to  $x_{max}$ . The update algorithm

is actually a generalised version of the ‘delete all’ endomorphism on sequences:

$$\begin{aligned}
& \text{update: } \text{SCANLINE} \longrightarrow (\text{ACT\_ET} \longrightarrow \text{ACT\_ET}) \\
& \text{update}[[i]]\Lambda \stackrel{\Delta}{=} \Lambda \\
& \text{update}[[i]](\langle (i, -, -) \rangle \wedge \tau) \stackrel{\Delta}{=} \text{update}[[i]]\tau \\
& \text{update}[[i]](\langle (y, x, \frac{dx}{dy}) \rangle \wedge \tau) \stackrel{\Delta}{=} \langle (y, \text{round}(x + \frac{dx}{dy}), \frac{dx}{dy}) \rangle \wedge \text{update}[[i]]\tau
\end{aligned}$$

In performing an update, the algorithm effectively uses a version of the simple DDA algorithm given in the first section of the Chapter. The final part of the process algorithm is a simple matter of drawing horizontal line segments:

$$\begin{aligned}
& \text{joined\_lines: } \text{SCANLINE} \longrightarrow (\text{ACT\_ET} \longrightarrow \text{DC\_LINE}^*) \\
& \text{joined\_lines}[[i]]\Lambda \stackrel{\Delta}{=} \Lambda \\
& \text{joined\_lines}[[i]](\langle (-, x_l, -), (-, x_r, -) \rangle \wedge \tau) \stackrel{\Delta}{=} \\
& \quad \langle [(x_l, i), (x_r, i)] \rangle \wedge \text{joined\_lines}[[i]]\tau
\end{aligned}$$

That concludes the specification of the basic polygon fill algorithm. There are many obvious refinements which should be carried out on the process phase. For example, one may wish to present a tail-recursive form for the process algorithm itself; the update and joined\_lines algorithms have interesting algebraic properties which may be exhibited using more abstract notation and some operator calculus. With respect to the entire fill algorithm, there are many directions for further development. One immediate simple generalisation is to extend it to process sequences of polygons rather than just a single polygon. I have already referred to the generalisation that converts the algorithm into a hidden surface algorithm (see Salmon and Slater 1987, 429–32), when I introduced the symmetric difference operator for sets in Chapter 2. Leaving aside the basic algorithms of computer graphics, I now present some material on the specification of curves and surfaces used in computer-aided design.

# GRAPHICS

## 4. Curves

There is space here to touch upon but a small portion of the specifications of curves and surfaces in computer-aided design. I shall limit myself to elementary material on Bézier curves and surface patches. For background material I might suggest Bézier's own work (1986) which has a firm geometrical flavour to it. This is definitely mathematically oriented in contradistinction to the algorithmic concerns of the computer scientist. But, it does contain a wealth of practical detail on real issues facing designers who use CAD systems. Naturally, most introductory computer graphics texts treat Bézier curves and surface patches, giving practical algorithmic details. Moreover, the CAD literature abounds with material on Bézier curves and surfaces. For a readable lucid account, I recommend the works by Piegl (1986), noting, in this particular context, his article on recursive algorithms.

A Bézier curve of degree  $n$  may be defined by

$$P_n(u) = \sum_{i=0}^n B_i^n(u) \mathbf{p}_i$$

where  $\mathbf{P}$  is the control point vector  $(\mathbf{p}_0, \dots, \mathbf{p}_i, \dots, \mathbf{p}_n)$ , each  $\mathbf{p}_i$  is a coordinate vector  $(x_{i1}, x_{i2}, \dots, x_{ij}, \dots, x_{im})$  over some space of dimension  $m$  and  $B_i^n(u)$ ,  $0 \leq i \leq n$ , are Bernstein basis functions:

$$B_i^n(u) = \binom{n}{i} (1-u)^{n-i} u^i$$

The parameter  $u$  ranges over the closed unit interval  $[0, 1]$ , i.e.,  $0 \leq u \leq 1$ . An illustration of a simple cubic curve is given on the following page. A note of explanation on the choice of the notation  $B_i^n(u)$  is probably required. It has been customary in texts dealing with the Bézier curves to use  $B_{i,n}(u)$ . However, I feel that the above is more expressive in so far as it calls to mind the fact that the very definition of  $B_i^n(u)$  involves the combinatorial function. Admittedly, to the unwary, the chosen notation might lead them to suppose that exponentiation is involved. All is to be interpreted in context.

Now the key to understanding the Bézier form is that *it is determined completely by its control point vector  $\mathbf{P}$* . This point can not be stressed enough and, indeed, may be said to be a keystone of any conceptual model of Bézier curves. The issue

will be highlighted in terms of the matrix representation now to be discussed below.

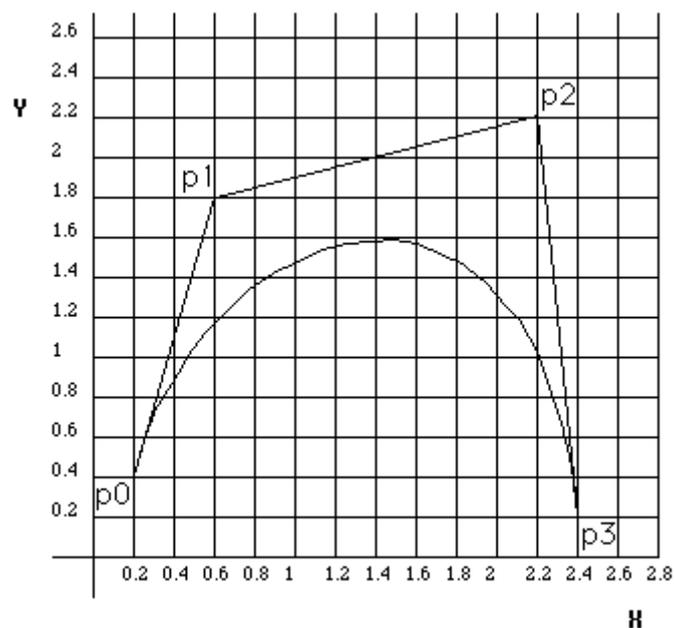


Figure: Bézier Cubic Curve

One might, of course, develop an algorithm to compute  $B_i^n(u)$  directly based on the form of the definition. By ‘directly based’ I mean that there is a sort of 1–1 correspondence between the explicit definition and the algorithm. This is precisely what the Second Edition of Newman and Sproull does (1979, 317), where they chose Pascal as the language of expression, remarking that it can be made more efficient. Indeed, further on in the text (p.327), they refer to the possibility of using Horner’s method for polynomial evaluation. With regard to same, they note that the method of forward differences may be used to avoid (expensive) multiplications. Finally, they also present the binary recursive subdivision algorithm for Bézier curves, noting at the same time, that it may be used for arbitrary polynomial forms since all such may be transformed into the Bézier basis. But, it must be stressed that conceptual models are often formed by examples and it is very likely that given the Pascal algorithm, most will adopt it as the norm, taking the route of least resistance, either encoding it directly if they are using Pascal or transliterating it into some other related imperative programming language. Although it clearly corresponds to the definition, in the sense that portions of the algorithm correspond with portions of the mathematical expression, one must note the conceptual distance between

## GRAPHICS

the succinctness of the mathematical definition and the size and verbosity of the Pascal code. Verification of correctness is likely to be very time consuming whereas implementation and testing would be straightforward. Needless to remark, as has already been indicated in Chapter 4, a naïve recursive approach would be equally disastrous.

The text by Foley and van Dam (1982) develops the Bézier curve from its matrix definition (presented below). With respect to an algorithm to evaluate a Bézier curve, they also give Horner's method for polynomial evaluation and details of the forward difference method. The recursive forms are not even mentioned.

Salmon and Slater (1987) present both Horner's method for polynomial evaluation (p.355) and the recursive subdivision algorithm (p.442). Interestingly, the latter is expressed in their form of the functional programming language ML.

The Bézier curve may be expressed in matrix terms as

$$P_n(u) = \mathbf{u}\mathbf{M}_b\mathbf{P}^t$$

$$= (u^n, \dots, u^i, \dots, 1) \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix} \begin{pmatrix} \mathbf{P}_0 \\ \vdots \\ \mathbf{P}_i \\ \vdots \\ \mathbf{P}_n \end{pmatrix}$$

where  $\mathbf{u}$  is the canonical basis vector,  $\mathbf{M}_b$  is the canonical Bézier matrix, and  $\mathbf{P}$  is the control point vector. One of the 'nice' features of the matrix representation is the dual representation of the Bézier curve both in terms of the Bernstein basis ( $\mathbf{u}\mathbf{M}_b$ ) and as an  $n$ th degree polynomial ( $\mathbf{M}_b\mathbf{P}^t$ ).

For a Bézier cubic, the matrix representation is

$$P_3(u) = (u^3, u^2, u, 1) \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{P}_0 \\ \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \end{pmatrix}$$

Now both the canonical basis vector  $\mathbf{u}$  and the matrix  $\mathbf{M}_b$  are invariants with respect to any fixed degree. It is only the control point vector which can determine the shape. Thus the remark above that the control point vector is the conceptual keystone.

There may be very good reasons to decide to use the matrix form for implementation, particularly on platforms that have matrix multiplication pipelines (c.f., the *Geometry Engine* project at Stanford University (Clark 1980)). Certainly, in the case of the cubic, one of the most important design curves, it is fortunate that the

matrix is of dimension  $4 \times 4$  and that in 3-D graphics the use of homogeneous coordinates also leads to  $4 \times 4$  matrices. Thus, processing may be carried out uniformly on  $4 \times 4$  multipliers.

#### 4.1. Recursive Definition of Bezier Curves

The recursive nature of the combinatorial function in the definition of the basis function leads immediately to a recursive definition of the Bézier curve. That is, using the recursive definition

$$\binom{n}{i} = \binom{n-1}{i} + \binom{n-1}{i-1}, \quad 1 \leq i \leq n$$

$$\binom{n}{0} = 1$$

one may readily develop the form:

$$\begin{aligned} P_n(u) &= (1-u) \sum_{i=0}^{n-1} B_i^{n-1}(u) \mathbf{p}_i + u \sum_{i=0}^{n-1} B_i^{n-1}(u) \mathbf{p}_{i+1} \\ &= \sum_{i=0}^{n-1} B_i^{n-1}(u) \left( (1-u) \mathbf{p}_i + u \mathbf{p}_{i+1} \right) \end{aligned}$$

But what exactly is to be understood by the recursive definition? Recalling the fact that the control points themselves determine the Bézier curve, it is clear that the recursive definition is essentially a statement about the transformation of the control point vector  $\mathbf{P}$ . This may be highlighted by introducing the linear interpolating function  $\mathcal{L}_u: (\mathbf{a}, \mathbf{b}) \mapsto (1-u)\mathbf{a} + u\mathbf{b}$ . It may be shown that  $\mathcal{L}_u$  is a linear operator over  $n$ -dimensional vector spaces. Then, one has

$$P_n(u) = \sum_{i=0}^{n-1} B_i^{n-1}(u) \left( \mathcal{L}_u(\mathbf{p}_i, \mathbf{p}_{i+1}) \right)$$

and the transformation may be expressed diagrammatically by

$$\begin{array}{c} (\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_i, \mathbf{p}_{i+1}, \dots, \mathbf{p}_{n-1}, \mathbf{p}_n) \\ \Downarrow \\ (\mathcal{L}_u(\mathbf{p}_0, \mathbf{p}_1), \mathcal{L}_u(\mathbf{p}_1, \mathbf{p}_2), \dots, \mathcal{L}_u(\mathbf{p}_i, \mathbf{p}_{i+1}), \dots, \mathcal{L}_u(\mathbf{p}_{n-1}, \mathbf{p}_n)) \end{array}$$

Now, if one defines the ‘reduction’ function  $F//$  over sequences:

$$\begin{aligned} F//\langle a, b \mid \tau \rangle &= \langle F(a, b) \rangle \wedge F//\langle b \mid \tau \rangle \\ F//\langle a, b \rangle &= \langle F(a, b) \rangle \end{aligned}$$

## GRAPHICS

then the control point vector transformation may be expressed simply by  $\mathcal{L}_u//\mathbf{P}$  and the required recursive algorithm is obtained by repeatedly applying the transformation operator  $\mathcal{L}_u//$  to give:

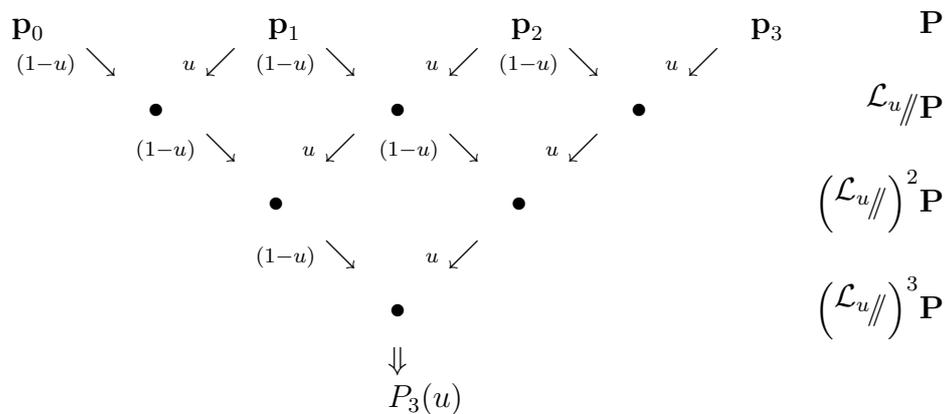
$$P_n(u) = \underbrace{\mathcal{L}_u \circ \mathcal{L}_u \circ \dots \circ \mathcal{L}_u}_{n \text{ times}} // \mathbf{P} = \left(\mathcal{L}_u//\right)^n \mathbf{P}$$

where  $n = \text{len } \mathbf{P} - 1$  is the degree of the Bézier curve.

It is essential to note that this **is** an alternative (algorithmic) way of defining the Bézier curve! Indeed, it is simply the *de Casteljau algorithm* to be discussed in the following subsection.

### 4.2. de Casteljau Algorithm

The *de Casteljau* algorithm, as expressed as a sequence of control point vector transformations, is simple to grasp diagrammatically, as illustrated here for the cubic:



Modern authors such as DeRose (1988) and Piegl (1986) give index-recursive definitions of the *de Casteljau* algorithm, a form which may be regarded as ‘standard’ in the literature. Moreover, the passage from such index-recursive forms to imperative forms is immediate. However, here it is shown that an elegant (and efficient) natural index-free tail-recursive algorithm that reflects the nature of the process of repeated control point vector transformation is obtained immediately upon consideration of the remark ‘sequence of control point vector transformations’. Effectively, the last element of the sequence

$$\langle \mathcal{L}_u//p, \left(\mathcal{L}_u//\right)^2 p, \left(\mathcal{L}_u//\right)^3 p, \dots, \left(\mathcal{L}_u//\right)^n p \rangle$$

is the sought-for result. This may be expressed recursively by

$$\begin{aligned}\alpha[\mathcal{L}_u]p &\triangleq \alpha[\mathcal{L}_u]\mathcal{L}_u//p \\ \alpha[\mathcal{L}_u]\langle e \rangle &\triangleq e\end{aligned}$$

where  $\alpha$  denotes application of the linear operator  $\mathcal{L}_u$  to the sequence of control points  $p$ .

There are a few points worth noting here. The number of applications of the transformation  $\mathcal{L}_u$  to the control point vector is governed completely by the degree of the Bézier curve and hence by the length of the control point vector itself. Since a vector of length 1 denotes a curve of degree zero—a point, then this determines the stopping condition, i.e., the base case. There is no need to count! Finally, note that instead of returning this one element vector, I have chosen to return the element itself, which is the usual requirement in the algorithm. More importantly, the algorithm may be directly transliterated into PROLOG to give an efficient practical algorithm for CAD.

#### 4.2.1. PROLOG Implementation

To illustrate the efficiency of the recursive *de Casteljau* algorithm, I present the equivalent in PROLOG. For clarity, the  $\alpha$  operator has been renamed `de_casteljau`. Although the encoding is verbose and lengthy compared with the specification of the algorithm given, it at least has the merit of accurately reflecting the structure of that algorithm.

```
de_casteljau(L, U, [E], E).
de_casteljau(L, U, P, Result) :-
    Function =.. [reduce, L, U, P, P1],
    call(Function),
    de_casteljau(L, U, P1, Result).
```

where the reduce function,  $L_u//$ , is given by

```
reduce(L, U, [E], E).
reduce(L, U, [A, B | Tau], [Luab | Rest]) :-
    Function =.. [L, U, A, B, Luab],
    call(Function),
    reduce(L, U, [B | Tau], Rest).
```

## GRAPHICS

and the linear interpolating function  $\mathcal{L}[u]$  is

```
linear_inter(U, θ, θ, θ).
linear_inter(U, [A | RestA], [B | RestB], [AB | Rest]) :-
    AB is (1 - U) * A + U * B,
linear_inter(U, RestA, RestB, Rest).
```

The algorithm is invoked thus

```
?- de_casteljau(linear_inter, 0.5, [[0, 0], [1, 2.4], [2, 2], [2.8, 0]], Result)
```

with  $Result = [1.475, 1.65]$ .

Note that in the definition of `de_casteljau`, the actual predicate `reduce` is used directly. One could have made the algorithm more generic by adding an extra parameter, say  $R$ , to the clause to allow for the possibility of a different instantiation.

### 4.2.2. Imperative Encoding

It will now be demonstrated that it is possible to derive the ‘standard’ imperative encoding found in the literature, by successive refinements. From either the mathematical definition

$$P_n(u) = (\mathcal{L}_u//)^n \mathbf{P}, \quad n = \deg(\mathbf{P})$$

or the recursive definition

$$\begin{aligned} \alpha[\mathcal{L}_u]p &\triangleq \alpha[\mathcal{L}_u](\mathcal{L}_u//)p \\ \alpha[\mathcal{L}_u]\langle e \rangle &\triangleq e \end{aligned}$$

an imperative form may be derived:

- 1) The important point to note is that the  $\alpha[\mathcal{L}_u]$  essentially forces  $\mathcal{L}_u//$  to be applied to  $p$  exactly  $n$  times, where  $n$  is the degree of the polynomial. Indeed it was precisely this observation that led to the development of the recursive algorithm in the first place! This information may be explicitly (and redundantly) incorporated thus:

$$\begin{aligned} \alpha[\mathcal{L}_u, \deg(p)]p &\triangleq \alpha[\mathcal{L}_u, \deg(p) - 1](\mathcal{L}_u//)p \\ \alpha[\mathcal{L}_u, 0]\langle e \rangle &\triangleq e \end{aligned}$$

2) Such a recursive form immediately gives rise to the skeletal imperative form

```

 $q \leftarrow p;$ 
for  $iter = [1 \dots \deg(q)]$  do
   $q \leftarrow (\mathcal{L}_u//)q;$ 
end for
return some element of  $q$ 

```

where  $q$  is overwritten in some manner yet to be determined.

3) The next stage in the process is to consider ' $q \leftarrow \mathcal{L}_u//q$ ';'. Let us introduce an auxiliary result vector  $r$ . Then using the slice notation  $[1 \dots \deg(q) + 1]$  and with a view to overwriting  $q$  on the left, the definition of  $\mathcal{L}_u//$  be written as

$$\begin{aligned} (\mathcal{L}_u//)q[i \dots n] &= r[i \mapsto \mathcal{L}_u(q[i], q[i + 1])] \cup (\mathcal{L}_u//)q[i + 1 \dots n] \\ (\mathcal{L}_u//)q[n - 1 \dots n] &= r[n - 1 \mapsto \mathcal{L}_u(q[n - 1], q[n])] \end{aligned}$$

where the algorithm is understood to be invoked with  $i = 1$ .

A sequence of length  $n$  may always be considered to be a map from  $\{1 \dots n\}$  to the elements of the sequence. In particular, the  $j$ th element of a sequence  $\sigma$  may be denoted  $\sigma_j$ , or equivalently  $[j \mapsto \sigma(j)]$ . Recall that the binary operator  $\cup$  denotes map extension which is defined by

$$m_1 \cup m_2 \triangleq \{[d \mapsto r] \mid r = m_1(d) \vee r = m_2(d), \text{dom } m_1 \cap \text{dom } m_2 = \emptyset\}$$

Note also that indexing is always assumed to start at 1. Admittedly, it is easier to obtain the imperative form if an indexing origin of 0 is chosen. However, it is informative to show the refinement within the context of such an indexing constraint. Clearly, one has the transform:

$$\mathcal{L}_u//: q[i \dots n] \mapsto r[i \dots n - 1]$$

and, in general, after  $s$  iterations:

$$(\mathcal{L}_u//)^s: q[i \dots n] \mapsto r[i \dots n - s]$$

In other words, the index range after  $s$  iterations is  $[i \dots n - s]$ . But, since we have  $\deg(q[i \dots n]) = n - i$ , then  $n = \deg(q[i \dots n]) + i$ . Hence, the index range may be written in the form  $[i \dots \deg(q) + i - s]$ . This gives us the imperative form

## GRAPHICS

```

q ← p;
for iter = [1 ... deg(q)] do
  for i = [1 ... deg(q) + 1 - iter] do
    r[i] ←  $\mathcal{L}_u(q[i], q[i + 1])$ ;
  end for
end for
return r[1]

```

- 4) Finally, to optimise this encoding, we shall eliminate the auxiliary  $r$  and overwrite  $q$ .

```

q ← p;
for iter = [1 ... deg(q)] do
  for i = [1 ... deg(q) + 1 - iter] do
    q[i] ←  $\mathcal{L}_u(q[i], q[i + 1])$ ;
  end for
end for
return q[1]

```

In practice, most CAD programmers would only be interested in the final result—an efficient imperative algorithm. But there are sound theoretical and pedagogical reasons why the passage of development from specification to encoding should be given in ‘excruciating’ detail. One obvious reason is the search for more efficient algorithms. Another concerns reuse, a principle to be illustrated below in consideration of the subdivision algorithm.

### 4.3. Construction of the Polyline

Heretofore, one has focused exclusively on the computation of a point  $P(u)$  on the Bézier curve. From the perspective of computer graphics, it is necessary to compute an appropriate polyline or segment chain. Given the requirement that the segment chain shall have a specific mesh, then an appropriate algorithm may be specified as follows:

```

build_curve: MESH → CURVE → POLYLINE
build_curve[[mesh]]p ≜
  let u = 0, step = 1/mesh in    -- mesh ≠ 0
    build[[u, step]]p

```

The build algorithm is specified by

$$\begin{aligned}
&\text{build: } \mathbf{R} \times \mathbf{R} \longrightarrow \text{CURVE} \longrightarrow \text{POLYLINE} \\
&\text{build}[[u, \textit{step}]]\mathbf{p} \triangleq \\
&\quad \text{let } n = \text{deg}(\mathbf{p}) \text{ in} \\
&\quad \quad \langle (\mathcal{L}_u //)^n \mathbf{p} \rangle \wedge \text{build}[[u + \textit{step}, \textit{step}]]\mathbf{p} \\
&\text{build}[[1, \textit{step}]]\mathbf{p} \triangleq \langle (\mathcal{L}_1 //)^n \mathbf{p} \rangle
\end{aligned}$$

where  $\wedge$  denotes sequence concatenation.

It is important to note that the above is a specification. In particular, exact real arithmetic is used and thus termination of the build algorithm is clearly guaranteed. Naturally, in the development path toward an executable algorithm, the refinement of the specification must give due concern to termination. In addition, the structural form of the recursive specification is isomorphic to other recursive algorithms, such as those for the simple DDA and Bresenham line drawing algorithms! Such is the expressive power of abstract specifications!

#### 4.3.1. A Refinement

From the specification, it is possible to derive an imperative algorithm intuitively. Indeed, such intuition rapidly comes through experience with application of the Arzac and Kodratoff method in transforming recursive algorithms to tail-recursive algorithms to imperative algorithms using while loops (Arzac and Kodratoff 1982):

$$\begin{aligned}
&\text{build\_curve}[[\textit{mesh}]]\mathbf{p} \triangleq \\
&\quad u \leftarrow 0; \\
&\quad \textit{step} \leftarrow 1/\textit{mesh}; \\
&\quad i \leftarrow 1; \\
&\quad \text{while } u < 1 + \textit{step}/2 \text{ do} \\
&\quad \quad pl[i] \leftarrow (\mathcal{L}_u //)^m \mathbf{p}; \\
&\quad \quad i \leftarrow i + 1; \\
&\quad \quad \textit{step} \leftarrow u + \textit{step}; \\
&\quad \text{end while}
\end{aligned}$$

The real arithmetic termination is handled by the boolean condition  $u < 1 + \textit{step}/2$ , which is that used in the presentation of the teapot wireframe algorithm (Crow 1987). But, it is obvious that an exact terminating algorithm may be obtained by focusing on the indexed result—the polyline. Hence, the chosen refinement strategy is to introduce a polyline slice into the specification:

## GRAPHICS

```

build_curve: MESH → CURVE → POLYLINE_SLICE
build_curve[mesh]p ≜
  let u = 0, step = 1/mesh in    - - - mesh ≠ 0
  let pl[1 . . . mesh + 1] ∈ POLYLINE_SLICE in
    build[p, u, step]pl[1 . . . mesh + 1]

```

and the build algorithm is

```

build: CURVE × R × R → POLYLINE_SLICE → POLYLINE_SLICE
build[p, u, step]pl[i . . . mesh + 1] ≜
  let n = deg(p) in
    pl[i ↦ (L_u//)^n p] ∪ build[p, u + step, step]pl[i + 1 . . . mesh + 1]
build[p, u, step]pl[mesh + 1] ≜
  let n = deg(p) in
    pl[mesh + 1 ↦ (L_u//)^n p]

```

We now have a tail-recursive algorithm which may be transliterated into PROLOG to give an efficient executable algorithm. Moreover it is immediately translatable into a for loop.

### 4.3.2. Imperative Encoding

There is not much more that need be said about the transformation except, of course, to note the form in which the *de Casteljau* algorithm is given in the body of the loop.

```

build_curve[mesh]p ≜
  u ← 0;
  step ← 1/mesh;
  for i = [1 . . . mesh + 1] do
    pl[i] ← (L_u//)^n p;
    u ← u + step;
  endfor
  return pl[1 . . . mesh + 1]
  where n = deg(p)

```

Finally, for those who want some optimisation and, in particular, are fussy about guaranteeing that the polyline starts and ends at the first and last control points, one may present the final variant:

```

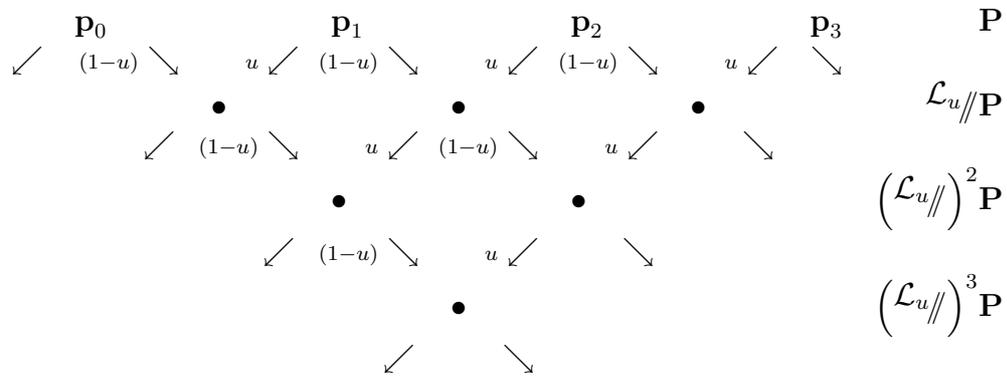
build_curve[[mesh]]p  $\triangleq$ 
  pl[1]  $\leftarrow$  p[1];
  step  $\leftarrow$  1/mesh;
  u  $\leftarrow$  step;
  for i = [2... mesh] do
    pl[i]  $\leftarrow$  ( $\mathcal{L}_u$ //)np;
    u  $\leftarrow$  u + step;
  endfor
  pl[mesh + 1]  $\leftarrow$  p[n + 1];
  return pl[1... mesh + 1]
  where n = deg(p)

```

The build\_curve algorithm has a significance above and beyond its immediate application. This will be illustrated in the context of the construction of the  $u$  and  $v$  curves of a Bézier surface.

#### 4.4. The Subdivision Algorithm

The *de Casteljau* algorithm may be modified to give the recursive subdivision algorithm for Bézier curves. It may be pictured diagrammatically as follows



A recursive algorithm which does the job may be specified by

$$\text{subdiv}[\mathcal{L}_u]p \triangleq \langle \text{first } p \rangle \wedge \left( \text{subdiv}[\mathcal{L}_u]\mathcal{L}_u//p \right) \wedge \langle \text{last } p \rangle$$

$$\text{subdiv}[\mathcal{L}_u]\langle e \rangle \triangleq \langle e \rangle \wedge \langle e \rangle$$

Introduction of the resulting sequences  $q$  and  $r$  leads to the index-free tail-recursive algorithm

$$\text{subdiv}[\mathcal{L}_u, p](q, r) \triangleq \text{subdiv}[\mathcal{L}_u, \mathcal{L}_u//p] \left( \langle \text{first } p \rangle \wedge q, r \wedge \langle \text{last } p \rangle \right)$$

$$\text{subdiv}[\mathcal{L}_u, \langle e \rangle](q, r) \triangleq (\langle e \rangle \wedge q, r \wedge \langle e \rangle)$$

## GRAPHICS

It is customary to choose the parameter  $u = 0.5$ . Now the significance of the recursive form lies in its obvious symmetry. This may be exploited in the following manner. Let  $q$  and  $r$  denote the ‘left hand’ and ‘right hand’ curves to be generated. Then there are three possible choices of development:

- 1) Copy  $p$  into  $r$ ; overwrite  $r$  on the left to generate the ‘right hand’ curve, as for the *de Casteljau* algorithm, and generate a new ‘left hand’ control point vector  $q$ .
- 2) Copy  $p$  into  $q$ ; overwrite  $q$  on the right to generate the ‘left hand’ curve, which requires a modification of the *de Casteljau* algorithm, and generate a new ‘right hand’ control point vector  $r$ . This is the choice made by Piegl (1986).
- 3) Refine the recursive algorithm to generate both ‘left hand’ and ‘right hand’ control point vectors in a single array  $qr$  and then choose either of the above options. This produces a sequence  $qr[1 \dots 2m]$ , where  $m = \deg(q) + 1$ , such that  $qr[1 \dots m]$  gives the ‘left-hand’ curve, and  $qr[m + 1 \dots 2m]$  the ‘right-hand’ curve. This particular approach is not discussed further here.

### 4.4.1. Imperative Encodings

The choice of option 1) gives the ‘cleanest’ approach to the development of the imperative algorithm, in the sense that it is a direct reuse of the *de Casteljau* algorithm. It is simply a matter of saving the first element of  $r$  at each iteration before it is overwritten:

```
 $r \leftarrow p;$   
for  $iter = [1 \dots \deg(r)]$  do  
   $q[iter] \leftarrow r[iter];$   
  for  $i = [1 \dots \deg(r) + 1 - iter]$  do  
     $r[i] \leftarrow \mathcal{L}_u(r[i], r[i + 1]);$   
  end for  
end for  
return  $q[1 \dots \deg(q)], r[1 \dots \deg(r)]$ 
```

In developing the imperative encoding for option 2) above and ignoring for the moment the issue of the generation of the new ‘right hand’ control point vector, one has immediately:

```

 $q \leftarrow p;$ 
for  $iter = [1 \dots \deg(q)]$  do
  for  $i = [\deg(q) + 1 \dots iter + 1]$  do
     $q[i] \leftarrow \mathcal{L}_u(q[i - 1], q[i]);$ 
  end for
end for

```

Note carefully the modification of the *de Casteljau* algorithm needed to obtain overwriting on the right. The only matter left to resolve is the issue of saving the  $n$ th value at each iteration:

```

 $r[\deg(q) + 1] \leftarrow q[\deg(q) + 1];$ 
for  $iter = [1 \dots \deg(q)]$  do
  for  $i = [\deg(q) + 1 \dots iter + 1]$  do
     $q[i] \leftarrow \mathcal{L}_u(q[i - 1], q[i]);$ 
  end for
   $r[\deg(q) + 1 - iter] \leftarrow q[\deg(q) + 1];$ 
end for
return  $q[1 \dots \deg(q)], r[1 \dots \deg(r)]$ 

```

It would prove to be a useful exercise to consider option 3) and apply the techniques described herein.

#### 4.5. Derivative of the Bézier Curve

It has already been stated that  $(\mathcal{L}_u//)^n \mathbf{P}$  is an alternative algorithmic *definition* of the Bézier curve. In this subsection, the claim that this definition may be used to establish various well-known properties of the Bézier curve is demonstrated. Again, to be brief, attention is focused solely on the derivatives. Starting from the algorithmic definition

$$P_n(u) = (\mathcal{L}_u//)^n \mathbf{P}$$

differentiation with respect to the parameter  $u$  gives

$$\begin{aligned} \frac{d}{du} P_n(u) &= n (\mathcal{L}_u//)^{n-1} \frac{d}{du} (\mathcal{L}_u//) \\ &= n (\mathcal{L}_u//)^{n-1} (\Delta//) \mathbf{P} \end{aligned}$$

Clearly,  $(\Delta//) \mathbf{P}$  is the control point vector of a Bézier curve of degree  $n - 1$ . Each control point of same is evidently a direction vector and the curve is in the space of tangent vectors to the original.

## GRAPHICS

*Proof:* First, observe that the differential of the linear interpolation operator  $\mathcal{L}_u$  is the difference operator  $\Delta$ , i.e.,

$$\frac{d}{du}\mathcal{L}_u = \Delta$$

This may be established simply as follows:

$$\begin{aligned}\frac{d}{du}\mathcal{L}_u(a, b) &= \frac{d}{du}\left((1-u)a + ub\right) \\ &= (-1)a + (1)b \\ &= \Delta(a, b)\end{aligned}$$

Then, it may be easily shown by structural induction that

$$\frac{d}{du}(\mathcal{L}_u//) = \Delta//$$

First, the differential of the base equation gives

$$\begin{aligned}\frac{d}{du}\mathcal{L}_u//\langle a, b \rangle &= \frac{d}{du}\left(\langle \mathcal{L}_u(a, b) \rangle\right) \\ &= \left(\langle \frac{d}{du}\mathcal{L}_u(a, b) \rangle\right) \\ &= \left(\langle \Delta(a, b) \rangle\right) \\ &= \Delta//\left(\langle \mathcal{L}_u(a, b) \rangle\right)\end{aligned}$$

and for the recursive equation, one has

$$\begin{aligned}\frac{d}{du}\mathcal{L}_u//\langle a, b \mid \tau \rangle &= \frac{d}{du}\left(\langle \mathcal{L}_u(a, b) \rangle \wedge \mathcal{L}_u//\langle b \mid \tau \rangle\right) \\ &= \langle \frac{d}{du}\mathcal{L}_u(a, b) \rangle \wedge \frac{d}{du}\mathcal{L}_u//\langle b \mid \tau \rangle \\ &= \langle \Delta(a, b) \rangle \wedge \Delta//\langle b \mid \tau \rangle, \quad \text{-- by induction} \\ &= \Delta//\langle a, b \mid \tau \rangle\end{aligned}$$

This establishes the basic theorem for the differentiation of the  $(\mathcal{L}_u//)^n$  operator:

**THEOREM 7.1.**

$$\begin{aligned}\frac{d}{du}(\mathcal{L}_u//)^n &= n(\mathcal{L}_u//)^{n-1} \frac{d}{du}(\mathcal{L}_u//) \\ &= n(\mathcal{L}_u//)^{n-1} \Delta//\end{aligned}$$

Other interesting theorems include 1) the commutativity of  $\mathcal{L}_u//$  and  $\mathcal{L}_v//$ , 2) the commutativity of  $\mathcal{L}_u//$  and  $\Delta//$ , and 3)  $\mathcal{L}_{u+h}// = \mathcal{L}_u// + h\Delta//$ . The passage to a tail-recursive index-free algorithm is then immediate:

$$\text{derivative}[\Delta, u]p \triangleq \text{let } n = \text{deg}(p) \text{ in } n \times \alpha[\mathcal{L}_u//](\Delta//)p$$

This may, of course be generalised to give an algorithm that computes the  $m$ th derivative:

$$\frac{d^m}{du} P_u(n) = n(n-1) \dots (n-m+1) (\mathcal{L}_{u//})^{n-m} (\Delta//)^m \mathbf{P}$$

In converting this to a tail-recursive algorithm, it is important to note that the recursion will be controlled by two factors: the order  $m$  of the derivative sought, and the degree  $n$  of the polynomial. Apart from the expression  $n(n-1) \dots (n-m+1)$ , it is clear that the recursive form is the iterated application of  $\Delta//$ , followed by the iterated application of  $\mathcal{L}_{u//}$ . Since the multiplication factor is order  $m$ , it seems reasonable to tie in its computation with that of  $(\Delta//)^m$ . Thus, let us define another extension of the  $\alpha$  operator by

$$\alpha[\Delta, n, m]\sigma \triangleq n \times \alpha[\Delta, n-1, m-1](\Delta//)\sigma$$

$$\alpha[\Delta, n, 0]\sigma \triangleq \sigma$$

Then, the specification of the  $m$ th derivate may be written in the form:

$$\text{derivative}[\Delta, u, m]p \triangleq \text{let } n = \text{deg}(p) \text{ in } \alpha[\mathcal{L}_u] \circ \alpha[\Delta, n, m]p$$

From the earlier discussion of previous algorithms, it ought to be clear how one now proceeds to an imperative encoding.

#### 4.5.1. Imperative Encoding

First, consider the computation of  $n(n-1) \dots (n-m+1)$ . One may construct the recursive function

$$f(n, m) = n \times f(n-1, m-1)$$

$$f(n, 1) = n$$

and obtain, by the Arzac and Kodratoff method, the tail- recursive form:

$$f[[n, m]]u = f[[n-1, m-1]](u \times n)$$

$$f[[n, 1]]u = u \times n$$

where the relation between  $f(n, m)$  and  $f[[n, m]]$  is given by

$$f[[n, m]]u = u \times f(n, m)$$

$$f(n, m) = f[[n, m]]1$$

and from which follows immediately the imperative encoding

## GRAPHICS

```
 $u \leftarrow 1;$   
while  $m \neq 1$  do  
   $u \leftarrow u \times n;$   
   $n \leftarrow n - 1;$   
   $m \leftarrow m - 1;$   
end while  
return  $u$ 
```

The while loop may be replaced by a corresponding for loop:

```
 $u \leftarrow 1;$   
for  $i = [1 \dots m]$  do  
   $u \leftarrow u \times n;$   
   $n \leftarrow n - 1;$   
end for  
return  $u \times n$ 
```

Now, considering the imperative encoding of  $(\Delta//)^m$  gives at once

```
for  $iter = [1 \dots m]$  do  
  for  $i = [1 \dots \deg(p) + 1 - iter]$  do  
     $p[i] \leftarrow \Delta(p[i], p[i + 1]);$   
  end for  
end for
```

Combining the two encodings gives

```
 $u \leftarrow 1;$   
for  $iter = [1 \dots m]$  do  
   $u \leftarrow u \times n;$   
   $n \leftarrow n - 1;$   
  for  $i = [1 \dots \deg(p) + 1 - iter]$  do  
     $p[i] \leftarrow \Delta(p[i], p[i + 1]);$   
  end for  
end for  
where  $u$  and  $p[1 \dots \deg(p) - m + 1]$  are available for further computation.
```

Finally, the de Casteljaou algorithm is applied to this result:

```

for  $iter = [1 \dots \deg(p) - m]$  do
  for  $i = [1 \dots \deg(p) - m + 1 - iter]$  do
     $p[i] \leftarrow \mathcal{L}_u(p[i], p[i + 1]);$ 
  end for
end for
return  $u \times p[1]$ 

```

Curves generalise naturally to surfaces. It will come as no surprise to find that *Meta-IV* curve specifications generalise to *Meta-IV* surface specifications. I conclude this Chapter with a very brief treatment of Bézier surfaces.

## 5. Bézier Surfaces

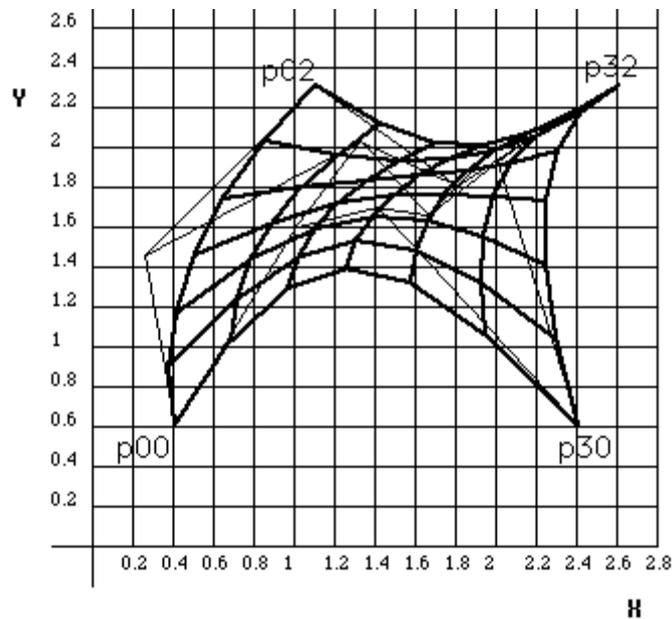


Figure: Bézier Cubic-Quadratic Surface Patch

A tensor product Bézier surface patch of degree  $n \times m$  may be defined (expressed) in the form of the double sum (Piegl 1986):

$$P(u, v) = \sum_{i=0}^n \sum_{j=0}^m B_i^n(u) B_j^m(v) \mathbf{p}_{ij}$$

## GRAPHICS

where the  $u$  and  $v$  boundary curves are of degree  $n$  and  $m$ , respectively. This may be written in the form

$$P(u, v) = \sum_{j=0}^m B_j^m(v) \mathbf{p}_j(u)$$

where

$$\mathbf{p}_j(u) = \sum_{i=0}^n B_i^n(u) \mathbf{p}_{ij}$$

To present a formal specification of the Bézier surface patch of degree  $n \times m$ , we will identify the patch with its tensor of control points  $\mathbf{P}$  of dimension  $n'm'$ ,  $n' = n + 1$ , and  $m' = m + 1$ :

$$\mathbf{P} = \begin{pmatrix} \mathbf{P}_{00} & \mathbf{P}_{01} & \dots & \mathbf{P}_{0j} & \dots & \mathbf{P}_{0m} \\ \mathbf{P}_{10} & \mathbf{P}_{11} & \dots & \mathbf{P}_{ij} & \dots & \mathbf{P}_{1m} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \mathbf{P}_{i0} & \mathbf{P}_{i1} & \dots & \mathbf{P}_{ij} & \dots & \mathbf{P}_{im} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \mathbf{P}_{n0} & \mathbf{P}_{n1} & \dots & \mathbf{P}_{nj} & \dots & \mathbf{P}_{nm} \end{pmatrix}$$

where the boundary curves are given by

$$\begin{aligned} \mathbf{P}(u, 0) &= (\mathbf{P}_{00}, \mathbf{P}_{10}, \dots, \mathbf{P}_{i0}, \dots, \mathbf{P}_{n0}) \\ \mathbf{P}(u, 1) &= (\mathbf{P}_{0m}, \mathbf{P}_{1m}, \dots, \mathbf{P}_{im}, \dots, \mathbf{P}_{nm}) \\ \mathbf{P}(0, v) &= (\mathbf{P}_{01}, \mathbf{P}_{02}, \dots, \mathbf{P}_{0j}, \dots, \mathbf{P}_{0m}) \\ \mathbf{P}(1, v) &= (\mathbf{P}_{n0}, \mathbf{P}_{n1}, \dots, \mathbf{P}_{nj}, \dots, \mathbf{P}_{nm}) \end{aligned}$$

Note carefully, that the tensor columns correspond to the control points for the  $u$ -curves. This is one of the ‘peculiarities’ of the tensor-product form. In the specification we will adhere to the ‘more natural’ notion of the tensor as sequence of rows, where each row specifies the control points of the  $u$ -curves. Thus, the ‘natural’ representation of a matrix as the sequence of rows, suggests that the appropriate semantic domains are:

$$\begin{aligned} \text{PATCH} &= \text{Tensor} \\ \text{Tensor} &= \text{Row}^* \\ \text{Row} &= \text{Point}^* \end{aligned}$$

where the invariant is given by

$$\begin{aligned} \text{inv-Tensor}: \text{Tensor} &\longrightarrow \mathbf{B} \\ \text{inv-Tensor}(t) &\triangleq (= /) \circ \text{len}^*(t) \wedge (\downarrow /) \circ \text{len}^*(t) \geq 2 \wedge \text{len}(t) \geq 2 \end{aligned}$$

Here,  $F/\sigma$  denotes reduction of a sequence  $\sigma$  by a binary operator  $F$  in the sense of APL. The application of  $F$  to every element of a sequence is denoted by  $F^*\sigma$ . Finally,  $\downarrow/\sigma$  denotes the minimum element of a sequence  $\sigma$ .

Now, application of the *de Casteljau* algorithm to the Bézier surface patch forms gives

$$\mathbf{p}_j(u) = \left(\mathcal{L}_u\right)^n \mathbf{P}_j$$

where  $\mathbf{P}_j$  denotes the  $j$ th row of the tensor  $\mathbf{P}$  and again

$$P(u, v) = \left(\mathcal{L}_v\right)^m \left(\left(\mathcal{L}_u\right)^n\right)^* \mathbf{P}$$

where the tensor is represented as a sequence of rows. This is the triangular scheme for the evaluation of a point on a Bézier surface patch (cf., (Piegl 1986)). It will be noted that the surface patch may also be expressed in the form

$$P(u, v) = \left(\mathcal{L}_u\right)^n \left(\left(\mathcal{L}_v\right)^m\right)^* \mathbf{P}^t$$

Note that taking the transpose of  $\mathbf{P}$  is indicated. Such an operation must actually be carried out when encoding in PROLOG. On the other hand, in developing an imperative algorithm which facilitates  $ij$  indexing, there is no issue. The imperative encoding of the *de Casteljau* algorithm for the surface patch is straightforward. Finally, let us turn our thoughts to the construction of an appropriate wire-frame.

### 5.1. Construction of the Wire-Frame

A high-level specification of the algorithm for the construction of a wire-frame, analogous to that for a Bézier curve, is immediately given by:

$$\begin{aligned} &\text{build\_patch: } MESH \longrightarrow PATCH \longrightarrow POLYLINE^* \times POLYLINE^* \\ &\text{build\_patch}[[mesh]]\mathbf{P} \triangleq \\ &\quad \text{let } w = 0, \text{ step} = 1/mesh \text{ in} \quad \text{--- } mesh \neq 0 \\ &\quad \text{build}[[mesh, w, step]]\mathbf{P} \end{aligned}$$

Note that, for the wire-frame, the parameters  $u$  and  $v$  have been replaced by the single parameter  $w$ . The build algorithm is specified by

$$\begin{aligned} &\text{build: } MESH \times \mathbf{R} \times \mathbf{R} \longrightarrow PATCH \longrightarrow V\_POLYLINE^* \times U\_POLYLINE^* \\ &\text{build}[[mesh, w, step]]\mathbf{P} \triangleq \\ &\quad \left( \langle \text{build\_curve}[[mesh]] \left(\left(\mathcal{L}_w\right)^n\right)^* \mathbf{P} \rangle, \langle \text{build\_curve}[[mesh]] \left(\left(\mathcal{L}_w\right)^m\right)^* \mathbf{P}^t \rangle \right) \\ &\quad \wedge \text{build}[[mesh, w + step, step]]\mathbf{P} \end{aligned}$$

with the termination case given by

## GRAPHICS

$$\text{build}[\text{mesh}, 1, \text{step}]\mathbf{P} \triangleq \left( \langle \text{build\_curve}[\text{mesh}]\left((\mathcal{L}_1//)^n\right)^* \mathbf{P} \rangle, \langle \text{build\_curve}[\text{mesh}]\left((\mathcal{L}_1//)^m\right)^* \mathbf{P}^t \rangle \right)$$

where the superscripts  $n$  and  $m$  are defined by

$$n = \text{deg}_u(\mathbf{P})$$

$$m = \text{deg}_v(\mathbf{P})$$

### 5.1.1. A Refinement

In making progress along the path towards an imperative encoding, let us take an approach similar to that for the development of `build_curve` above by introducing some slices to hold the result:

$$\begin{aligned} \text{build\_patch}: \text{MESH} &\longrightarrow \text{PATCH} \\ &\longrightarrow \text{POLYLINE\_SLICES}^* \times \text{POLYLINE\_SLICES}^* \\ \text{build\_patch}[\text{mesh}]\mathbf{P} &\triangleq \\ \text{let } w = 0, \text{step} = 1/\text{mesh} &\text{ in } \quad \text{-- } \text{mesh} \neq 0 \\ \text{let } \text{vpls}[1 \dots \text{mesh} + 1], &\text{upls}[1 \dots \text{mesh} + 1] \in \text{POLYLINE\_SLICES} \text{ in} \\ \text{build}[\mathbf{P}, \text{mesh}, w, \text{step}] &(\text{vpls}[1 \dots \text{mesh} + 1], \text{upls}[1 \dots \text{mesh} + 1]) \end{aligned}$$

where the refinement of the build algorithm is specified by

$$\begin{aligned} \text{build}: \text{PATCH} \times \text{MESH} \times \mathbf{R} \times \mathbf{R} & \\ &\longrightarrow \text{POLYLINE\_SLICES}^* \times \text{POLYLINE\_SLICES}^* \\ &\longrightarrow \text{POLYLINE\_SLICES}^* \times \text{POLYLINE\_SLICES}^* \\ \text{build}[\mathbf{P}, \text{mesh}, w, \text{step}] &(\text{vpls}[i \dots \text{mesh} + 1], \text{upls}[i \dots \text{mesh} + 1]) \triangleq \\ \left( \text{vpls}[i \mapsto \text{build\_curve}[\text{mesh}] &\left((\mathcal{L}_w//)^n\right)^* \mathbf{P}], \right. \\ \left. \text{upls}[i \mapsto \text{build\_curve}[\text{mesh}] &\left((\mathcal{L}_w//)^m\right)^* \mathbf{P}^t] \right) \\ \cup \text{build}[\mathbf{P}, \text{mesh}, w + \text{step}, &\text{step}] \\ (\text{vpls}[i + 1 \dots \text{mesh} + 1], &\text{upls}[i + 1 \dots \text{mesh} + 1]) \end{aligned}$$

with the termination case given by

$$\begin{aligned} \text{build}[\mathbf{P}, \text{mesh}, w, \text{step}] &(\text{vpls}[\text{mesh} + 1], \text{upls}[\text{mesh} + 1]) \triangleq \\ \left( \text{vpls}[\text{mesh} + 1 \mapsto \text{build\_curve}[\text{mesh}] &\left((\mathcal{L}_w//)^n\right)^* \mathbf{P}], \right. \\ \left. \text{upls}[\text{mesh} + 1 \mapsto \text{build\_curve}[\text{mesh}] &\left((\mathcal{L}_w//)^m\right)^* \mathbf{P}^t] \right) \end{aligned}$$

Note that the signature of `build` is now in tail-recursive form. The next level of refinement might be developed either in the context of this recursive form or in the context of a skeletal imperative encoding of said form. Here, the latter path is chosen.

### 5.1.2. A Refinement — Skeletal Imperative Encoding

The body of `build_patch` translates directly into initialisation code and a loop, where the latter is derived directly from the build algorithm:

```

build_patch[[mesh]]P  $\triangleq$ 
  w  $\leftarrow$  0;
  step  $\leftarrow$  1/mesh;    -- mesh  $\neq$  0
  for i = [1 ... mesh + 1] do
    vpls[i]  $\leftarrow$  build_curve[[mesh]]((L_w//)^n)*P;
    upls[i]  $\leftarrow$  build_curve[[mesh]]((L_w//)^m)*P^t;
  end for
  return (vpls[1 ... mesh + 1], upls[1 ... mesh + 1])

```

Now, it should not be assumed that one must build the data structures `vpls` and `upls`! Although the sequence (and sequence slice) naturally suggests a data structure, one always has the choice in the final stages of the development to replace sequence elements with procedure calls (or, of course, inline code). On the other hand, from the point of view of specification, we have effectively defined the patch to be the cartesian product of sequences of polylines. The next obvious stage in the refinement is to consider unravelling the two invocations to build a curve: `build_curve[[mesh]]((L_w//)^n)*P` and `build_curve[[mesh]]((L_w//)^m)*P^t`.

### 5.1.3. Implementation of the Tensor

All of the diagrams presented in this Chapter were generated by graphics programs written in HyperTalk and executed in HyperCard. Said programs were directly encoded from the corresponding *Meta-IV* specifications. Indeed, I used the *Meta-IV* to develop a highly interactive prototype for computer-aided design, primarily for the purpose of research in formal specifications with respect to computer graphics. It has also served as a pedagogic tool. To conclude this section, the actual implementation of the above wire-frame algorithm in the HyperTalk programming language of HyperCard is presented. Although one is accustomed today to direct *ij* indexing in imperative languages, it is interesting to note how one used to have make do with linear arrays for all matrix work. In HyperCard, the *field* structure may be used to implement matrices.

A HyperCard field is a (possibly empty) sequence of lines. Each line is a (pos-

## GRAPHICS

sibly empty) sequence of items, each of which are separated by commas. Ignoring the commas, the field may be formally specified by

$$FIELD = LINE^*$$

$$LINE = ITEM^*$$

$$ITEM = \dots$$

where *ITEM* may be a character, a word, a phrase, a number, etc.

The tensor is to be mapped linearly onto the HyperCard field in row order giving the slice  $tensor[1 \dots m'n']$  where  $m'$  is the number of rows and  $n'$  is the number of columns. Each line of the field contains a control point. For example, a mixed quadratic-cubic patch tensor is mapped onto the right field:

<b>P</b> <sub>00</sub>	$x_{00}, y_{00}, z_{00}$
<b>P</b> <sub>10</sub>	$x_{10}, y_{10}, z_{10}$
<b>P</b> <sub>20</sub>	$x_{20}, y_{20}, z_{20}$
<b>P</b> <sub>30</sub>	$x_{30}, y_{30}, z_{30}$
<b>P</b> <sub>01</sub>	$x_{01}, y_{01}, z_{01}$
<b>P</b> <sub>11</sub>	$x_{11}, y_{11}, z_{11}$
<b>P</b> <sub>21</sub>	$x_{21}, y_{21}, z_{21}$
<b>P</b> <sub>31</sub>	$x_{31}, y_{31}, z_{31}$
<b>P</b> <sub>02</sub>	$x_{02}, y_{02}, z_{02}$
<b>P</b> <sub>12</sub>	$x_{12}, y_{12}, z_{12}$
<b>P</b> <sub>22</sub>	$x_{22}, y_{22}, z_{22}$
<b>P</b> <sub>32</sub>	$x_{32}, y_{32}, z_{32}$

The field on the left hand side is simply for (user) convenience. The injection mapping is, of course, simply a flattening  $\wedge / \mathbf{P}$ :

$$\wedge / tensor = tensor[1 \dots m'n']$$

A repartitioning of the field into  $m'$  disjoint slices, each of length  $n'$ , gives

$$\langle r_1[1 \dots n'], r_2[n' + 1 \dots 2n'], \dots, r_i[n'(i-1) + 1 \dots n'i], \dots, r_{m'}[n'(m'-1) + 1 \dots m'n'] \rangle$$

which may be succinctly written as  $\langle r_i[n'(i-1) + 1 \dots n'i] \mid i = [1 \dots m'] \rangle$ . Verification that the indexed partitions cover the field may be obtained by applying the reduction  $\cup /$ :

$$\cup / \langle r_i[n'(i-1) + 1 \dots n'i] \mid i = [1 \dots m'] \rangle = tensor[1 \dots m'n']$$

From the partitioning scheme, the  $i$ th row may be recovered:

$$sel\_row[[i]](tensor[1 \dots m'n'], m', n') \triangleq tensor[n'(i-1) + 1 \dots n'i]$$

The  $j$ th column may be recovered by selecting the  $j$ th offset element from the  $i$ th row:

$$\text{sel\_col}[[j]](\text{tensor}[1 \dots m'n'], m', n') \triangleq \left[ i \mapsto \text{tensor}[n'(i-1) + j] \mid i = [1 \dots m'] \right]$$

Finally, of course, for those who wish an  $ij$  indexing scheme to access elements of the tensor directly, one may use:

$$\text{sel}[[i, j]](\text{tensor}[1 \dots m'n'], m', n') \triangleq \text{tensor}[n'(i-1) + j]$$

Now, consider the assignment:

$$vpls[i] \leftarrow \text{build\_curve}[[mesh]] \left( (\mathcal{L}_w//)^n \right)^* \mathbf{P};$$

The expression on the right hand side may be successively refined as follows:

$$\begin{aligned} & \left( (\mathcal{L}_w//)^n \right)^* \mathbf{P} \\ & \quad \downarrow \\ & \left( (\mathcal{L}_w//)^n \right)^* \text{tensor}[1 \dots m'n'] \\ & \quad \downarrow \\ & \text{for } k = [1 \dots m'] \text{ do} \\ & \quad \quad q[k] \leftarrow (\mathcal{L}_w//)^n \text{sel\_row}[[k]](\text{tensor}[1 \dots m'n'], m', n'); \\ & \text{end for} \\ & \text{return } q[1 \dots m'] \end{aligned}$$

Finally, one then has the assignment

$$vpls[i] \leftarrow \text{build\_curve}[[mesh]] q[1 \dots m'];$$

A similar refinement may be applied to the expression on the right hand side of the assignment:

$$upls[i] \leftarrow \text{build\_curve}[[mesh]] \left( (\mathcal{L}_w//)^m \right)^* \mathbf{P}^t;$$

to give

$$\begin{aligned} & \text{for } k = [1 \dots n'] \text{ do} \\ & \quad \quad q[k] \leftarrow (\mathcal{L}_w//)^m \text{sel\_col}[[k]](\text{tensor}[1 \dots m'n'], m', n'); \\ & \text{end for} \\ & \text{return } q[1 \dots n'] \end{aligned}$$

Putting it all together gives the required imperative encoding and the result shown in the diagram at the head of this section.

# GRAPHICS

## 6. Summary

The focus in this Chapter, in which the concept of ‘discovering specifications’ has been further elaborated, was on reverse-engineering. From hard practical experience I have concluded that trying to understand and teach the ‘specifications’ of computer graphics algorithms, which are customarily given as encodings in some programming language: C, Pascal, APL, Ada, etc., is very wasteful of precious time. The reverse-engineering of such algorithms has proven to be a more fruitful activity. It is exceedingly difficult to explain in writing how one actually does reverse-engineer such algorithms. In the domain of computer graphics, visual feedback seems to be very important. After all, discovery of specifications is very much an exploratory activity.

Computer graphics and mathematics are inextricably linked. Where graphics algorithms have been derived from mathematical equations, the transformation from encoding to *Meta-IV* specification has proven to require not too much effort. This is certainly true of the specifications of the basic graphics algorithms presented in the first part of the Chapter. Similar remarks apply to the curve and surface algorithms of computer-aided design. Much of the specification that I presented is not new mathematical material. My contribution has been to set the material in a universal *Meta-IV* context. For example, the special ‘reduction’ form of the definition of a Bézier curve that I derived may equally well be recast in terms of finite difference operators. But, being in reduction form, one can immediately relate it to other reduction forms in different non-mathematical problem domains.

As *pure mathematical entities*, Bézier curves and surface patches are beautiful. It is only fitting that such beauty should be preserved (be an invariant) in the translation to the corresponding algorithmic forms. It has been demonstrated that the *de Casteljau* algorithm does just that, and that said beauty is readily revealed when appropriate algorithmic notations are employed. Additionally, it comes as a surprise that the algorithms themselves may serve as (constructive mathematical) definitions, and may be subjected to the same sort of operations as traditional forms. Finally, that the algorithms may be directly transliterated into efficient programs in languages such as PROLOG gives one hope that in the future CAD ‘programming’ will avoid distortions of the original mathematical definitions, as is currently the

case when, say, FORTRAN or C, is used.

At the time of writing, I had only used PROLOG encodings to give numerical results. HyperCard provided a more stimulating interactive graphics environment. There is still much work to be done in mapping the *Meta-IV* specifications into PROLOG encodings in an interactive graphics environment. This done, then the marriage of classical CAD and AI becomes a real possibility.

The *VDM* has been traditionally used to give the denotational semantics of (programming) languages. I have even used it to specify the static parts of interactive computer graphics interfaces: menus, windows, etc. For a long time I had believed that the very nature of (user) interaction called for a different approach to building specifications. I now know that this is not the case. There is a smooth conceptual natural transition from static structure to communication, and from communication to behaviour, and this may be adequately and faithfully expressed within the notational forms of the Irish School of the *VDM*. I now address this issue, one which impinges directly on an understanding of the user's conceptual model, the very point of departure.



# Chapter 8

## Communications & Behaviour

### 1. Introduction

How is it, that one accustomed to the notation and method of the *VDM*, should find the concepts and notation of process algebras difficult to assimilate? Why do people make remarks such as ‘the *VDM* does not have constructs for expressing concurrency’, etc., and “... some process algebras, for instance CSP, have no way of expressing functionality. Formalisms combining events and functionality are only just beginning to appear ...” (Took 1990, 70)? It seems to me that both the nature of specification and the nature of method of specification are poorly understood. Indeed, I see the problem in terms of ‘conceptual model inadequacy’. With respect to the members of those communities who work in *VDM*,  $\mathcal{Z}$ , CSP, etc., surely they see the ‘obvious’? The phenomenon just described is not unique to the world of formal specifications. Kuhn ([1962] 1970), in his work on *The Structure of Scientific Revolutions* ascribes the nature of scientific revolutions in part to just such a phenomenon and that the ‘paradigm shift’ involves retention of existing terminology and concepts together with a reconstruction of the conceptual models of the ‘revolutionaries’ engaged in change:

“Just because it did not involve the introduction of additional objects or concepts, the transition from Newtonian to Einsteinian mechanics illustrates with particular clarity the scientific revolution as a displacement of the conceptual network through which scientists view the world” (Kuhn [1962] 1970, 102).

I will demonstrate that in the case of formal specifications with particular reference to the association between structure and behaviour, such conceptual model rebuilding often involves a change in *Gestalt*, a process to which Kuhn also refers ([1962] 1970, 85), and mentioned briefly by Pólya ([1962] 1981, 2:68) in the context

## COMMUNICATIONS & BEHAVIOUR

of (mathematical) problem solving. I note that initially one's resolution to the apparent conflict between two conceptual models such as those that underlie *VDM* and *CSP* specifications is a matter of accommodation—in the following strict sense of that term. To accommodate is to make room for, to tolerate the existence of something with which one is not yet 'at home'. What is desired and to be achieved is a complete assimilation—whereby that which was once strange is inseparably part of our understanding. In solving problems, assimilation is the last phase of three identified by Pólya for efficient learning: “*an exploratory phase should precede the phase of verbalization and concept formation and, eventually, the material learned should be merged in, and contribute to, the integral mental attitude of the learner*” (Pólya [1962] 1981, 2:104).

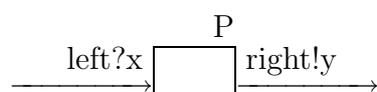
Now philosophy clearly makes distinctions between entities or objects and agents, the importance of such distinction for conceptual models in computing having been exhaustively elaborated in Chapter 6 on Requirements. For my part, there ought to be a clear conceptual progression of development from structure to behaviour. Entities ought to be identified and their structure exposed. Next, the question of whether such structure ought to be internalized within some agent needs to be examined. Agents are autonomous by nature. It is clear that they engage at times in communication with one another. There may be other forms of interaction; but for the present only the property of the ability to communicate seems worthwhile at the moment. Finally, and where necessary, does time enter the picture. Developing a specification that follows this pattern of thought is, for me, a natural method of arriving at a conceptual model of some system. In the following subsection I have tried to indicate just how such a *Gestalt* shift ought to occur from the point of view of one familiar with the *VDM* and attempting to assimilate *CSP*. The remainder of the Chapter elaborates the theory of Conceptual Models with respect to Communication and Behaviour and explains in detail the method by which one ought to proceed.

## 1.1. Communicating Sequential Processes

Curiously, Hoare addresses his seminal work on Communicating Sequential Processes (CSP) to “the programmer who aspires to greater understanding and skill in the practice of an intellectually demanding profession” (Hoare 1985, 11). Whereas there is no question of the soundness of the mathematical approach, I suspect that the ‘programmer’ to whom the material is addressed finds the material most perplexing. I suspect that the word ‘programmer’ was intended for market consumption. For suppose that the work had been addressed to the *system’s architect*, one who had the ability to specify and design, then I should imagine that the number of copies sold would have been very few indeed.

The study of CSP promises insight that “includes as special cases many of the advanced structuring ideas which have been explored in recent research into programming languages and programming methodology—the monitor, class, module, package, critical region, envelope, form, and even the humble subroutine” (Hoare 1985 11). Then why does Took make the remark quoted above? In my theory of conceptual models, the appropriate entry point into CSP begins with the material on communication.

To illustrate the soundness of CSP as a means for expressing the specification of behaviour, specifically communication and, that *indeed* functionality is implicit if not explicit, let me demonstrate that a piece of CSP may be conveniently recast in the context of structure (functionality) expressed in the VDM. To illustrate the point, I have chosen the example of the stack (Hoare 1985, 138):



the specification of which is given by

$$STACK = P_{\langle \rangle}$$

$$\begin{array}{l} \text{where } P_{\langle \rangle} = \left( \text{empty} \rightarrow P_{\langle \rangle} \mid \text{left?x} \rightarrow P_{\langle x \rangle} \right) \\ \text{and } P_{\langle x \rangle \wedge s} = \left( \text{right!x} \rightarrow P_s \mid \text{left?y} \rightarrow P_{\langle y \rangle \wedge \langle x \rangle \wedge s} \right) \end{array}$$

Now it is clear from this specification that stack operations such as *push* and *pop* are involved. That it should be clear is due in particular to the happy use by Hoare of a notation that is exactly that of the VDM *Meta-IV*, to wit,  $\langle x \rangle \wedge s$ . It was

## COMMUNICATIONS & BEHAVIOUR

just this very notation that historically supplied one of the triggers for the shift in *Gestalt*. A tentative VDM specification written in a style to fit the CSP might look like the following. First the stack is modelled by:

$$STACK = X^*$$

Then, the appropriate operations are

$$Push: X \longrightarrow STACK \longrightarrow STACK$$

$$Push[x]\Lambda \triangleq \langle x \rangle$$

$$Push[x']\langle x \rangle \wedge s \triangleq \langle x' \rangle \wedge \langle x \rangle \wedge s$$

and

$$Pop: STACK \longrightarrow STACK \times X$$

$$Pop(\langle x \rangle \wedge s) \triangleq (s, x)$$

Recalling earlier remarks on the linear form of VDM specification, I wish to note once more that a 2-dimensional form is extremely suggestive of labelled transition systems. Thus, the form  $Push[x']\langle x \rangle \wedge s \triangleq \langle x' \rangle \wedge \langle x \rangle \wedge s$  may just as readily be presented as

$$\langle x \rangle \wedge s \xrightarrow{Push[x']} \langle x' \rangle \wedge \langle x \rangle \wedge s$$

providing yet a second trigger for the ‘Gestalt’ shift. Before demonstrating, how the CSP and VDM fit comfortably together, I introduce explicitly the notion of state as distinct from the notion of structure:

$$\Sigma :: STACK$$

in order to reify my understanding that the process managing the stack has memory which ‘contains’ the stack as distinct from the stack itself. Although I am generally in favour of subscripting where possible, in general complex states prohibit such an approach. Thus, an expression such as  $P_{\langle x \rangle \wedge s}$ , will usually be written in the form  $P[mk-\Sigma(\langle x \rangle \wedge s)]$ . Finally, I just need to replace the quaint homely channel names ‘left’ and ‘right’ with their operational equivalents and rearrange the positioning of the channel arrows to give the tentative partial result:

$$\begin{array}{c} \text{P} \\ // \xrightarrow{Push?y} \boxed{mk-\Sigma(\langle x \rangle \wedge s)} \\ // \xleftarrow{Pop!x} \end{array}$$

with specification

$$\begin{aligned}
 & \text{STACK} = P[\Lambda] \\
 \text{where } & P[mk-\Sigma(\Lambda)] = \left( \text{empty} \rightarrow P[mk-\Sigma(\Lambda)] \right. \\
 & \quad \left. | \text{Push?}x \rightarrow P[mk-\Sigma(\langle x \rangle)] \right) \\
 \text{and } & P[mk-\Sigma(\langle x \rangle \wedge s)] = \left( \text{Pop!}x \rightarrow P[mk-\Sigma(s)] \right. \\
 & \quad \left. | \text{Push?}y \rightarrow P[mk-\Sigma(\langle y \rangle \wedge \langle x \rangle \wedge s)] \right)
 \end{aligned}$$

Note the introduction of the state  $mk-\Sigma(\langle x \rangle \wedge s)$  into the ‘process’ box labelled  $P$ . The relationship between the diagrammatic form and the formal expression is further strengthened by the adoption of the notation  $P[mk-\Sigma(\langle x \rangle \wedge s)]$  where the square brackets may be deemed to stand for the box in the diagram. The occurrence of the special channel arrow delimiter  $//$  indicates that there must be something out there to which the channels are ultimately connected, be that another process, or just the environment.

Now I have not obtained exactly the fit that I am seeking between the *VDM* and the *CSP*. I have not tried to resolve the *empty* event with respect to the *VDM* model, which event I still consider to be counter-intuitive from a conceptual model point of view. Moreover, the event name *Pop!x* seems to me to be very strange. To achieve complete assimilation I turn to the issue of some possible interpretations—implementations, if you will—of my specification. Firstly, I ‘see’ the possibility that the events *Push?y* and *Pop!x*, which ought to correspond directly with the stack operations in an obvious way, might be implemented as selections from a display menu on a graphical user interface. Then, conceptually, a *push* operation must have an argument  $x$ , the element to be pushed on to the stack. This is in exact agreement with the model of *push* as a piece of abstract syntax. Given the syntactic domain:

$$\text{Push} :: X$$

Then  $mk\text{-}Push(x)$  denotes a typical *push* command. The important point to note is that the menu selection would not of itself normally supply the argument; thus selecting *push* is conceptually a two phase operation. The argument must either already be selected before *push* is invoked or must be chosen after the invocation. A second implementation scenario which confirms this user conceptual model viewpoint is the more obvious interpretation of *push* as a command language operation as strongly suggested by the traditional connotations of abstract syntax. Finally, a

## COMMUNICATIONS & BEHAVIOUR

third scenario is worth mentioning—the event driven interface of HyperCard, particularly in view of some of the real potential human factors inadequacies of the menu implementation concept. If *push* is the menu selection, then how is one to obtain the argument? A conventional solution is to prompt for that argument, an implementation that exhibits a certain discontinuity in mode of operation. The command language implementation does not have such discontinuity. Now in a HyperCard context, one might very well use a ‘button’ labelled ‘push’ for the *push* operation, which when entered by the mouse changes form to a cross, say, indicating to the user that the argument is to be selected. Upon placing the cross over the argument and clicking, the desired effect is achieved. All of the above scenarios are just some of the possible reifications of the abstract specification given.

In the case of *pop*, it is clear that the operation is selected without arguments and then the result is obtained. Formally, the *VDM* specification ought to have been

$$Pop \llbracket \rrbracket (\langle x \rangle \wedge s) \triangleq \dots$$

an expression that seems to be slightly pedantic. To reinforce this point a corresponding syntactic domain might be

$$Pop :: \{nil\}$$

giving the command *mk-Pop(nil)*. The *Pop* operation involves both state change and the return of a result. The usual abstract data type approach of using *pop* to change state and *top* to return the result is conceptually more elegant and may be transformed exactly into appropriate CSP form. Again, with reference to *Pop*, there is no straightforward labelled transition form of the specification. On the contrary, splitting it into two conceptually distinct operations:

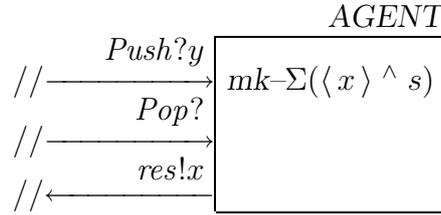
$$\begin{aligned} Pop \llbracket \rrbracket (\langle x \rangle \wedge s) &\triangleq s \\ Top \llbracket \rrbracket (\langle x \rangle \wedge s) &\triangleq x \end{aligned}$$

leads directly to the dual

$$\begin{array}{ccc} \langle x \rangle \wedge s & \xrightarrow{Pop \llbracket \rrbracket} & s \\ \langle x \rangle \wedge s & \xrightarrow{Top \llbracket \rrbracket} & x \end{array}$$

Returning then to the transformation of the CSP, I note that I need two channels for the *pop* operation—one for the command itself and a result channel. This seems

to agree with the convention of Hoare himself that “channels are used for communication in only one direction ...” (1985, 134). Consequently I construct:



with specification

$$\begin{aligned}
 \text{AGENT}[\text{mk-}\Sigma(\Lambda)] &= \text{Push?}x \rightarrow \text{AGENT}[\text{mk-}\Sigma(\langle x \rangle)] \\
 \text{AGENT}[\text{mk-}\Sigma(\langle x \rangle \wedge s)] &= \left( \text{Pop?} \rightarrow \text{res!}x \rightarrow \text{AGENT}[\text{mk-}\Sigma(s)] \right. \\
 &\quad \left. | \text{Push?}y \rightarrow \text{AGENT}[\text{mk-}\Sigma(\langle y \rangle \wedge \langle x \rangle \wedge s)] \right)
 \end{aligned}$$

where the meaningful concept of *AGENT* has been introduced in place of the anonymous *P*. Now I am satisfied that I have made the transition, noting that I have dropped the anomalous occurrence of the *empty* event altogether.

## 2. Communicating Agents

Having achieved some degree of correspondence between the notations of the *VDM Meta-IV* and Hoare’s *CSP* with respect to a ‘toy’ data structure such as *STACK*, I turned my attention to the sort of activities that normal human beings might engage in and considered a full-scale *Gedanken* experiment based on the concept of looking up a dictionary. Ever at the back of my mind was the thought that entities in the world have both form and content. Form, I interpreted as being synonymous with structure and structure could be handled easily with *Meta-IV* domain models. Operations on structure, which in a programming language such as Ada might be interpreted as the procedures/functions that operate on some encapsulated data type within a package, were clearly amenable to treatment within the denotational semantics framework of the *VDM*. That the *CSP* channel names could be associated with the names of operations effectively permitted the *Gestalt* shift between two distinct conceptual views of the same thing. I now had a connection between operations and communication. It was a simple matter to draw the

## COMMUNICATIONS & BEHAVIOUR

conclusion that structure ought to be embedded within something, which I called an agent, that permitted me to distinguish between the structure within and the entity that guarded or controlled access to the structure. Realistically such agents must be permitted to be autonomous and that, in addition to functionality offered by an agent and the possibility of communication with other agents, their behaviour must also be described. From the perspective of formal specification, just as structure was independent of communication, so the specification of behaviour ought to be independent of both. I now present the original *Gedanken* experiment—looking up a dictionary.

### 2.1. Entities and Structure

There are occasions when I need to look up the definition of a word. For this purpose, I usually consult a dictionary, such as the *Oxford English Dictionary*. I might also use it to check the spelling of a word; but my primary focus at present is in the dictionary as an entity that associates definitions with words. Formally, I model this as

$$DICT_0 = WORD \xrightarrow{m} \mathcal{PDEF}$$

As a user of the dictionary, *the most important operation from my present point of view* is that which enables me to look up the definition of a word  $w$  in a dictionary  $\delta_0$ . Formally, this is specified by the function:

$$\begin{aligned} Lkp_0: WORD &\longrightarrow DICT_0 \longrightarrow \mathcal{PDEF} \\ Lkp_0[[w]]\delta_0 &\triangleq \\ \chi[[w]]\delta_0 &\rightarrow \delta_0(w) \\ &\rightarrow \perp \end{aligned}$$

The emphasis on lookup as the most important operation that I want to perform on a dictionary sharply distinguishes between the concept of semantics of an operation and the user's psychological need or viewpoint. Indeed, it is this latter need that characterises 'end-user requirements'.

Note the significance of the use of the characteristic function  $\chi[[w]]\delta_0$ . It reads 'if the word  $w$  is in the the dictionary  $\delta_0$  then ... else ...'. There is no question of doing a *search* for the word; the word is either there or it is not. Secondly, I do not care whether the intended word is spelled correctly or not—that is a separate issue to be addressed later. Now I would like to make a few remarks on the intended

semantics of the symbol  $\perp$ . At present I take it to mean ‘undefined’, or better still, ‘do not care at this stage’. As we shall see, it will be given a very precise fixed meaning when we come to consider the dictionary as an entity or object in the world of communicating agents.

## 2.2. Agents and Entities

Now I have given the impression that I was thinking of a dictionary as a book. But this is not really the case. I might just as easily assume that there is a helpful person present whom I deem to be expert enough to supply me with the definitions of words that I need. I understand that this person has a dictionary in his head, in his memory if you will, and that he is prepared to respond to my requests to look up words. Such a person I consider to be an agent, in the sense that the person is totally independent of me and is prepared to do other things quite apart from fielding my questions. Perhaps, I am also considering the possibility of using a computer to help me in my work and whilst engaged in the task of, say, writing this thesis, I find the need to look up the definition of a word. To my pleasant surprise, I find that there is indeed a dictionary ‘in the machine’ that meets my needs; but access to this dictionary is managed or governed by a running program. I see no reason why this program should not be called an agent. Although it is limited in the sorts of actions that it is prepared to perform, it seems just as helpful as the person of whom I spoke earlier. These considerations lead me, therefore, to the following formal concepts.

Let  $AGENT_0[\sigma_0]$  denote the agent in state  $\sigma_0 = mk-\Sigma_0(\delta_0)$  that manages the dictionary  $\delta_0$  where  $\Sigma_0$  is defined by

$$\Sigma_0 :: DICT_0$$

I have wrapped up the dictionary between ‘covers’ which I call state. This word is suitably general to capture the concepts of ‘book’—in the case of the *Oxford English Dictionary*—and ‘memory’—in the case of the person and the computer. I must now revisit the specification for the *lookup* function. Only minor, almost cosmetic, changes are necessary:

$$\begin{aligned} Lkp_0: WORD &\longrightarrow \Sigma_0 \longrightarrow \mathcal{PDEF} \\ Lkp_0[[w]](mk-\Sigma_0(\delta_0)) &\triangleq \\ \chi[[w]]\delta_0 &\rightarrow \delta_0(w) \\ &\rightarrow \perp \end{aligned}$$



since my understanding of the relationship between structure and behaviour as developed in this thesis, agrees with the observations of others. I will not go so far as to accept the point of ‘theoretical equivalence’. My concept of ‘functionality’ is limited to the context of ‘function over structure’—which has very precise mathematical meaning for me. It is to be further noted that the conceptual model that I am developing here is very similar to the approach taken by Sufrin himself in the context of  $\mathcal{Z}$  (Sufrin and He 1990).

Given an entity wrapped up in some state, then the general rule for a function  $F$  with parameter  $x$  which has the form

$$F: X \longrightarrow \Sigma \longrightarrow VAL$$

$$F[[x]]\sigma \triangleq \dots$$

is that the indexed name  $F[[x]]$  becomes the input event  $F?x$ , as seen by the agent, where the name  $F$  becomes the channel name and the argument  $x$  becomes the message; the successful result of the computation  $v$  is transmitted as the output event  $Res!v$  where  $Res$  is a result channel associated with  $F$ ; in the event that no value can be computed, i.e., there is the possibility of  $\perp$ , then a message is transmitted on an exception channel:  $Xcptn!msg$ , associated with  $F$ . In conclusion every function on a structure, from the point of view of the agent managing it, or responsible for it, gives rise to three channels—the request channel denoted by the function itself, the result channel, and the exception channel.

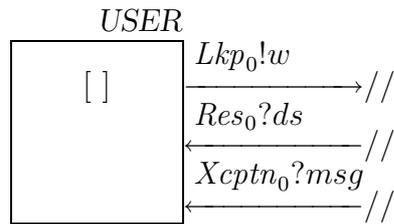
### 2.3. The User as Agent

Thus far, in developing the conceptual model of communicating agents, I have supposed myself to be that user who requests the definitions of words. Moreover, the dictionary entity has been regarded as a real object which has existence independent of myself. Now it is clear that there is no reason why I should not schizophrenically regard myself as both user and the agent who manages the dictionary, thus, modelling if you will, my own ability to recall the definitions of words. Rather than explore this avenue here, let me take another point of view—that of observer—watching the behaviour of some other user making requests for the definitions of words. Such a user may of course be a piece of software as much as a person. Now the first issue to be addressed, is whether or not to posit the existence of some further entity that said user might manage—say his own internal memory. In keeping

## COMMUNICATIONS & BEHAVIOUR

with the method delineated thus far, it might seem appropriate to do so and then to fill in the behaviour.

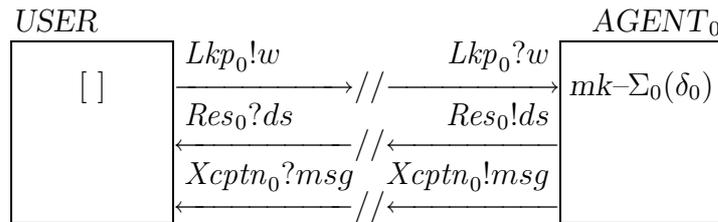
Instead, let us consider the application of Ockham's Razor and examine how far one can get by considering behaviour alone. Such a user is an agent without state:  $USER[ ]$ . I observe that the user is prepared to ask for the definition of a word:  $Lkp_0!w$ ; and then either expects to get the corresponding definitions or some sort of message to the effect that such definitions do not exist. Formally, this behaviour may be specified by:



where

$$\begin{aligned}
 \text{USER}[ ] &\triangleq \\
 &Lkp_0!w \rightarrow \left( Res_0?ds \rightarrow \text{USER}[ ] \right. \\
 &\quad \left. | Xcptn_0?msg \rightarrow \text{USER}[ ] \right)
 \end{aligned}$$

This description of the user's behaviour is limited to his interest in looking up the definition of words in a dictionary. If definitions result from such a request or if an exception message is received then the user will issue another *lookup* command. There is no indication that the user 'does' anything with said definitions or that receipt of the exception message causes him to behave differently. This is not a very enlightening model of the behaviour of a real user. But my attention is not at all focused on 'real' user behaviour. The interesting point to note is that the user is at the other end of the communication channels to the agent. Putting both halves of the specification together gives us the 'complete' picture:



The rôle of the channel markers // may now be explained fully. It may be the case that there is a 'direct line' of communication between user and agent, say by

telephone for example. On the other hand I do not wish to rule out the possibility that something intervenes between the two. For instance, it is reasonable to imagine the situation that the user issues *lookup* commands faster than the agent can handle them and that some sort of buffering mechanism needs to be inserted. Of course, once we consider the possibility of many users all trying to access the same dictionary—a shared resource—then we can let our imaginations run riot on concepts of multiplexing, packet switching, etc. It is all potentially present in the specification.

One is tempted to ask if users can enter words and their definitions into the dictionary, whether the dictionary is local or distributed, each such question leading into other evolutionary specification paths. I would like to stay with the simple notion of *lookup* for the present; for much may yet be learned.

#### 2.4. Context Sensitive Communication

Let us now suppose that in looking up the definition of a word  $w$ , the user prefers to see the definitions one at a time instead of the entire set of definitions, i.e., upon issuing the *lookup* command, gets a single definition and then has the option of issuing a *next* command to obtain the next definition, if there is one. Clearly, the agent managing the dictionary must be able to remember which definition the user has already seen and which ones may yet be requested by the *next* command. In other words, the semantics of *next* will depend on context, one which determines what has already been seen and what is yet to be seen. I might introduce immediately an abstract model for such context. Instead I have chosen here to extend the state in an appropriate manner:

$$\Sigma_1 :: DICT_0 \times PDEF$$

There are essentially two kinds of state:  $mk-\Sigma_1(\delta_0, \{d_1, d_2, \dots, d_n\})$ , in which the agent can respond directly to an invocation of the *next* command, and  $mk-\Sigma_1(\delta_0, \emptyset)$  in which the context is empty, denoted by  $\emptyset$ , and in which the agent would raise an exception in the case of an invocation of the *next* command.

Let me hasten to remark that I do **not** consider this new model of state to be a reification in the standard sense of that term used among members of the VDM community. It is in reality an elaboration or refinement of the abstract model (at the same abstract level as the preceding model). Such elaboration I term *evolution*, since

## COMMUNICATIONS & BEHAVIOUR

both the terms elaboration and refinement have connotations that I wish expressly to avoid. There is no question of the new model being more concrete than the previous one, or of moving in the direction of an implementation, i.e., being reified. I am merely taking account of those matters that arise as I consider the appropriateness of the original model to capture all that needs to be captured in my evolving conceptual model.

The new abstract model calls for a change in the specification of the *lookup* command which will both modify the state and return a single definition. Technically, in the classical use of the *VDM* for the specification of denotational semantics, *lookup* is strictly a command and no longer a function—it affects the state:

$$\begin{aligned}
 Lkp_1: WORD &\longrightarrow \Sigma_1 \longrightarrow \Sigma_1 \times DEF \\
 Lkp_1[w](mk\text{-}\Sigma_1(\delta_0, ds)) &\triangleq \\
 \chi[w]\delta_0 & \\
 \rightarrow \text{let } ds' = \delta_0(w) \text{ in} & \\
 ds' = \emptyset & \\
 \rightarrow \perp & \\
 \rightarrow \text{let } d \in ds' \text{ in} & \\
 (mk\text{-}\Sigma_1(\delta_0, ds' \setminus \{d\}), d) & \\
 \rightarrow \perp &
 \end{aligned}$$

Note that I have changed the subscript of the *lookup* operation from 0 to 1 to indicate the fact that it is associated with the state:  $\Sigma_1$ , and not with the dictionary:  $DICT_0$ .

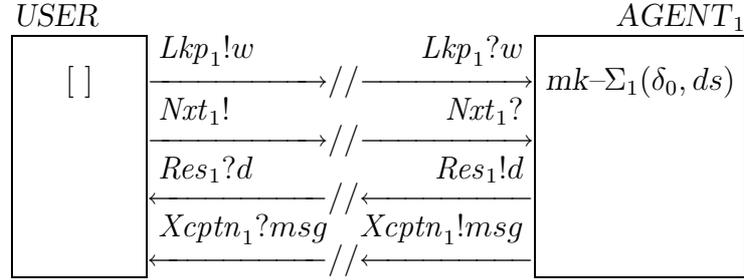
The semantics of the *next* command may be given:

$$\begin{aligned}
 Nxt_1: \Sigma_1 &\longrightarrow \Sigma_1 \times DEF \\
 Nxt_1(mk\text{-}\Sigma_1(\delta_0, ds)) &\triangleq \\
 ds = \emptyset & \\
 \rightarrow \perp & \\
 \rightarrow \text{let } d \in ds \text{ in} & \\
 (mk\text{-}\Sigma_1(\delta_0, ds \setminus \{d\}), d) &
 \end{aligned}$$

Comparing the two specifications, one is immediately struck by the fact that the *next* command semantics forms a subset of those of the *lookup* command, an observation that leads readily to the following form for the *lookup* command:

$$\begin{aligned}
 Lkp_1: WORD &\longrightarrow \Sigma_1 \longrightarrow \Sigma_1 \times DEF \\
 Lkp_1[w] &(mk-\Sigma_1(\delta_0, ds)) \triangleq \\
 &\chi[w]\delta_0 \\
 &\rightarrow \text{let } ds' = \delta_0(w) \text{ in} \\
 &\quad Nxt_1(mk-\Sigma_1(\delta_0, ds')) \\
 &\rightarrow \perp
 \end{aligned}$$

This is a very satisfying result, one which highlights the intuitive feeling that there is not really all that much distinction between *lookup* and *next*. With these specifications one immediately obtains:



where

$$\begin{aligned}
 AGENT_1[\sigma_1] &\triangleq \text{let } mk-\Sigma_1(\delta_0, ds) = \sigma_1 \text{ in} \\
 &\left( Lkp_1?w \rightarrow \chi[w]\delta_0 \right. \\
 &\quad \rightarrow \text{let } ds' = \delta_0(w) \text{ in} \\
 &\quad \quad \text{let } \sigma'_1 = mk-\Sigma_1(\delta_0, ds') \text{ in} \\
 &\quad \quad \quad AGENT_1[Nxt_1(\sigma'_1)] \\
 &\quad \rightarrow Xcptn_1!msg \rightarrow AGENT_1[\sigma_1] \\
 &\quad | Nxt_1? \rightarrow ds \neq \emptyset \\
 &\quad \rightarrow \text{let } \in ds \text{ in} \\
 &\quad \quad \text{let } \sigma'_1 = mk-\Sigma_1(\delta_0, ds \setminus \{d\}) \text{ in} \\
 &\quad \quad \quad Res_1!d \rightarrow AGENT_1[\sigma'_1] \\
 &\quad \left. \rightarrow Xcptn_1!msg \rightarrow AGENT_1[\sigma_1] \right)
 \end{aligned}$$

Note the occurrence of the expression  $AGENT_1[Nxt_1(\sigma'_1)]$  above in response to the event  $Lkp_1?w$ . This is interpreted to mean that  $AGENT_1$  behaves as if the event had been  $Nxt_1?$  but while in state  $\sigma'_1$ . By this means, the description of the behaviour of  $AGENT_1$  mirrors exactly the VDM specification of the operations  $Lkp_1[w]$  and  $Nxt_1[ ]$  over  $\Sigma_1$ . We also need to consider the description of the behaviour of the user. Since the model of the user agent does not yet have any state or memory, then I would expect to find something like

## COMMUNICATIONS & BEHAVIOUR

$$USER[ ] \triangleq \dots \\ \left( \begin{array}{l} Lkp_1!w \rightarrow \dots \\ | \text{Next}_1! \rightarrow \dots \end{array} \right)$$

In other words the user is prepared to issue *lookup* or *next* commands. But what, one might ask, is the basis for the user's choice? Surely the issuing of a *next* command depends on a user having issued a *lookup* command? Does not this imply that the user must remember what state he is in? We must inevitably conclude that indeed it must be so! It is now appropriate to introduce more structure into the user's conceptual model.

### 2.5. The User's Context

At the very least the user agent ought to be as well equipped as the agent managing the dictionary. The user will know about words and definitions; for otherwise how can said user interact meaningfully with the agent? These thoughts lead me to give the user a structure identical to that of the agent:

$$\Sigma_1 :: DICT_0 \times \mathcal{P}DEF$$

There is no danger here of getting confused as to which of the two agents,  $USER_1$  or  $AGENT_1$ , a particular state  $\sigma_1$  refers, as we will see shortly. I will not worry about the sort of (internal) operations that a user might perform. Such operations will naturally be forced by considering the user's behaviour. First, let us explore what is needed for a user to ask the question  $Lkp_1!w$ . Obviously, he must know of the existence of the commands *lookup* and *next* and their intended meaning and this knowledge ought to be incorporated into the user's model. But let us not focus on this at the moment. The user must also know a word  $w$  in order to be able to issue  $Lkp_1!w$ . Such a word must be found in his own internal dictionary, in his memory. Therefore, more of the behaviour may be filled in:

$$USER_1[\sigma_1] \triangleq \text{let } mk-\Sigma_1(\delta_0, ds) = \sigma_1 \text{ in} \\ \text{let } w \in \text{dom } \delta_0 \text{ in} \\ \left( \begin{array}{l} Lkp_1!w \rightarrow (Res_1?d \rightarrow \dots \\ \quad | Xcptn_1?msg \rightarrow \dots) \\ | \text{Next}_1! \rightarrow (Res_1?d \rightarrow \dots) \\ \quad | Xcptn_1?msg \rightarrow \dots \end{array} \right)$$

A remark on the expression  $w \in \text{dom} \delta_0$  is indicated: briefly it asserts the fact that the user knows the word  $w$ . Let us now consider the case where the user receives a response:  $\text{Res}_1?d$ , to the *lookup* request:  $\text{Lkp}_1!w$ . What would the user be expected to do with such a definition  $d$ . Naturally, the first thing is to record it in ‘short term memory’ giving the new state  $\sigma'_1 = \text{mk-}\Sigma_1(\delta_0, \{d\})$ . Note that I am ignoring such issues as Miller’s often cited  $7 \pm 2$  factor (see (Shneiderman 1980, 46) for a reference). To proceed further we must really know the user’s motivation for asking for the definition of a word in the first place. It seems reasonable to suppose that the user just wanted to compare the dictionary definitions with his own internal definitions. To specify this requires an internal *lookup* command, a dictionary operation with which we are already familiar. If indeed he already knows the definition then he will probably ask for another—via the *next* command. On the other hand, if the definition is new then he will want to update his own internal dictionary and then ask for a new definition. Thus we are forced to consider a new operation on a dictionary, that of *update*:

$$\begin{aligned} \text{Upd}_0: \text{WORD} \times \text{DEF} &\longrightarrow \text{DICT}_0 \longrightarrow \text{DICT}_0 \\ \text{Upd}_0[[w, d]]\delta_0 &\triangleq \delta_0 + [w \mapsto \delta_0(w) \cup \{d\}] \end{aligned}$$

But having accounted for the response  $d$ , it is just as likely that the user might decide to issue another *lookup* command instead of the *next* command. Therefore, the evolved behaviour (for the response to *lookup*) may be described by

$$\begin{aligned} \text{Res}_1?d \rightarrow & \\ & \text{let } ds' = \delta_0(w) \text{ in} \\ & \text{let } ds'' = \{d\} \text{ in} \\ & \chi[[w]]ds' \\ & \rightarrow \text{let } \sigma'_1 = \text{mk-}\Sigma_1(\delta_0, ds'') \text{ in} \\ & \quad \text{USER}_1[\sigma'_1] \\ & \rightarrow \text{let } \sigma'_1 = \text{mk-}\Sigma_1(\delta_0 + [w \mapsto ds' \cup \{d\}], ds'') \text{ in} \\ & \quad \text{USER}_1[\sigma'_1] \end{aligned}$$

Here I have given the specification in ‘expanded’ form. It is a customary feature of my method to do thus and, subsequently, to condense it. Such a condensed form facilitates my ability ‘to see at a glance’ the transformations involved. An equivalent condensed form is

## COMMUNICATIONS & BEHAVIOUR

$$\begin{aligned}
 Res_1?d &\rightarrow \\
 \chi[w](Lkp_0[w]\delta_0) & \\
 &\rightarrow USER_1[mk-\Sigma_1(\delta_0, \{d\})] \\
 &\rightarrow USER_1[mk-\Sigma_1(Upd_0[w, d]\delta_0, \{d\})]
 \end{aligned}$$

Note the occurrence of the original *lookup* function:  $Lkp_0[w]$ . Consider what the user's reaction might be to the receipt of an exception message instead of a definition. There are two possible cases to consider:

1. The user has requested the definition of a valid word, i.e., one which exists and is correctly spelt, and the external dictionary does not contain any definitions for such a word;
2. The user has misspelt the word.

In the first case, the external dictionary is of no use to the user who must now have recourse to some other external dictionary. In such a situation, as far as the communication between the two agents  $USER_1$  and  $AGENT_1$  is concerned, the only possible behaviour of the user is simply to carry on:

$$Xcptn_1?msg \rightarrow USER_1[\sigma_1]$$

In the second case, the user has a real problem to solve. Either we now enrich the functionality of the external dictionary to provide some sort of browsing mechanism whereby the user can 'look for' the word in question or again the user must have recourse to some other mechanism to check the spelling.

But what can we say about the overall problem of the exception message, particularly as far as principles of system specification are concerned? Most assuredly it is clear that, according as our formal specification evolves (both structurally and behaviourally), important issues are raised which must be decided one way or another. From the above scenario, we are faced with the decision whether or not to enrich the functionality of the external dictionary to permit the 'browsing' feature, and consequently, whether such 'browsing' be automatically invoked by the agent:  $AGENT_1$ , or there is a visible command which the user may invoke. Having raised the issues, we will be content with recording them and describe the behaviour in both cases simply by

$$Xcptn_1?msg \rightarrow USER_1[\sigma_1]$$

Therefore, the entire behaviour in response to the user's request to look up the definition of word will be specified by:

$$\begin{aligned}
 USER_1[\sigma_1] &\triangleq \text{let } mk\text{-}\Sigma_1(\delta_0, ds) = \sigma_1 \text{ in} \\
 &\text{let } w \in \text{dom } \delta_0 \text{ in} \\
 &\left( Lkp_1!w \rightarrow \right. \\
 &\quad (Res_1?d \rightarrow \\
 &\quad\quad \chi[[w]](Lkp_0[[w]]\delta_0) \\
 &\quad\quad \rightarrow USER_1[mk\text{-}\Sigma_1(\delta_0, \{d\})] \\
 &\quad\quad \rightarrow USER_1[mk\text{-}\Sigma_1(Upd_0[[w, d]]\delta_0, \{d\})] \\
 &\quad | Xcptn_1?msg \rightarrow USER_1[\sigma_1]) \\
 &| Nxt_1! \rightarrow \\
 &\quad (Res_1?d \rightarrow \dots \\
 &\quad | Xcptn_1?msg \rightarrow \dots) \left. \right)
 \end{aligned}$$

With respect to the description of the user's behaviour upon issuing a *next* command, it ought to be clear that the only significant difference between such behaviour and that in response to the *lookup* command is that the 'short term memory' is updated to  $ds' = ds \cup \{d\}$ . In the event of an exception message being received, there is in addition to the previous cases, also the further possibility that there are 'no more' definitions for a given word, in which case the behaviour is as before. Finally, to conclude this subsection, I suppose that I ought to mention that in an implementation *msg* will be specific. However, this is a refinement issue and will not be considered further here.

## 2.6. History

Real users have a tendency to forget. Quite apart from wishing to see definitions, one at a time, they may forget what they have already seen, and wish to return to a previous definition. We require a further evolution of the state of the agent managing the dictionary. Clearly the state is growing in complexity. Such complexity is kept under control by the introduction of ancilliary structures. It is now time to revisit our previous conceptual model of state and introduce formally the concept of context denoted  $\Xi_0$ :

$$\Sigma_1 :: DICT_0 \times \Xi_0$$

where, for now, the context  $\xi_0 \in \Xi_0$  is merely a mechanism for wrapping up the set

## COMMUNICATIONS & BEHAVIOUR

of definitions retrieved under the *lookup* command:

$$\Xi_0 :: \mathcal{PDEF}$$

I suppose that I had better remark at this point that what I am referring to as context might equally be called *short term memory*, a term that is deliberately suggestive, especially where we might wish to give the user agent some state or memory, as was the case for  $USER_1$  above. That is to say, one ought not get too carried away with the idea that the name is inherently significant in itself. In a formal specification, the name is used to bridge the mathematical structure with the ‘real’ world. Over-reliance on the significance of names will lead to severe inflexibility and prohibit us from exploiting the potential for reuse of specifications. For that purpose, I have deliberately chosen the Greek symbol  $\Xi_0$  to denote context rather than something mundane and misleading such as  $CONTEXT_0$ . Even in the choice of Greek symbol, at least in its capitalised form, I do try to suggest the concept of context—the bit in the middle is surrounded by the bit above and the bit below. Doubtless one then wonders why  $DICT_0$  has been retained in its present symbolic form. True, what is here called a dictionary is only a model of same and is, strictly speaking, no more a dictionary than  $\Xi_0$  is a context. But were the name to be replaced with something more symbolic at this stage, then there would be a danger of losing all touch with the reality of the intended application of the conceptual model.

Again, lest I forget to mention it later, the introduction of the ancillary structure  $\Xi_0$  brings before me an image of state that suggests most strongly an implementation of a computer based dictionary, to wit, the dictionary will probably reside on some mass storage medium—‘long term memory’ and there will be a local context where the action of *lookup* will take place. More, I can see that this context might be re-interpreted as a *cache*. These remarks triggered the actual specification of dictionary as cache/mass storage which I presented in Chapter 6 and which ultimately provided a correlation to Johnson’s scheduling algorithm.

Returning from the aside, such a change in the structure of the model requires corresponding modifications to both the *lookup* and *next* commands. But these changes are really only of a cosmetic nature and at any rate, we are going to modify the context to incorporate a mechanism to remember previously seen definitions:

$$\Sigma_2 :: DICT_0 \times \Xi_1$$

Obviously, the new context will conserve what was in the old  $\mathcal{P}DEF$ . To remember what has already been seen, seems to me to require the notion of order. Thus, it appears appropriate that I introduce some concept such as that of a stack:

$$\Xi_1 :: \mathcal{P}DEF \times STACK_0$$

the intention being that each definition transmitted is also pushed onto the stack; previous definitions can then be recalled in order by reading the top of the stack and then popping the stack. Again a note on nomenclature is needed. I have deliberately decided that ‘stack’ is the name I wish to use, even though the name of this subsection is ‘history’ for that is indeed what is in question. The model of the stack is simply:

$$STACK_0 = DEF^*$$

Now I am in a position to modify the already existing commands and introduce the new command. Again with a view to keeping the complexity of the specification under control, I give the semantics of *lookup* in terms of that of *next*:

$$\begin{aligned} Lkp_2: WORD &\longrightarrow \Sigma_2 \longrightarrow \Sigma_2 \times DEF \\ Lkp_2[[w]](mk-\Sigma_2(\delta_0, mk-\Xi_1(ds, \sigma_0))) &\triangleq \\ \chi[[w]]\delta_0 & \\ \rightarrow \text{let } ds' = \delta_0(w) \text{ in} & \\ \quad Nxt_2(mk-\Sigma_2(\delta_0, mk-\Xi_1(ds', \Lambda))) & \\ \rightarrow \perp & \end{aligned}$$

The only real new piece is the initialization of the history stack to the empty stack  $\Lambda$ . For the *next* semantics, the important modification is pushing the transmitted definition  $d$  onto the stack:

$$\begin{aligned} Nxt_2: \Sigma_2 &\longrightarrow \Sigma_2 \times DEF \\ Nxt_2(mk-\Sigma_2(\delta_0, mk-\Xi_1(ds, \sigma_0))) &\triangleq \\ ds = \emptyset & \\ \rightarrow \perp & \\ \rightarrow \text{let } d \in ds \text{ in} & \\ \quad (mk-\Sigma_2(\delta_0, mk-\Xi_1(ds \setminus \{d\}, \langle d \rangle \wedge \sigma_0)), d) & \end{aligned}$$

Finally, I give the semantics of the *previous* command.

## COMMUNICATIONS & BEHAVIOUR

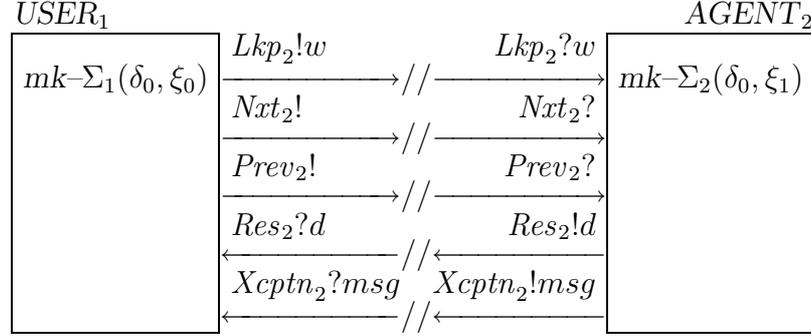
$$\begin{aligned}
 &Prev_2: \Sigma_2 \longrightarrow \Sigma_2 \times DEF \\
 &Prev_2(mk\text{-}\Sigma_2(\delta_0, mk\text{-}\Xi_1(ds, \sigma_0))) \triangleq \\
 &\quad \sigma_0 = \Lambda \\
 &\quad \rightarrow \perp \\
 &\quad \rightarrow \text{let } \sigma_0 = \langle s \rangle \wedge \tau \text{ in} \\
 &\quad \quad (mk\text{-}\Sigma_2(\delta_0, mk\text{-}\Xi_1(ds \setminus \{d\}, \tau)), d)
 \end{aligned}$$

A moment's reflection will indicate that once a previous definition has been reviewed, it will not be possible to review it again without reinvoking the *lookup* command. Another observation is probably in order. Since there is an order in reviewing previous definitions, why is there not an order in viewing definitions? Well, there is no good reason why one should not now do so. A next evolution of the model may replace  $\mathcal{PDEF}$  with—another stack! This would indeed introduce an elegant symmetry into the model, one moreover which would permit the reviewing of previous definitions which had already been previewed, but at the expense that the *next* command might not really give the next definition but one which had already been previewed. Having gone this far, it is clear that we can experiment further by retaining  $\mathcal{PDEF}$  and introduce a second stack to examine whether what we have in mind is indeed useful to a real user.

Returning again to the semantics of these commands, I wish to note that I eschewed introducing stack operation names such as *is-empty*, *push*, *top*, and *pop*. This choice was deliberate and in direct keeping with the method being proposed. It might have been considered that to give 'meaningful' names to the operations is an important tactic in making the specification 'clearer' and, indeed, this is the kind of philosophy behind the proponents for modularity in specifications as has already been mentioned. Nothing could be more disastrous at the specification stage, the creative activity that goes into building the conceptual model. If one has already fixed the specification and no further experimentation is called for, then for the purposes of archiving, by all means modularise and use 'meaningful' names, but only on the understanding that should one wish to revisit the specification for later enhancement then one must go back to the operational forms—the expressions—rather than the operations' names.

The next stage is, of course, to give the behavioural specification. In keeping with the method presented thus far, we will retain the existing model of the

user:  $USER_1$ , where the state is now written in the form  $mk-\Sigma_1(\delta_0, \xi_0)$  and focus exclusively on the evolved agent, now denoted  $AGENT_2$ . Following the VDM functional specification as closely as possible, the required behavioural specification is, therefore:



where

$$\begin{aligned}
 AGENT_2[\sigma_2] &\triangleq \text{let } mk-\Sigma_2(\delta_0, \xi_1) = \sigma_2 \text{ in} \\
 &\quad \text{let } mk-\Xi_1(ds, \sigma_0) = \xi_1 \text{ in} \\
 &\quad \left( Lkp_2?w \rightarrow \chi[[w]]\delta_0 \right. \\
 &\quad \quad \rightarrow \text{let } ds' = \delta_0(w) \text{ in} \\
 &\quad \quad \quad \text{let } \xi'_1 = mk-\Xi_1(ds', \Lambda) \text{ in} \\
 &\quad \quad \quad \text{let } \sigma'_2 = mk-\Sigma_2(\delta_0, \xi_1) \text{ in} \\
 &\quad \quad \quad \quad AGENT_2[Nxt_2(\sigma'_2)] \\
 &\quad \quad \rightarrow Xcptn_2!msg \rightarrow AGENT_2[\sigma_2] \\
 &\quad | Nxt_2? \rightarrow ds \neq \emptyset \\
 &\quad \quad \rightarrow \text{let } d \in ds \text{ in} \\
 &\quad \quad \quad \text{let } \xi'_1 = mk-\Xi_1(ds \setminus \{d\}, \langle d \rangle \wedge \sigma_0) \text{ in} \\
 &\quad \quad \quad \text{let } \sigma'_2 = mk-\Sigma_2(\delta_0, \xi'_1) \text{ in} \\
 &\quad \quad \quad \quad Res_2!d \rightarrow AGENT_2[\sigma'_2] \\
 &\quad \quad \rightarrow Xcptn_2!msg \rightarrow AGENT_2[\sigma_2] \\
 &\quad | Prev_2? \rightarrow \sigma_0 \neq \Lambda \\
 &\quad \quad \rightarrow \text{let } \sigma_0 = \langle d \rangle \wedge \tau \text{ in} \\
 &\quad \quad \quad \text{let } \xi'_1 = mk-\Xi_1(ds \setminus \{d\}, \tau) \text{ in} \\
 &\quad \quad \quad \text{let } \sigma'_2 = mk-\Sigma_2(\delta_0, \xi'_1) \text{ in} \\
 &\quad \quad \quad \quad Res_2!d \rightarrow AGENT_2[\sigma'_2] \\
 &\quad \quad \left. \rightarrow Xcptn_2!msg \rightarrow AGENT_2[\sigma_2] \right)
 \end{aligned}$$

At this point I terminated the *Gedanken* experiment. I had accomplished the goal which I had set before me—to demonstrate that one may use, effectively and naturally, the VDM *Meta-IV* to express concepts, for which notations in CSP or CCS

## COMMUNICATIONS & BEHAVIOUR

are customarily employed. The experiment proved to be a major milestone in my research on conceptual models in computing, a milestone which signalled the end of a particular development in the study of ‘static’ structure and pointed to the beginning of similar work on communication and behaviour. It was time to communicate the results.

### 3. Communicating Results

The *Gedanken* experiment which I have just described brings me right back to the point of departure—the nature of conceptual models in computing. Whereas it might have at first sight seemed to be intuitively obvious that the concept of computing should be more closely related to the concept of computer and all that *that* entails, than to the ‘ordinary’ concepts of the human being who does the computing, it ought to be evident that, in a certain sense, computers, programming languages, etc., are simply projections of the human mind, albeit reified in some fashion that suits us.

In placing emphasis on *user’s* conceptual model, there is always the danger that anthropomorphisms might dominate the more abstract concepts of structure and transformation. It is appealing and pedagogically sound to use terms such as communication, behaviour, and agent, to describe some of the significant aspects of modern computing systems. But, unless there is formal theoretical support for such concepts, which is fortunately the situation, then very little progress will be made.

I am of the opinion that computing theory has been compartmentalised, or partitioned, into discrete domains, each with its own formal notation and language and with adherents who seem to be unwilling or unable to communicate with each other. This situation results as much from the exponential growth of the field within the last thirty years as it does from the attitude that *if you want to get ahead, get a theory*. Even though mathematics today is equally compartmentalised, it did take several millenia to reach such a state. Indeed, I have heard it observed that ‘there is more good mathematics being forgotten than new mathematics being discovered’.

There is now the same danger in computing.

To unify the field of computing it is necessary to find some sort of ‘common language’ for communication. There will *never* be agreement on a common programming language in which to represent algorithms. FORTRAN and Algol once did appear as candidates for such a purpose. They are no longer considered so. The language of mathematics: sets, relations, functions, sequences, cartesian products, etc., is widely used in all branches of theoretical computing and can be the only basis for any kind of common understanding.

I would not like to put forward the argument that the *VDM Meta-IV* ought to be considered as a candidate for the common language. For even within the very narrow domain of formal specifications, there are other candidates, and the *Meta-IV* itself is split into dialects, dialects which are characterised by both style of use and peculiarity of notation. This thesis has added yet another one; but one which is distinctly different from the others. The very term ‘formal’ which is used to qualify both ‘specification’ and ‘method’ seems to have different connotations on either side of the Atlantic. On the European side, it is taken to denote the use of mathematics or logic. Whereas it is ‘natural’ to suppose that logic is more formal than mathematics, it is not surprising to find a greater emphasis on the former by developers of formal specifications. Even the association of boolean algebra with computing machinery might seem to reinforce the significant rôle that logic ought to play. In addition, the development of theorem-proving tools and logic programming languages tilt the balance in its favour. To redress that balance I have focused on the use of an operator calculus for formal specifications.

That ‘notation is a tool of thought’ seems to me to be incontrovertible. Taking the particular notation of the Danish School of the *VDM* as a basis, I have not hesitated to extend and modify it by borrowing extensively from algebra and APL. There was always the danger that ultimately I might end up re-inventing APL, a programming language which suffers from a proliferation of unfamiliar operators and, I might note in passing, that the same sort of problem now besets  $\mathcal{Z}$ .

To have opted for the development of an operator calculus for the *Meta-IV* led to the surprising discovery that the domain of formal specifications was a rich discrete mathematical domain in the traditional sense. Lemmas and theorems quickly emerged from the act of doing proofs and solving problems. I soon found it necessary

## COMMUNICATIONS & BEHAVIOUR

to organise the new material within the algebraic framework of structures. I had sufficient material to build a particular foundation for my view of conceptual models and intended to avoid completely the whole area of communication and behaviour, simply because I could not see how the *Meta-IV* notation might be employed to express my ideas of same. The work described in this Chapter is but an indication for future research directions.

### 4. Directions for Future Research

The development of an operator calculus for the *VDM* is now well-established. The discovery of theorems, and the proofs of correctness with respect to invariants and retrieve functions will be extended to many more formal models of systems. Such work will undoubtedly necessitate the introduction of more powerful operators and probably call for old theorems to be revisited and reproven. I have only briefly indicated that the *VDM* may also be employed effectively outside of its usual domain of application. The Theory of Formal Languages, Automata Theory, and Graph Theory, are domains that immediately spring to mind. In the previous section I suggested that ‘formal’ carried connotations of the mathematical and the logical. There is another sense of formal that is particularly apt—there are no hidden assumptions. Everything is exposed down to the smallest detail. There are many theorems and proofs, in Formal Language Theory, for example, which ought to be ‘formalised’ using the *VDM Meta-IV*.

Knuth’s extant three volumes on the *Art of Computer Programming* will always find a place in the literature of computing. Undoubtedly, there will come a time when even his flowchart algorithms and the MIX machine will be of historical interest. But Knuth was and is, at heart, a mathematician.

“Even though computers are widely regarded as belonging to the domain of ‘applied mathematics’, there are ‘pure mathematicians’ such as myself who have found many intriguing connections between computers and abstract mathematics. From this standpoint, parts of these books may be thought

of as ‘a pure mathematician’s view of computers’” (Knuth 1973, 1:ix).

Bjørner is effectively using the *VDM* to show that many diverse aspects of software systems may be described within a single notational framework and has proposed a six volume work to record his findings. However, it is clear that the size of the task is beyond any single human being and that Computer Science is much in need of a mathematical Bourbaki.

There is still an enormous abyss that separates the practising software engineers from those who work in formal specifications. Classical engineers rely on classical mathematics, but have the advantage that the latter is a very mature field. Even though the *VDM* is still arguably the most widely used formal *method* in software engineering, and here I emphasise method in contrast to notation or language, there is the unfortunate technological trend within the computing community to invent new ‘methods’ and to suppose that computer-tool support will solve many of their problems. I have demonstrated that the method of the *VDM* may be extended to cover communication and behaviour *just as well* as it does structure. Enormous effort and energy are being expended to supplant the *VDM* with RAISE in the belief that the supposed defects of the former, with respect to communication and behaviour on the one hand, and modular specifications on the other, have been removed by the latter. I believe that this effort is misdirected. The seeds of a much more interesting and important research goal for formal specification was set down by Hoare (Bjørner, Hoare and Langmaack 1990, ix). He proposed that we find “obvious and definitive answers” to the following basic questions:

1. Is one notation (or the other, or both) adequate for both specification and development? [*Hoare was referring to the notations of the English School of the VDM and Z*].
2. If distinct notations are desirable, can this be justified by appeal to general scientific, mathematical or engineering principles?
3. In the latter case, what measures are recommended for easy and reliable transition between notations?

There is a certain amount of parochialism behind the questions. I have successfully demonstrated in this thesis that the operator calculus notation and method of the Irish School of the *VDM* already answers the questions and more.

For the specification of communication and behaviour, the important issues that

## COMMUNICATIONS & BEHAVIOUR

need to be addressed are (1) the modelling of time, and (2) the application of reification. My proposal for the latter is currently dictated by the natural progression of

‘structure  $\longrightarrow$  communication  $\longrightarrow$  behaviour’

Reification of structure should give corresponding reification of communication and behaviour. I have not begun to address the issue of time.

The development of formal specifications under software product-deadline is not conducive to the establishment of a sound body of discrete mathematics that is of enduring significance. Few, if any, formal specifications are revisited and polished. This is in stark contrast to the development of classical mathematics. Currently, such specifications are application driven and may be said to belong to the domain of applied discrete mathematics. There is a need to develop the field of ‘pure’ formal specifications, some elements of which I have presented in this thesis. Finally, I believe that I have met the ultimate goal that research should lead to results that endure rather than those which are ephemeral.

# Appendix A

## The Dictionary

### 1. Introduction

The concept of *dictionary* is a familiar one. I have chosen it as the basic example for illustrating what I mean by *model* and *conceptual model* in the thesis. In this Appendix, I present the details of a sequence of *Meta-IV* specifications of a dictionary to which I have already frequently referred, to illustrate aspects of my method. In particular an exhaustive demonstration of what I mean by the use of an operator calculus is presented, a demonstration which enables me to point out how many important lemmas, used in the operator calculus of the Irish *VDM*, arose in practice.

The dictionary was also used extensively by Jones to demonstrate reification, and proof by retrieve functions (1986, 205). Thus, my method may be contrasted directly with his. Referring back to Chapter 8, on Communications and Behaviour, one will realize, that when I say that this Appendix contains a sequence of specifications of a dictionary, that I am really talking about concrete realizations of very general concepts. Thus, for example, I am also discussing here the concept of *symbol table* as used in compiler theory, or the *environment* of denotational semantics which is briefly presented in Appendix B. It is really the abstract structure which matters and not the details of the names of the domains and operations.

I present this material in the context of the thesis as a whole, i.e., I am assuming that the philosophy and method of the Irish School of the *VDM* are already understood. Therefore, I take the liberty of being more brief than usual. That the material should be self-contained, I have unavoidably included some repetition.

# THE DICTIONARY

## 2. Model 0 — Set

The model of a dictionary to be considered here in this section conforms to the concept of a dictionary as a ‘spelling checker’. Although, operations on such a dictionary are presented in an order such as one might expect to find in an equivalent abstract data type specification, I would emphasise that conceptually, the really interesting operations are a) the ability to check if a word is already in the dictionary— $Lkp_0$ , and b) the ability to enter new words into an existing dictionary— $Ent_0$ , and in that order. The appropriate model that is a) sufficiently abstract, and b) adequate for the conceptual model of ‘spelling checker’ is to consider a dictionary  $\delta$  to be simply a set of words:

$$DICT_0 = \mathcal{P}WORD$$

where the domain  $WORD$  is not further specified but that it may be assumed that it contains elements which conform with the usual understanding of ‘words’. In other words I rely on intuition to supply the intended meaning. Abstractly, this section is a study of powersets:

$$MODEL_0 = \mathcal{P}X$$

### 2.1. The Operations

The operations that might be expected for such a dictionary are:

1. Creation of a new empty dictionary:  $New_0$ ;
2. Entry of a (new) word into a dictionary:  $Ent_0$ ;
3. Removal of a word from a dictionary:  $Rem_0$ ;
4. Looking up a word in a dictionary:  $Lkp_0$ ;
5. Determining the size of a dictionary:  $Size_0$ .

#### 2.1.1. The New Command

In this specification a new dictionary is considered to be empty:

$$\begin{aligned} New_0 &: \longrightarrow DICT_0 \\ New_0 &\triangleq \emptyset \end{aligned}$$

Technically,  $New_0$  is a constant function. In the context of  $\mathcal{P}X$ , this is simply a statement that  $\emptyset \in \mathcal{P}X$ .

### 2.1.2. The Enter Command

New words may be entered into the dictionary one at a time. Note, however, that the specification given here does not capture this notion exactly, in so far that attempts to add duplicates have no effect due to the nature of the union operator. This is a deliberate choice, not to use some sort of pre-condition. As will be seen later, I will be forced to alter this specification, as a consequence of following any reification path that incorporates the sequence domain. A similar ‘problem’ occurs when I choose to use symmetric difference in place of set union in  $DICT_1$ :

$$\begin{aligned} Ent_0: WORD &\longrightarrow (DICT_0 \longrightarrow DICT_0) \\ Ent_0[[w]]\delta &\triangleq \{w\} \cup \delta \end{aligned}$$

### 2.1.3. The Remove Command

A word may be removed from the dictionary. Note again that there is no pre-condition. Again I rely exclusively on the concept of set difference to cover all eventualities:

$$\begin{aligned} Rem_0: WORD &\longrightarrow (DICT_0 \longrightarrow DICT_0) \\ Rem_0[[w]]\delta &\triangleq \{w\} \leftarrow \delta \end{aligned}$$

### 2.1.4. The Lookup Command

This is the most significant operation from the viewpoint of the user who is only interested in checking the spelling of a word:

$$\begin{aligned} Lkp_0: WORD &\longrightarrow (DICT_0 \longrightarrow \mathbf{B}) \\ Lkp_0[[w]]\delta &\triangleq \chi[[w]]\delta \end{aligned}$$

Conceptually, the command is a ‘search’ at the highest level of abstraction.

### 2.1.5. The Size Command

One may wish to know how big the dictionary is:

$$\begin{aligned} Size_0: DICT_0 &\longrightarrow \mathbf{N} \\ Size_0(\delta) &\triangleq card \delta \end{aligned}$$

**Remarks:** From a conceptual point of view, the basic set operations are given a ‘real-world’ interpretation. Thus, set union is used to add a word, set difference is used to remove a word. In a sense, the dictionary model is just one interpretation of

## THE DICTIONARY

the behaviour of a set under certain operations. A conceptual difficulty that usually arises is due to the usual understanding of a dictionary as a collection of words in lexicographic order! From the point of view of the user who is only interested in checking spelling, order is not important. One just wants to know whether a word is present or not. For this purpose the set model is abstractly adequate.

### 2.2. Development

Now it is tempting to halt the specification effort once the basic model and operations have been defined. However, this would be quite wrong! The whole point of the model is to have a basis, a starting point, for experimentation and exploration. Computer Scientists working in the area of formal specifications need to learn this fundamental lesson, one which is taken very much for granted by mathematicians, for example. Now clearly, one of the interesting questions that immediately comes to mind, is ‘Can we enter more than one word at a time into the dictionary’? Formally, this may be specified by

$$\begin{aligned} \circ/Ent_0: \mathcal{P}WORD &\longrightarrow \mathcal{P}WORD \longrightarrow \mathcal{P}WORD \\ \circ/Ent_0[[ws]]\delta &\triangleq ws \cup \delta \end{aligned}$$

where I have chosen to denote the extended ‘enter’ operation by  $\circ/Ent_0$  and replaced the name  $DICT_0$  with its corresponding structure  $\mathcal{P}WORD$ . From the chapter on tail-recursion, it is immediately clear that  $\circ/Ent_0$  is the set union operator:

$$\circ/Ent_0[[ws']] \circ \circ/Ent_0[[ws]] = \circ/Ent_0[[ws' \cup ws]]$$

Indeed, from the form of the signature, one may interpret this function as a merge operation on dictionaries:

$$\begin{aligned} merge(\delta_1, \delta_2) &\triangleq \circ/Ent_0[[\delta_1]]\delta_2 \\ &= \delta_1 \cup \delta_2 \end{aligned}$$

and then generalized in the obvious way to merge sets of dictionaries using the distributed union operator:

$$merge/\delta_s = \cup/\delta_s$$

Technically, we have just shifted up a level of structure to  $\mathcal{P}MODEL_0 = \mathcal{P}\mathcal{P}X$ . Similarly, we may generalize the remove command to obtain

$$\begin{aligned} \circ/Rem_0: \mathcal{P}WORD &\longrightarrow \mathcal{P}WORD \longrightarrow \mathcal{P}WORD \\ \circ/Rem_0[[ws]]\delta &\triangleq ws \triangleleft \delta \end{aligned}$$

and the symmetric difference of two dictionaries  $\delta_1$  and  $\delta_2$  may then be expressed in the form

$$\delta_1 \triangle \delta_2 = \text{merge}({}^\circ/\text{Rem}_0[\delta_1]\delta_2, {}^\circ/\text{Rem}_0[\delta_2]\delta_1)$$

Finally, consider generalising the lookup command. Instead of ‘searching’ for a single word, we wish to lookup a set of words.

$$\begin{aligned} {}^\circ/\text{Lkp}_0: \mathcal{P}\text{WORD} &\longrightarrow \mathcal{P}\text{WORD} \longrightarrow \mathbf{B} \\ {}^\circ/\text{Lkp}_0[\text{ws}]\delta_0 &\triangleq \text{ws} \subseteq \delta_0 \end{aligned}$$

Thus, the subset operator is simply a generalisation of the characteristic function:

$$\chi[\text{ws}]\delta = \text{ws} \subseteq \delta$$

Now it is of interest to demonstrate how one may obtain a corresponding particular operational form. Let us suppose that we have

$$\text{ws} = \{w_1, w_2, \dots, w_j, \dots, w_n\}$$

Then  $\text{ws} \subseteq \delta = \chi[\text{ws}]\delta$  may be expressed as

$$\begin{aligned} \chi[\text{ws}]\delta &= \chi[\{w_1, w_2, \dots, w_j, \dots, w_n\}]\delta \\ &= \chi[w_1]\delta \wedge \chi[w_2]\delta \wedge \dots \wedge \chi[w_j]\delta \wedge \dots \wedge \chi[w_n]\delta \\ &= \wedge/\{\chi[w_1]\delta, \chi[w_2]\delta, \dots, \chi[w_j]\delta, \dots, \chi[w_n]\delta\} \\ &= \wedge/\{\chi[w_1], \chi[w_2], \dots, \chi[w_j], \dots, \chi[w_n]\}\delta \\ &= \wedge/\mathcal{P}\chi[-]\{w_1, w_2, \dots, w_j, \dots, w_n\}\delta \\ &= \wedge/(\mathcal{P}\chi[-]\text{ws})\delta \end{aligned}$$

giving the lemma for subset

$$\text{LEMMA A.1. } S \subseteq X = \wedge/(\mathcal{P}\chi[-]S)X.$$

It is in just this manner that many of the results of the Irish *VDM* were obtained—a simple example of following the advice of Pólya in *How to Solve It* ([1945] 1957).

# THE DICTIONARY

## 3. Model 1 — Set Partition

Model 0 of the spelling checker dictionary was basically ‘flat’. I would now like to consider structuring the dictionary in the following sense. Words which share some common characteristic are to be grouped together into a subset of the dictionary called a partition. For example, we might suppose that words consist of letters and those words which have the same first letter might be in the same partition which is ‘named’ by that letter. This is a lexicographical ordering approach. Alternatively, we might assume the existence of a function  $h$  which maps words into a finite subset of the natural numbers, say  $\{1 \dots p\}$ , where  $p$  is a prime. Then the words  $w$ , for which  $h(w) = j$ , will be grouped together in partition  $j$ . This models the notion of hashing. Naturally, there are many other ways in which we might group words. In all cases, we will use a ‘magic’ function  $f$  that assigns a given word  $w$  to one and only one partition  $f(w)$ . In other words I do not wish to commit to any particular design choice at this stage. I only wish to introduce the concept of partitioning. Letting  $PART$  denote the domain of partitions, the refined model is then given by:

$$DICT_1 = PART^*$$

$$PART = \mathcal{P}WORD$$

As will be seen later on, the concept of ‘pagination’ is essentially a partitioning of a structure/model. A typical partitioned dictionary  $\delta$  has the form

$$\delta = \langle P_1, P_2, \dots, P_j, \dots, P_n \rangle$$

$$P_j = \{w_{j1}, w_{j2}, \dots, w_{jk}, \dots, w_{jm}\}$$

Abstractly, we are considering the model

$$MODEL_1 = (\mathcal{P}X)^*$$

which is the basis of the set partitioning algorithm in (Aho, Hopcroft and Ullman 1974, 157). A different, though related, set of results would be obtained by considering the more general model of partitioning:  $\mathcal{P}\mathcal{P}X$ .

3.1. The Invariant

Recall that the invariant is used to determine precisely the domain of discourse. Again, for the purpose of formal specification, and not wishing to commit too much to a particular design, I propose to use a reasonably weak invariant at this stage, to wit, that a) the partitions are indeed such in the formal sense of set theory—the partitions are disjoint subsets of the ‘flat’ dictionary:

$$\begin{aligned} \text{inv-DICT}_1: \text{DICT}_1 &\longrightarrow \mathbf{B} \\ \text{inv-DICT}_1(\delta) &\triangleq \\ &(\forall j, k \in \delta)(\delta[j] \cap \delta[k] = \emptyset) \end{aligned}$$

and b) there is at least one partition:

$$\begin{aligned} \text{inv-DICT}_1: \text{DICT}_1 &\longrightarrow \mathbf{B} \\ \text{inv-DICT}_1(\delta) &\triangleq \text{len } \delta \geq 1 \end{aligned}$$

The invariant is, therefore, the logical conjunction of these two. In the Irish School of the *VDM*, the first part of the invariant is in a form which is not directly amenable to the operator calculus. I have chosen to use the following form as the basis for the first part of the invariant:

$$\text{card} \circ \Delta / \delta = + / \circ \text{card}^* \delta$$

The relationship may be expressed succinctly by the commuting diagram:

$$\begin{array}{ccc} \mathcal{P}X & \xrightarrow{\text{card}} & \mathbf{N} \\ \uparrow \Delta / & & \uparrow + / \\ (\mathcal{P}X)^* & \xrightarrow{\text{card}^*} & \mathbf{N}^* \end{array}$$

It states that if I take the union of all the partitions  $\Delta / \delta$  then the cardinality of the resulting set is equal to the sum,  $+ /$ , of all the cardinalities in each partition  $\text{card}^* \delta$ . Technically,  $+ / \circ \text{card}^*$  maps the problem into the partition problem of number theory. For example, let  $S = \{a, b, c, d\}$ , with  $\text{card } S = 4$ , and  $\delta = \langle \{a\}, \{b, c\}, \{d\} \rangle$ . Then,  $\text{card}^* \delta = \langle 1, 2, 1 \rangle$  and

$$+ / \text{card}^* \delta = 1 + 2 + 1$$

a partition of 4. At first I had supposed that  $\Delta /$  alone would have been adequate to capture the invariant. This was based on the observation that if there was an element in common to  $P_i \Delta P_j$ , say, then  $\Delta /$  would not be able to recover the

## THE DICTIONARY

set. Clearly,  $\cup/$  would have been inadequate in such a case. However, a simple counter-example exposed the weakness of my conjecture:

$$\Delta/\langle \{a\}, \{a, b, c\}, \{a, d\} \rangle = \{a, b, c, d\}$$

The problem arises from the ‘odd-even’ nature of symmetric difference, which was used to effect in the polyline scanfill algorithm. An even number of occurrences of a common element removes it, an odd number of common occurrences adds it. The observation that the invariant for the set partition problem was isomorphic to the partition problem in number theory led to the final result.

In the abstract model  $\mathcal{PPX}$ , the problem of determining the invariant is much more complex since an analogous approach leads to a loss of information. Consider the same example where  $\delta = \{\{a\}, \{b, c\}, \{d\}\}$ . Clearly,  $\Delta/\delta = S$ . But the counterpart to  $\text{card}^*\delta$  gives

$$\begin{aligned} \mathcal{P} \text{card} \delta &= \mathcal{P} \text{card} \{\{a\}, \{b, c\}, \{d\}\} \\ &= \{\text{card} \{a\}, \text{card} \{b, c\}, \text{card} \{d\}\} \\ &= \{1, 2\} \end{aligned}$$

the loss of information being due to the uniqueness property of sets. In such cases, the bag or multiset may be used to overcome the difficulty—an important observation for the method. Specifically, let us define an injection operator  $j: \mathcal{PS} \setminus \{\emptyset\} \longrightarrow (\mathcal{PS} \setminus \{\emptyset\} \xrightarrow{m} \mathbf{N}_1)$  which maps a set  $S \in \mathcal{PS} \setminus \{\emptyset\}$  into a bag:

$$j: S \mapsto [S \mapsto |S|]$$

Then the corresponding invariant for this model is

$$\text{card} \circ \Delta/\delta = |\oplus/ \circ \mathcal{P}j\delta|$$

where  $\oplus$  denotes the usual bag addition operator and  $|-|$  denotes the size of the bag.

Before leaving this discourse on the invariant, I wish to demonstrate that such considerations as we have been carrying out lead to useful lemmas in the operator calculus:

LEMMA A.2. *The composite map  $\text{card} \circ \Delta/$  is a homomorphism of  $DICT_1$ .*

Formally, let  $\delta$  be a partitioned dictionary of the form  $\delta_l \wedge \delta_r$ . Then

$$\text{card} \circ \Delta/(\delta_l \wedge \delta_r) = (\text{card} \circ \Delta/\delta_l) + (\text{card} \circ \Delta/\delta_r)$$

*Proof:*

$$\begin{aligned} \text{card} \circ^\Delta / (\delta_l \wedge \delta_r) &= \text{card}(\Delta / \delta_l \triangle \Delta / \delta_r) \\ &= \text{card}(\Delta / \delta_l) + \text{card}(\Delta / \delta_r) \\ &= (\text{card} \circ^\Delta / \delta_l) + (\text{card} \circ^\Delta / \delta_r) \end{aligned}$$

The validity of the proof, specifically that  $\text{card}(S_1 \triangle S_2) = \text{card} S_1 + \text{card} S_2$ , depends critically on the invariant. Now, if we have a partitioned dictionary of the form  $\delta = \delta_l \wedge \langle P_j \rangle \wedge \delta_r$ , then by the preceding lemma, it follows that

$$\begin{aligned} \text{card} \circ^\Delta / (\delta_l \wedge \langle P_j \rangle \wedge \delta_r) &= \text{card} \circ^\Delta / \delta_l + \text{card} \circ^\Delta / \langle P_j \rangle + \text{card} \circ^\Delta / \delta_r \\ &= \text{card} \circ^\Delta / \delta_l + \text{card} \delta_{h(w)} + \text{card} \circ^\Delta / \delta_r \end{aligned}$$

This second lemma is similar to the first:

LEMMA A.3. *The composite map  $+ / \circ \text{card}^*$  is a homomorphism of  $\text{DICT}_1$ .*

Formally, let  $\delta$  be a partitioned dictionary of the form  $\delta_l \wedge \delta_r$ . Then

$$+ / \circ \text{card}^*(\delta_l \wedge \delta_r) = (+ / \circ \text{card}^* \delta_l) + (+ / \circ \text{card}^* \delta_r)$$

*Proof:*

$$\begin{aligned} + / \circ \text{card}^*(\delta_l \wedge \delta_r) &= + / (\text{card}^* \delta_l \wedge \text{card}^* \delta_r) \\ &= + / (\text{card}^* \delta_l) + + / (\text{card}^* \delta_r) \\ &= (+ / \circ \text{card}^* \delta_l) + (+ / \circ \text{card}^* \delta_r) \end{aligned}$$

Again considering a partitioned dictionary of the form  $\delta = \delta_l \wedge \langle P_j \rangle \wedge \delta_r$ , then by the preceding lemma, it follows that

$$\begin{aligned} + / \circ \text{card}^*(\delta_l \wedge \langle P_j \rangle \wedge \delta_r) &= (+ / \circ \text{card}^* \delta_l) + (+ / \circ \text{card}^* \langle P_j \rangle) + (+ / \circ \text{card}^* \delta_r) \\ &= + / \circ \text{card}^* \delta_l + \text{card} \delta_{h(w)} + + / \circ \text{card}^* \delta_r \end{aligned}$$

As a consequence of these two lemmas, the first part of the invariant may be restated in the form

$$\begin{aligned} (\text{card} \circ^\Delta / \delta_l) + (\text{card} \circ^\Delta / \langle P_j \rangle) + (\text{card} \circ^\Delta / \delta_r) &= (+ / \circ \text{card}^* \delta_l) + (+ / \circ \text{card}^* \langle P_j \rangle) \\ &\quad + (+ / \circ \text{card}^* \delta_r) \end{aligned}$$

and after simplifying

$$\begin{aligned} (\text{card} \circ^\Delta / \delta_l) + (\text{card} P_j) + (\text{card} \circ^\Delta / \delta_r) &= (+ / \circ \text{card}^* \delta_l) + (\text{card} P_j) \\ &\quad + (+ / \circ \text{card}^* \delta_r) \end{aligned}$$

which gives a structural induction form of the invariant for a partitioned dictionary.

## THE DICTIONARY

### 3.2. The Retrieve Function

Before proceeding with the specification of the operations it is to be noted that in practice, the construction of same may be guided by the ‘retrieve’ function. Although, logically it belongs in the next section, I introduce it here to illustrate the point. It is, of course, rather obvious—to take the union of all the partitions:

$$\begin{aligned}\mathfrak{R}_{10}: DICT_1 &\longrightarrow DICT_2 \\ \mathfrak{R}_{10}(\delta) &\triangleq \bigcup \delta\end{aligned}$$

Note that the retrieve function actually figures in the invariant which might now be written as

$$card \circ \mathfrak{R}_{10}(\delta) = + / \circ card^* \delta$$

Finally, observing that  $Size_0 \delta_0 = card \delta_0$ , and looking ahead to the specification of  $Size_1$ , we note that it permits us to write the invariant even more concisely

$$Size_0 \circ \mathfrak{R}_{10}(\delta) = Size_1(\delta)$$

which is actually what is to be proved in the case of the  $Size_1$  operation under the retrieve function. Thus, the ‘Size’ command, which initially seemed to be an interesting operation to perform on a dictionary turns out to exhibit a fundamental property of partitioned dictionaries.

### 3.3. The Operations

The commands to be performed on this model are almost exactly the same as those for  $DICT_0$ . The only significant difference is the ‘new’ command which is defined to construct a partition of fixed size.

#### 3.3.1. The New Command

The new command introduces an empty partitioned set. I will model this by fixing the number of partitions  $p$ ,  $p \geq 1$ :

$$\begin{aligned}New_1: \mathbf{N}_1 &\longrightarrow DICT_1 \\ New_1(n) &\triangleq \langle P_j \mid P_j = \emptyset, 1 \leq j \leq n \rangle\end{aligned}$$

Note that I have effectively taken care of the second part of the invariant by ensuring that there is at least one partition. The only other place where I need to check the invariant is in the ‘remove’ command. Recall that the sequence constructor

$\langle P_j \mid P_j = \emptyset, 1 \leq j \leq n \rangle$  produces the ordered sequence

$$\langle P_1, \dots, P_j, \dots, P_n \rangle$$

To check the first part of the invariant, I simply compute:

$$\begin{aligned} + / \circ \text{card}^* \text{New}_1(n) &= + / \circ \text{card}^* \langle P_j \mid P_j = \emptyset, 1 \leq j \leq n \rangle \\ &= + / \langle \text{card } P_j \mid P_j = \emptyset, 1 \leq j \leq n \rangle \\ &= + / \langle v_j \mid v_j = 0, 1 \leq j \leq n \rangle \\ &= 0 \end{aligned}$$

and

$$\begin{aligned} \text{card} \circ^\cup / \text{New}_1(n) &= \text{card} \circ^\cup / \langle P_j \mid P_j = \emptyset, 1 \leq j \leq n \rangle \\ &= \text{card } \emptyset \\ &= 0 \end{aligned}$$

I have chosen to use  $\circ^\cup /$  rather than  $\Delta /$  in this case. Their effect is equivalent.

### 3.3.2. The Enter Command

The enter command takes advantage of a ‘magic’ function  $h$  to pick out the correct partition. Note that I have not included this function in the signature.

$$\begin{aligned} \text{Ent}_1: \text{WORD} &\longrightarrow (\text{DICT}_1 \longrightarrow \text{DICT}_1) \\ \text{Ent}_1 \llbracket w \rrbracket \delta &\triangleq \\ \text{let } \delta &= \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \text{ in } \delta_l \wedge \langle \delta_{h(w)} \Delta \{w\} \rangle \wedge \delta_r \end{aligned}$$

A word on the form ‘let  $\delta = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r$  in ...’ is in order. This is my standard abbreviation for the sequence of ‘let’ expressions:

$$\begin{aligned} \text{let } j &= h(w) \text{ in} \\ \text{let } P_j &= \delta[j] \text{ in} \\ \text{let } \delta &= \delta_l \wedge \langle P_j \rangle \wedge \delta_r \text{ in} \\ &\dots \end{aligned}$$

Elimination of these would give

$$\text{let } \delta = \delta_l \wedge \langle \delta[h(w)] \rangle \wedge \delta_r \text{ in } \dots$$

and I prefer to write  $\delta_{h(w)}$  in place of  $\delta[h(w)]$ . The  $\text{Ent}_1$  command is subject to the constraint

$$\begin{aligned} \text{pre-Ent}_1: \text{WORD} &\longrightarrow (\text{DICT}_1 \longrightarrow \mathbf{B}) \\ \text{pre-Ent}_1 \llbracket w \rrbracket \delta &\triangleq \\ \text{let } \delta &= \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \text{ in } \neg \chi \llbracket w \rrbracket \delta_{h(w)} \end{aligned}$$

## THE DICTIONARY

The constraint here is essential to guarantee the validity of the result using the symmetric difference operator. However, we now have a pre-condition for which there is no counterpart in the original abstract model, a point mentioned in the context of the  $Ent_0$  command specification. In view of the lemmas above, the only interesting point to note, with respect to the invariant, is that

$$card \langle \delta_{h(w)} \Delta \{w\} \rangle = card \delta_{h(w)} + 1$$

### 3.3.3. The Remove Command

The remove command is used to delete a word from the appropriate partition:

$$Rem_1: WORD \longrightarrow (DICT_1 \longrightarrow DICT_1)$$

$$Rem_1[[w]]\delta \triangleq$$

$$\text{let } \delta = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \text{ in } \delta_l \wedge \langle \delta_{h(w)} \Delta \{w\} \rangle \wedge \delta_r$$

Again I am going to take advantage of the properties of the symmetric difference operator. The corresponding pre-condition is obvious. Again with respect to the invariant we note that

$$card \langle \delta_{h(w)} \Delta \{w\} \rangle = card \delta_{h(w)} - 1$$

### 3.3.4. The Lookup Command

In searching for a word, we must first identify the appropriate partition. It is clear that the process of reification is moving us into a high-level design phase.

$$Lkp_1: WORD \longrightarrow (DICT_1 \longrightarrow \mathbf{B})$$

$$Lkp_1[[w]]\delta \triangleq$$

$$\text{let } \delta = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \text{ in } \chi[[w]]\delta_{h(w)}$$

### 3.3.5. The Size Command

I have already noted the significance of this operation with respect to the invariant above.

$$Size_1: DICT_1 \longrightarrow \mathbf{N}$$

$$Size_1(\delta) \triangleq + / \circ card^* \delta$$

With respect to this specification, everything depends on the properties of the ‘magic’ function  $h : WORD \longrightarrow \{1 \dots n\}$ , where  $n$  is the argument to the  $New_1$

## *Formal Relationship of Model 1 to Model 0*

command. The most important property of  $h$  is to guarantee that a word  $w$  is assigned to a unique partition. It is already assumed that one is familiar with hashing and that such a function may be found (Knuth 1973, 3). However, there is nothing in the specification that forces us to move to a design stage where we are committed to using hashing. Any function with the property that guarantees to assign each word to a unique partition will suffice.

On the other hand, assuming without loss of generality, that a hashing function is indeed intended, then what might we say about the particular model that we have chosen? Assuming that  $n$  is reasonably big, but not too big, then we must conclude that if the dictionary is to be of any use at all, either a) the words must be such that each partition consists of at most one word and, hence, the number of words is less than or equal to  $n$ —this is the case for the ‘dictionary’ of key words in a programming language such as Pascal or Ada, or b) there are ‘collisions’ and *PWORD* models ‘overflow’.

### 4. Formal Relationship of Model 1 to Model 0

The purpose of this section is to demonstrate in detail that  $DICT_1$  is a reification of  $DICT_0$ . This is accomplished mainly by verifying that the retrieve function commutes with the model operators, i.e., in establishing the correctness of certain commuting diagrams. The details of the proofs may seem tedious. But they are precisely the means by which certain facts about the *VDM Meta-IV* models and operators are learned. In short, they serve as a ‘mnemotechnic’ system (Pólya [1945] 1957, 217).

For the Irish School of the *VDM* the proofs give rise to fundamental lemmas which are frequently employed in specifications. First, let us consider the retrieve function.

# THE DICTIONARY

## 4.1. The Retrieve Function

It is customary to use distributed union as a retrieve function for a partitioned dictionary (Jones 1986, 209). I usually do so myself. However, I believe that that is the case solely because one is more familiar with set union and its properties rather than symmetric difference. Reduction with respect to symmetric difference is more precise:

$$\begin{aligned} \mathfrak{R}_{10}: DICT_1 &\longrightarrow DICT_0 \\ \mathfrak{R}_{10}(\delta) &\triangleq \Delta/\delta \end{aligned}$$

As a consequence of experience with proofs in this model, the following lemma has turned out to be fundamental:

LEMMA A.4. *The retrieve function  $\mathfrak{R}_{10} = \Delta/$  is a homomorphism of the dictionary  $DICT_1$ .*

Formally, for a dictionary  $\delta = \delta_l \wedge \delta_r$ , we have

$$\mathfrak{R}_{10}(\delta_l \wedge \delta_r) = (\mathfrak{R}_{10}(\delta_l)) \Delta (\mathfrak{R}_{10}(\delta_r))$$

We have already used this fact before. Now consider a dictionary of the form  $\delta = \delta_l \wedge \{w \Delta P_j\} \wedge \delta_r$ . Then,

$$\mathfrak{R}_{10}(\delta_l \wedge \langle \{w\} \Delta P_j \rangle \wedge \delta_r) = \{w\} \Delta \mathfrak{R}_{10}(\delta_l \wedge \langle P_j \rangle \wedge \delta_r)$$

*Proof:*

$$\begin{aligned} \mathfrak{R}_{10}(\delta_l \wedge \langle \{w\} \Delta P_j \rangle \wedge \delta_r) &= \mathfrak{R}_{10}\delta_l \Delta \mathfrak{R}_{10}\langle \{w\} \Delta P_j \rangle \Delta \mathfrak{R}_{10}\delta_r \\ &= \mathfrak{R}_{10}\delta_l \Delta \{w\} \Delta \mathfrak{R}_{10}\langle P_j \rangle \Delta \mathfrak{R}_{10}\delta_r \\ &= \{w\} \Delta \mathfrak{R}_{10}\delta_l \Delta \mathfrak{R}_{10}\langle P_j \rangle \Delta \mathfrak{R}_{10}\delta_r \\ &= \{w\} \Delta \mathfrak{R}_{10}(\delta_l \wedge \langle P_j \rangle \wedge \delta_r) \end{aligned}$$

Q. E. D.

## Formal Relationship of Model 1 to Model 0

### 4.2. The New Operation

For simplicity I use  $\mathcal{D}_j$  in place of  $DICT_j$  in the commuting diagrams

$$\begin{array}{ccc}
 & \xrightarrow{New_0} & \mathcal{D}_0 \\
 & & \uparrow \mathfrak{R}_{10} \\
 \mathbf{N}_1 & \xrightarrow{New_1(n)} & \mathcal{D}_1 \\
 New_1(n) \triangleq \langle P_j \mid P_j = \emptyset, 1 \leq j \leq n \rangle & & 
 \end{array}$$

▷ Required to show that

$$(\mathfrak{R}_{10} \circ New_1(n)) = New_0$$

▷ *Proof:*

$$\begin{aligned}
 & (\mathfrak{R}_{10} \circ New_1(p)) \\
 &= \mathfrak{R}_{10}(\langle P_j \mid P_j = \emptyset, 1 \leq j \leq n \rangle) \\
 &= \triangle / (\langle P_j \mid P_j = \emptyset, 1 \leq j \leq n \rangle) \\
 &= \emptyset \quad \text{Q.E.D.}
 \end{aligned}$$

### 4.3. The Enter Operation

$$\begin{array}{ccc}
 Ent_0[[w]]\delta_0 \triangleq \{w\} \cup \delta_0 & & \\
 \mathcal{D}_0 & \xrightarrow{Ent_0[[w]]} & \mathcal{D}_0 \\
 \uparrow \mathfrak{R}_{10} & & \uparrow \mathfrak{R}_{10} \\
 \mathcal{D}_1 & \xrightarrow{Ent_1[[w]]} & \mathcal{D}_1 \\
 Ent_1[[w]]\delta_1 \triangleq \text{let } \delta_1 = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \text{ in } \delta_l \wedge \langle \{w\} \triangle \delta_{h(w)} \rangle \wedge \delta_r & & 
 \end{array}$$

▷ Required to show that

$$(Ent_0[[w]] \circ \mathfrak{R}_{10})\delta_1 = (\mathfrak{R}_{10} \circ Ent_1[[w]])\delta_1$$

## THE DICTIONARY

▷ *Proof:*

▷ LHS

$$\begin{aligned} & (Ent_0[[w]] \circ \mathfrak{R}_{10})\delta_1 \\ &= Ent_0[[w]](\mathfrak{R}_{10}(\delta_1)) \\ &= \{w\} \cup \mathfrak{R}_{10}(\delta_1) \end{aligned}$$

Note that I do not bother to expand this any further by applying the retrieve function. Expansion is not usually necessary, since the RHS will be shown to reduce to this form.

▷ RHS

$$\begin{aligned} & (\mathfrak{R}_{10} \circ Ent_1[[w]])\delta_1 \\ &= \mathfrak{R}_{10}(Ent_1[[w]]\delta_1) \\ &= \mathfrak{R}_{10}(\text{let } \delta_1 = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \\ &\quad \text{in } \delta_l \wedge \langle \{w\} \triangle \delta_{h(w)} \rangle \wedge \delta_r) \\ &= \text{let } \delta_1 = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \\ &\quad \text{in } \mathfrak{R}_{10}(\delta_l \wedge \langle \{w\} \triangle \delta_{h(w)} \rangle \wedge \delta_r) \\ &= \{w\} \triangle \mathfrak{R}_{10}(\delta_1) \text{ by the above lemma.} \end{aligned}$$

Equality is guaranteed by the pre-condition.

### 4.4. The Remove Operation

$$Rem_0[[w]]\delta_0 \triangleq \{w\} \wp \delta_0$$

$$\begin{array}{ccc} \mathcal{D}_0 & \xrightarrow{Rem_0[[w]]} & \mathcal{D}_0 \\ \uparrow \mathfrak{R}_{10} & & \uparrow \mathfrak{R}_{10} \\ \mathcal{D}_1 & \xrightarrow{Rem_1[[w]]} & \mathcal{D}_1 \end{array}$$

$$Rem_1[[w]]\delta_1 \triangleq \text{let } \delta_1 = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \text{ in } \delta_l \wedge \langle \{w\} \triangle \delta_{h(w)} \rangle \wedge \delta_r$$

Since I am using symmetric difference to express removal in  $DICT_1$ , then there is nothing more to prove. The commands  $Ent_1$  and  $Rem_1$  are distinguished solely by their pre-conditions.

## Formal Relationship of Model 1 to Model 0

### 4.5. The Lookup Operation

$$\begin{array}{ccc}
 Lkp_0[[w]]\delta_0 & \stackrel{\Delta}{=} & \chi[[w]]\delta_0 \\
 \mathcal{D}_0 & \xrightarrow{Lkp_0[[w]]} & \mathbf{B} \\
 \uparrow \mathfrak{R}_{10} & & \uparrow \mathcal{I} \\
 \mathcal{D}_1 & \xrightarrow{Lkp_1[[w]]} & \mathbf{B} \\
 Lkp_1[[w]]\delta_1 & \stackrel{\Delta}{=} & \text{let } \delta_1 = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \text{ in } \chi[[w]]\delta_{h(w)}
 \end{array}$$

▷ Required to prove that

$$Lkp_0[[w]] \circ \mathfrak{R}_{10} = \mathcal{I} \circ Lkp_1[[w]]$$

▷ *Proof:*

▷ LHS

$$\begin{aligned}
 & (Lkp_0[[w]] \circ \mathfrak{R}_{10})\delta_1 \\
 &= Lkp_0[[w]](\mathfrak{R}_{10}(\delta_1)) \\
 &= \chi[[w]]\mathfrak{R}_{10}(\delta_1) \\
 &= \chi[[w]]\mathfrak{R}_{10}(\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r), \text{ substituting } \delta_1 = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \\
 &= \chi[[w]]^{\Delta} / (\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r) \\
 &= \chi[[w]](\Delta / \delta_l \Delta \Delta / \langle \delta_{h(w)} \rangle \Delta \Delta / \delta_r) \\
 &= \chi[[w]](\Delta / \delta_l \Delta \delta_{h(w)} \Delta \Delta / \delta_r) \\
 &= \chi[[w]]^{\Delta} / \delta_l \vee \chi[[w]]\delta_{h(w)} \vee \chi[[w]]^{\Delta} / \delta_r, \chi[[w]] \text{ is a homomorphism} \\
 &= \chi[[w]]\delta_{h(w)}
 \end{aligned}$$

▷ RHS

$$\begin{aligned}
 & (\mathcal{I} \circ Lkp_1[[w]])\delta_1 \\
 &= Lkp_1[[w]]\delta_1 \\
 &= Lkp_1[[w]](\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r) \\
 &= Lkp_1[[w]]\delta_{h(w)} \\
 &= \chi[[w]]\delta_{h(w)} \quad \text{Q.E.D.}
 \end{aligned}$$

# THE DICTIONARY

## 4.6. The Size Operation

$$\begin{array}{ccc}
 & & \text{Size}_0(\delta_0) \triangleq \text{card } \delta_0 \\
 & & \\
 \mathcal{D}_0 & \xrightarrow{\text{Size}_0} & \mathbf{N} \\
 \uparrow \mathfrak{R}_{10} & & \uparrow \mathcal{I} \\
 \mathcal{D}_1 & \xrightarrow{\text{Size}_1} & \mathbf{N} \\
 \text{Size}_1(\delta_1) \triangleq +/\text{card}^* \delta_1 & & 
 \end{array}$$

▷ Required to prove that

$$\text{Size}_0 \circ \mathfrak{R}_{10} = \mathcal{I} \circ \text{Size}_1$$

▷ *Proof:*

▷ LHS

$$\begin{aligned}
 & (\text{Size}_0 \circ \mathfrak{R}_{10})\delta_1 \\
 & = \text{Size}_0(\mathfrak{R}_{10}(\delta_1)) \\
 & = \text{Size}_0(\triangle/\delta_1) \\
 & = \text{card} \circ \triangle/\delta_1
 \end{aligned}$$

▷ RHS

$$\begin{aligned}
 & (\mathcal{I} \circ \text{Size}_1)\delta_1 \\
 & = \text{Size}_1\delta_1 \\
 & = +/\circ \text{card}^*\delta_1
 \end{aligned}$$

and the result is established since

$$\text{card} \circ \triangle/ = +/\circ \text{card}^*$$

## 5. Model 2 — Sequence

Having already specified a dictionary as a set of words earlier, it is interesting to see how one might use the sequence for the same purpose. In effect, this reification illustrates the *naïve* use of sequences to implement sets. I emphasise the qualifier ‘naïve’ to signal that there are many other more sophisticated ways in which one might implement sets on a computer, whose memory is, of course, modelled faithfully by the sequence type. A dictionary is considered to be an ordered sequence of words:

$$DICT_2 = WORD^*$$

Naturally, as it stands, the domain equation specifies more than required. In particular, a dictionary such as  $\langle w, w, w, w, w \rangle$ , which has five occurrences of the word  $w$ , is permissible. This is not the desired result. For this example, one invariant is that a sequence shall contain no duplicates. Let us call this the *uniqueness invariant*. Another problem with this model arises due to the inherent ordering of sequences. Thus,  $\langle a, b, c \rangle$ ,  $\langle b, a, c \rangle$ , and  $\langle c, a, b \rangle$  are all distinct realizations of  $\{a, b, c\}$ . To overcome this particular difficulty we must use the notion of equivalence class.

### 5.1. The Invariant

To eliminate the possibility of duplicates, we might use the following uniqueness invariant which is based on the notion of membership:

$$\begin{aligned} \text{inv-}DICT_2: DICT_2 &\longrightarrow \mathbf{B} \\ \text{inv-}DICT_2(\Lambda) &\triangleq true \\ \text{inv-}DICT_2(\langle e \rangle \wedge \tau) &\triangleq \neg \chi[[e]]\tau \wedge \text{inv-}DICT_2(\tau) \end{aligned}$$

Alternatively, we might consider a uniqueness invariant based on the notion of size of the dictionary. If the dictionary contains no duplicates then its length must be the same as the number of words it contains:

$$\text{inv-}DICT_2(\delta) \triangleq |elems \delta| = |\delta|$$

where I have used  $|-|$  in place of *card* and *len*, respectively. However, given such an invariant, the key conceptual idea is that operations may only be specified correctly if they preserve the invariant. In other words, given some dictionary  $\delta$  and a possibly parameterised operation  $op$ , then assuming  $\text{inv-}DICT_2(\delta) = true$ , it must be the

## THE DICTIONARY

case that  $inv-DICT_2(op\ \delta) = true$ . That a formal proof of invariant conservation is readily feasible is an important concern for the application of the VDM.

As noted above, even with this invariant, the ‘implementation’ contains more than the original abstraction. The solution is to provide an appropriate equivalence relation and to specify the corresponding equivalence classes. One solution is to say that two dictionaries (as sequences) are equivalent if and only if one is a permutation of the other. But this requires us to speak about pairs of dictionaries and our operations will operate on only one dictionary at a time. A different approach is to use the fact that every function  $f: X \rightarrow Y$  gives an equivalence relation on its domain (Mac Lane and Birkhoff 1979, 33). In the case of the dictionary as sequence,  $len$  gives the equivalence classes based on size. Thus, all dictionaries of size  $n$ ,  $n \in \mathbf{N}$  may be represented by the equivalence class  $[\delta]_n$ . The function  $elems$  will also map dictionaries into an equivalence class determined by the set of elements in the sequence and this is just what we need. A little thought shows that this is also the retrieve function that we require for this model.

### 5.2. The Retrieve Function

The retrieve function, denoted  $\mathfrak{R}_{20}$ , is that which maps sequences into sets. We already know from Chapter 2 that the  $elems$  operator on sequences is the natural choice:

$$\begin{aligned} \mathfrak{R}_{20}: DICT_2 &\longrightarrow DICT_0 \\ \mathfrak{R}_{20}(\delta) &\triangleq elems\ \delta \end{aligned}$$

Just as in the case of  $DICT_1$ , I will now demonstrate the relation between the retrieve function and the second form of the uniqueness invariant. A simple substitution gives the invariant

$$|\mathfrak{R}_{20}(\delta)| = |\delta|$$

The expression on the right hand side is the size of the dictionary. Therefore, a further substitution gives

$$|\mathfrak{R}_{20}(\delta)| = Size_2(\delta)$$

Finally, noting that the left hand side is basically the size of the dictionary, considered as a set, gives the result

$$Size_0(\mathfrak{R}_{20}(\delta)) = Size_2(\delta)$$

which is exactly the same form as we obtained for  $DICT_1$ .

### 5.3. The Operations

The operations for this model are exactly the same as those for  $DICT_0$ . However, I take the opportunity here to exhibit my method of ensuring that each operation satisfies the invariant. In addition, I will also make remarks on the use of sequence to implement set.

#### 5.3.1. The New Command

The empty sequence corresponds to the empty set. Therefore, the specification of  $New_2$  is simply

$$\begin{aligned} New_2: & \longrightarrow DICT_2 \\ New_2 & \triangleq \Lambda \end{aligned}$$

It trivially satisfies the first form of the invariant. For the second form we have:

$$\begin{aligned} |elems \circ New_2| &= |elems \Lambda| = |\emptyset| = 0 \\ |New_2| &= |\Lambda| = 0 \end{aligned}$$

Satisfaction of the invariant is simply a statement about the identity element of the respective monoids.

#### 5.3.2. The Enter Command

The specification of the enter command is the most problematic of all. Since a sequence is ordered we must say where a given word is to be entered. In the following specification I have chosen to place it at the head of the sequence.

$$\begin{aligned} Ent_2: & WORD \longrightarrow (DICT_2 \longrightarrow DICT_2) \\ Ent_2[[w]]\delta & \triangleq \langle w \rangle \wedge \delta \end{aligned}$$

and I must specify the constraint that the word does not already occur in the dictionary in order to satisfy the uniqueness constraint:

$$\begin{aligned} pre-Ent_2: & WORD \longrightarrow (DICT_2 \longrightarrow \mathbf{B}) \\ pre-Ent_2[[w]]\delta & \triangleq \neg\chi[[w]]\delta \end{aligned}$$

Conservation of the invariant may readily be established:

$$\begin{aligned} inv-DICT_2(Ent_2[[w]]\delta) &= inv-DICT_2(\langle w \rangle \wedge \delta) \\ &= \neg\chi_\delta(w) \wedge inv-DICT_2(\delta) \\ &= \neg\chi_\delta(w) \end{aligned}$$

## THE DICTIONARY

which is exactly the precondition! Note that I have chosen to write  $\chi_\delta(w)$  in place of  $\chi[[w]]\delta$ . Let us now consider the operation with respect to the second form of the uniqueness invariant. First, we have

HYPOTHESIS A.1.  $|elems \delta| = |\delta|$ .

Then a simple computation gives

$$\begin{aligned} |elems \circ Ent_2[[w]]\delta| &= |elems(\langle w \rangle \wedge \delta)| \\ &= |\{w\} \cup elems \delta| \\ &= 1 + |elems \delta|, \text{ because of the precondition} \\ |Ent_2[[w]]\delta| &= |\langle w \rangle \wedge \delta| \\ &= 1 + |\delta|, \text{ because } len \text{ is a homomorphism} \end{aligned}$$

Again it is worth reiterating that it was in the effort to build an operator calculus, in order to carry out such computations, that much of the theory of monoids and semigroups was found to be essential for the VDM. Now let us consider the problem of using sequences to implement sets.

The specification of the enter command is intended to model set union. Let  $w$  and  $w'$  be two distinct words not in the dictionary  $\delta$ . Then it is simple to show that

$$Ent_2[[w']] \circ Ent_2[[w]]\delta \neq Ent_2[[w]] \circ Ent_2[[w']]\delta$$

Thus,  $Ent_2$  is non-commutative, whereas set union is commutative. If we are really interested in implementing sets using sequences and if we naïvely suppose that this ought to be done using the above specification then we understand little about formal specifications. To avoid such misunderstanding a non-deterministic form of specification may be used:

$$\begin{aligned} Ent_2[[w]]\delta &\triangleq \\ \delta &= \Lambda \\ &\rightarrow \langle w \rangle \\ &\rightarrow \text{let } \delta = \delta_l \wedge \delta_r \text{ in } \delta_l \wedge \langle w \rangle \wedge \delta_r \end{aligned}$$

Now we do not know where in a given sequence a particular word will be placed. We can not say for certain whether  $Ent_2$  is commutative or not.

### 5.3.3. The Remove Command

Removal of a word from a dictionary is specified as

$$\begin{aligned} \text{Rem}_2: \text{WORD} &\longrightarrow (\text{DICT}_2 \longrightarrow \text{DICT}_2) \\ \text{Rem}_2[[w]]\delta &\triangleq \{w\} \Leftarrow \delta \end{aligned}$$

where I have used the operator symbol  $\Leftarrow$  to denote the ‘algorithm’ to remove all occurrences of an element from a sequence. I say ‘all’ rather than ‘the first’ even though there is only one if the invariant is maintained. The reason is simple. The algorithm to remove all occurrences is a sequence endomorphism. For an arbitrary sequence, the algorithm for removal of the first occurrence of an element is not a homomorphism. Let us now check the invariant:

HYPOTHESIS A.2.  $|\text{elems } \delta| = |\delta|$ .

Again, a simple computation gives

$$\begin{aligned} |\text{elems} \circ \text{Rem}_2[[w]]\delta| &= |\text{elems}(\{w\} \Leftarrow \delta)| \\ &= \begin{cases} |\text{elems } \delta| - 1, & \text{if } \chi[[w]]\delta \\ |\text{elems } \delta|, & \text{otherwise} \end{cases} \\ |\text{Rem}_2[[w]]\delta| &= |\{w\} \Leftarrow \delta| \\ &= \begin{cases} |\delta| - 1, & \text{if } \chi[[w]]\delta \\ |\delta|, & \text{otherwise} \end{cases} \end{aligned}$$

Note that I have chosen not to give a pre-condition.

### 5.3.4. The Lookup Command

To lookup a word in the dictionary I use the overloaded form of the characteristic function:

$$\begin{aligned} \text{Lkp}_2: \text{WORD} &\longrightarrow (\text{DICT}_2 \longrightarrow \mathbf{B}) \\ \text{Lkp}_2[[w]]\delta &\triangleq \chi[[w]]\delta \end{aligned}$$

It may be justified by noting that

$$\begin{aligned} \text{Lkp}_2[[w]]\delta &= \chi[[w]]\delta \\ &= \chi[[w]] \text{elems } \delta \\ &= \chi[[w]] \mathfrak{R}_{20}(\delta) \\ &= \text{Lkp}_0[[w]] \mathfrak{R}_{20}(\delta) \end{aligned}$$

which is just the proof that  $\text{Lkp}_2$  is the reification of  $\text{Lkp}_0$ .

## THE DICTIONARY

### 5.3.5. The Size Command

The specification of the size of the dictionary is straightforward:

$$\begin{aligned} \text{Size}_2: \text{DICT}_2 &\longrightarrow \mathbf{N} \\ \text{Size}_2(\delta) &\triangleq |\delta| \end{aligned}$$

I have already demonstrated the rôle of  $\text{Size}_2$  with respect to the invariant.

### 5.4. Further Developments

Much of the specification of  $\text{DICT}_2$  has been about exploring the properties of the  $\Sigma^*$ -homomorphisms  $\text{elems}$  and  $\text{len}$ . The next stage in the method is to consider possible generalisations such as we have done in the other sections. One such generalisation is to consider the  $\Sigma^*$ -homomorphism  $\text{items}$  which maps sequences into bags or multisets. Specifically, we might consider the equivalent retrieve function

$$\mathfrak{R}_{20}(\delta) = \text{dom} \circ \text{items } \delta$$

and the size operation defined by

$$\text{Size}_2(\delta) = |\text{items } \delta|$$

where  $|-|$  denotes the size of a bag. Carrying out the proofs again with respect to these forms helps to establish other facts which will prove useful in different specifications.

Alternatively, the definitions of the  $\Sigma^*$ -homomorphisms  $\text{elems}$  and  $\text{len}$  may be replaced with their reduction forms  $^{\cup}/F^*$  and  $^+/G^*$ , where  $F: w \mapsto \{w\}$  and  $G: w \mapsto 1$ , respectively. Carrying out the proofs yet once again helps establish both the correctness of the results and serves to exercise the use of these operators. For example, checking the invariant in the case of the  $\text{Ent}_2$  command (non-deterministic case) takes the form

HYPOTHESIS A.3.  $|\cup/F^*\delta| = |+/G^*\delta|$ .

## Formal Relationship of Model 2 to Model 0

*Proof:*

$$\begin{aligned}
 |\cup/F^*Ent_2[[w]]\delta| &= |\cup/F^*(\delta_l \wedge \langle w \rangle \wedge \delta_r)| \\
 &= |\cup/(F^*\delta_l \wedge F^*\langle w \rangle \wedge F^*\delta_r)| \\
 &= |\cup/F^*\delta_l \cup \cup/F^*\langle w \rangle \cup \cup/F^*\delta_r| \\
 &= |\cup/F^*\delta_l \cup \cup/\langle F(w) \rangle \cup \cup/F^*\delta_r| \\
 &= |\cup/F^*\delta_l \cup F(w) \cup \cup/F^*\delta_r| \\
 &= |\cup/F^*\delta_l \cup \{w\} \cup \cup/F^*\delta_r| \\
 &= |\cup/F^*\delta_l \cup \cup/F^*\delta_r| + 1 \\
 &= |\cup/F^*(\delta_l \wedge \delta_r)| + 1
 \end{aligned}$$

and similarly for the right hand side.

## 6. Formal Relationship of Model 2 to Model 0

In this section I have chosen to replace *WORD* with  $X$  and the names  $DICT_0$  and  $DICT_2$  with their respective structures. In other words, the formal relationship between the two models is exhibited in a slightly more abstract form than usual. From a conceptual model point of view I am proposing that we consider the relationship of ‘set implemented as sequence’ to ‘set’, independently of any application domain.

### 6.1. The Retrieve Function

Recall that from the previous section, the ‘usual’ retrieve function from sequences to sets is simply the *elems* operator on sequences:

$$\begin{aligned}
 \mathfrak{R}_{20}: X^* &\longrightarrow \mathcal{P}X \\
 \mathfrak{R}_{20}(\delta) &\triangleq elems \delta
 \end{aligned}$$

Alternatively, as I have already indicated in the previous section, one may construct the bag of items in the sequence and then take the domain of that bag.

$$\begin{aligned}
 \mathfrak{R}_{20}: X^* &\longrightarrow \mathcal{P}X \\
 \mathfrak{R}_{20}(\delta) &\triangleq dom \circ items \delta
 \end{aligned}$$

# THE DICTIONARY

It is this latter form of the retrieve function that I have chosen to employ in the proofs.

## 6.2. The New Operation

$$\begin{array}{ccc}
 & & New_0 \triangleq \emptyset \\
 & & \downarrow \\
 & & \xrightarrow{New_0} PX \\
 & & \uparrow \mathfrak{R}_{20} \\
 & & \xrightarrow{New_2} X^* \\
 & & \downarrow \\
 & & New_2 \triangleq \Lambda
 \end{array}$$

▷ Required to show that

$$(\mathfrak{R}_{20} \circ New_2) = New_0$$

▷ Proof

$$\begin{aligned}
 \mathfrak{R}_{20} \circ New_2 & \\
 &= \mathfrak{R}_{20}(\Lambda) \\
 &= dom \circ items \Lambda \\
 &= dom \theta \\
 &= \emptyset \quad \text{Q.E.D.}
 \end{aligned}$$

## 6.3. The Enter Operation

$$\begin{array}{ccc}
 Ent_0[[w]]\delta_0 \triangleq \{w\} \cup \delta_0 & & \\
 & & \downarrow \\
 \mathcal{P}X & \xrightarrow{Ent_0[[w]]} & \mathcal{P}X \\
 \uparrow \mathfrak{R}_{20} & & \uparrow \mathfrak{R}_{20} \\
 X^* & \xrightarrow{Ent_2[[w]]} & X^* \\
 Ent_2[[w]]\delta_2 \triangleq \langle w \rangle \wedge \delta_2 & &
 \end{array}$$

▷ Required to show that

$$(Ent_0[[w]] \circ \mathfrak{R}_{20}) = (\mathfrak{R}_{20} \circ Ent_2[[w]])$$

▷ LHS

$$\begin{aligned} & (Ent_0[[w]] \circ \mathfrak{R}_{20})\delta \\ &= Ent_0[[w]](\mathfrak{R}_{20}(\delta)) \\ &= \{w\} \cup \mathfrak{R}_{20}(\delta) \end{aligned}$$

▷ RHS

$$\begin{aligned} & (\mathfrak{R}_{20} \circ Ent_2[[w]])\delta \\ &= \mathfrak{R}_{20}(Ent_2[[w]]\delta) \\ &= \mathfrak{R}_{20}(\langle w \rangle \wedge \delta) \\ &= dom \circ items(\langle w \rangle \wedge \delta) \\ &= dom([w \mapsto 1] \oplus items \delta) \quad \text{-- items is a homomorphism} \\ &= dom([w \mapsto 1]) \cup dom(items \delta) \quad \text{-- dom is a homomorphism} \\ &= \{w\} \cup dom \circ items \delta \\ &= \{w\} \cup \mathfrak{R}_{20}(\delta) \quad \text{Q.E.D.} \end{aligned}$$

#### 6.4. The Remove Operation

$$\begin{array}{ccc} & & Rem_0[[w]]\delta_0 \triangleq \{w\} \triangleleft \delta_0 \\ & & \\ \mathcal{P}X & \xrightarrow{Rem_0[[w]]} & \mathcal{P}X \\ \uparrow \mathfrak{R}_{20} & & \uparrow \mathfrak{R}_{20} \\ X^* & \xrightarrow{Rem_2[[w]]} & X^* \\ & & Rem_2[[w]]\delta_2 \triangleq \{w\} \triangleleft \delta_2 \end{array}$$

▷ Required to show that

$$(Rem_0[[w]] \circ \mathfrak{R}_{20}) = (\mathfrak{R}_{20} \circ Rem_2[[w]])$$

▷ LHS

$$\begin{aligned} & (Rem_0[[w]] \circ \mathfrak{R}_{20})\delta_0 \\ &= Rem_0[[w]](\mathfrak{R}_{20}(\delta_0)) \\ &= \{w\} \triangleleft \mathfrak{R}_{20}(\delta_0) \end{aligned}$$

# THE DICTIONARY

▷ RHS

$$\begin{aligned}
 & (\mathfrak{R}_{20} \circ Rem_2[w]) \\
 &= \mathfrak{R}_{20}(Rem_2[w]\delta) \\
 &= \mathfrak{R}_{20}(\{w\} \leftarrow \delta) \\
 &= dom \circ items(\{w\} \leftarrow \delta) \\
 &= dom(items \delta \ominus [w \mapsto 1]) \\
 &= dom \circ items \delta - dom([w \mapsto 1]) \\
 &= \{w\} \leftarrow dom \circ items \delta \\
 &= \{w\} \leftarrow \mathfrak{R}_{20}(\delta) \quad \text{Q.E.D.}
 \end{aligned}$$

## 6.5. The Lookup Operation

$$\begin{array}{ccc}
 Lkp_0[w]\delta_0 \triangleq \chi[w]\delta_0 & & \\
 \mathcal{P}X \xrightarrow{Lkp_0[w]} \mathbf{B} & & \\
 \uparrow \mathfrak{R}_{20} & & \uparrow \mathcal{I} \\
 X^* \xrightarrow{Lkp_2[w]} \mathbf{B} & & \\
 Lkp_2[w]\delta_2 \triangleq \chi[w]\delta_2 & & 
 \end{array}$$

▷ Required to show that

$$(Lkp_0[w] \circ \mathfrak{R}_{20}) = (\mathcal{I} \circ Lkp_2[w])$$

▷ LHS

$$\begin{aligned}
 & (Lkp_0[w] \circ \mathfrak{R}_{20})\delta \\
 &= Lkp_0[w](\mathfrak{R}_{20}(\delta)) \\
 &= \chi[w]\mathfrak{R}_{20}(\delta) \\
 &= \chi[w] dom \circ items(\delta)
 \end{aligned}$$

▷ RHS

$$\begin{aligned}
 & (\mathcal{I} \circ Lkp_2[w])\delta \\
 &= Lkp_2[w]\delta \\
 &= \chi[w]\delta \\
 &= \chi[w] dom \circ items \delta \quad \text{-- by definition. Q.E.D.}
 \end{aligned}$$

6.6. The Size Operation

$$\begin{array}{ccc}
 & & Size_0(\delta_0) \triangleq card \delta_0 \\
 & & \\
 \mathcal{P}X & \xrightarrow{Size_0} & \mathbf{N} \\
 \uparrow \mathfrak{R}_{20} & & \uparrow \mathcal{I} \\
 X^* & \xrightarrow{Size_2} & \mathbf{N} \\
 & & Size_2(\delta_2) \triangleq len \delta_2
 \end{array}$$

▷ Required to show that

$$(Size_0 \circ \mathfrak{R}_{20}) = (\mathcal{I} \circ Size_2)$$

▷ LHS

$$\begin{aligned}
 & (Size_0 \circ \mathfrak{R}_{20})\delta \\
 &= Size_0(\mathfrak{R}_{20}(\delta)) \\
 &= card \mathfrak{R}_{20}(\delta) \\
 &= card \circ dom \circ items \delta
 \end{aligned}$$

▷ RHS

$$\begin{aligned}
 & (\mathcal{I} \circ Size_2)\delta \\
 &= Size_2(\delta) \\
 &= len \delta \quad \text{Q.E.D.}
 \end{aligned}$$

## 7. Model 3 — Sequence Partition

Let us now consider a reification of  $DICT_2$  by partitioning the sequence. The conceptual model that I have in mind here is a hash table. The appropriate domain equations are

$$DICT_3 = PAGE^*$$

$$PAGE = WORD^*$$

where I have decided to use the name  $PAGE$  rather than  $PART$  as in the case of  $DICT_1$ . Abstractly, we are dealing with the model

$$MODEL_3 = (X^*)^*$$

Indeed, as will be made clear later this model is a reification not only of  $DICT_2$  but also of  $DICT_1$  and  $DICT_0$ . Since the abstract domain denotes all possible structures of sequences of sequences, then we need to be more specific about the domain of discourse. In other words we need an invariant.

### 7.1. The Invariant

Keeping in mind that we wish to consider a hash table type of structure, then I must be quite specific about the type of hash table, there being many different kinds (Knuth 1973, 3). I choose to consider that which uses the strategy of ‘overflow chaining’ in the case of collisions where the overflow is a sequence. In addition, the hash function  $h$ , which is not further specified, has the property that it computes a number in the range  $\{0 \dots p - 1\}$  where  $p$  is prime. The invariant is, therefore

$$inv-DICT_3: DICT_3 \longrightarrow \mathbf{B}$$

$$inv-DICT_3(\delta) \triangleq len \delta = p \wedge is\text{-}prime(p) \wedge p > 100$$

$$\wedge inv-DICT_2(\delta) \wedge \wedge / \circ inv-DICT_2^*(\delta)$$

Note that I have taken advantage of the uniqueness invariant of  $DICT_2$  to assert that there are no duplicate pages and there are no duplicate words on any page. I do not say how big the hash table ought to be, merely that it should contain at least 100 ‘pages’. Even to say this much is probably too much.

## 7.2. The Retrieve Functions

I now take the opportunity to show that a very rich computational field of elementary mathematics awaits to be discovered in formal specifications. Specifically, I posit the existence of retrieve functions from the current model to each of the other models that have already been discussed. Moreover, said retrieve functions ought to be expressible as compositions of other retrieve functions. In the method of the Irish School it is imperative to explore different avenues of development. A simple proof that one model is a reification of another is insufficient. There may be more than one way to establish correctness and one seeks that which is aesthetically satisfying and appeals to the intuition in the sense of Kant. The conjecture that such composable retrieve functions exist will be tested in the subsequent section.

First, the retrieve function that maps the current model into the simple model using sequences is obtained by ‘flattening’—a reduction homomorphism:

$$\begin{aligned} \mathfrak{R}_{32}: DICT_3 &\longrightarrow DICT_2 \\ \mathfrak{R}_{32}(\delta) &\triangleq \wedge / \delta \end{aligned}$$

From here I can get back to the original model of sets using the *elems* homomorphism. Thus, we have the retrieve function:

$$\begin{aligned} \mathfrak{R}_{30}: DICT_3 &\longrightarrow DICT_0 \\ \mathfrak{R}_{30}(\delta) &\triangleq (\mathfrak{R}_{32} \circ \mathfrak{R}_{20})\delta = elems \circ \wedge / \delta \end{aligned}$$

Alternatively, it is clear that  $DICT_3$  is also a reification of  $DICT_1$  and, consequently, one has the retrieve homomorphism  $elems^*$  giving

$$\begin{aligned} \mathfrak{R}_{31}: DICT_3 &\longrightarrow DICT_1 \\ \mathfrak{R}_{31}(\delta) &\triangleq elems^* \delta \end{aligned}$$

and from here we can also get back to the original abstract model:

$$\begin{aligned} \mathfrak{R}_{30}: DICT_3 &\longrightarrow DICT_0 \\ \mathfrak{R}_{30}(\delta) &\triangleq (\mathfrak{R}_{31} \circ \mathfrak{R}_{10})\delta = \Delta / \circ elems^* \delta \end{aligned}$$

# THE DICTIONARY

These results may be summarised by the commuting diagram

$$\begin{array}{ccc}
 & \mathcal{P}X & \\
 \Delta/ & & elems \\
 & \Delta/ \circ elems^* & \\
 (\mathcal{P}X)^* & & X^* \\
 & elems \circ \wedge/ & \\
 elems^* & & \wedge/ \\
 & (X^*)^* &
 \end{array}$$

Thus from just very simple specifications, it is clear that there is an enormous wealth of interesting algebra to be found, such algebra being indispensable for the operator calculus of the Irish School of the *VDM* and, in my opinion being a much more interesting approach than those which rely on predicate calculus to do proofs.

## 7.3. The Operations

The operations to be specified are similar to those of the *DICT*<sub>1</sub> model. Indeed, there is so much similarity between the two specifications that I do not comment very much on the specification given here. This is particularly the case in view of the retrieve functions elaborated above.

### 7.3.1. The New Command

$$\begin{aligned}
 New_3: \mathbf{N} &\longrightarrow DICT_3 \\
 New_3[[p]] &\triangleq \langle \tau_k \mid \tau_k = \Lambda \wedge 0 \leq k \leq p-1 \rangle
 \end{aligned}$$

### 7.3.2. The Enter Command

$$\begin{aligned}
 Ent_3: WORD &\longrightarrow (DICT_3 \longrightarrow DICT_3) \\
 Ent_3[[w]]\delta &\triangleq \text{let } \delta = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \\
 &\quad \text{in } \delta_l \wedge \langle \langle w \rangle \wedge \delta_{h(w)} \rangle \wedge \delta_r
 \end{aligned}$$

subject to the pre-condition

$$\begin{aligned}
 pre-Ent_3: WORD &\longrightarrow (DICT_3 \longrightarrow \mathbf{B}) \\
 pre-Ent_3[[w]]\delta &\triangleq \text{let } \delta = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \\
 &\quad \text{in } \neg\chi[[w]]\delta_{h(w)}
 \end{aligned}$$

Note that I do not care to be non-deterministic!

### 7.3.3. The Remove Command

$$\begin{aligned} Rem_3: WORD &\longrightarrow (DICT_3 \longrightarrow DICT_3) \\ Rem_3[[w]]\delta &\triangleq \text{let } \delta = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \\ &\quad \text{in } \delta_l \wedge \langle \{w\} \Leftarrow \delta_{h(w)} \rangle \wedge \delta_r \end{aligned}$$

### 7.3.4. The Lookup Command

$$\begin{aligned} Lkp_3: WORD &\longrightarrow (DICT_3 \longrightarrow DICT_3) \\ Lkp_3[[w]]\delta &\triangleq \text{let } d = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \\ &\quad \text{in } \chi[[w]]\delta_{h(w)} \end{aligned}$$

### 7.3.5. The Size Command

$$\begin{aligned} Size_3: DICT_3 &\longrightarrow \mathbf{N} \\ Size_3(\delta) &\triangleq +/len^*\delta \end{aligned}$$

## 8. Formal Relationship of Model 3 to Model 0

In the discussion of the model  $DICT_3$  in the previous section, I proposed a number of conjectures concerning its relationship with the three earlier models under retrieve functions. Here I would like to demonstrate the ‘proofs and refutations’ method of Lakatos (1976). First, for each of the operations, I show that under the retrieve function  $\mathfrak{R}_{30}: DICT_3 \longrightarrow DICT_0$ , where  $\mathfrak{R}_{30} = \cup / \circ elems^*$ , the model  $DICT_3$  is a reification of  $DICT_0$ .

### 8.1. The Retrieve Function

Let us first consider the ‘direct’ retrieve functions  $\mathfrak{R}_{30}$  which map  $DICT_3$  to  $DICT_0$ . For the purpose of proof these are abbreviated as follows:

$$\begin{aligned} \mathfrak{R}_1(\delta_3) &\triangleq \cup / \circ elems^*(\delta_3) \\ \mathfrak{R}_2(\delta_3) &\triangleq elems \circ \wedge / (\delta_3) \end{aligned}$$

# THE DICTIONARY

## 8.2. The New Operation

We are required to prove that  $New_3$  is a reification of  $New_0$  with respect to  $\mathfrak{R}_1 = \cup / \circ elems^*$ .

$$\begin{array}{ccc}
 New_0 \stackrel{\Delta}{=} \emptyset & & \\
 \xrightarrow{New_0} \mathcal{D}_0 & \xrightarrow{\quad} & \mathcal{D}_0 \\
 \xrightarrow{New_3[[p]]} \mathcal{D}_3 & \xrightarrow{\quad} & \uparrow \mathfrak{R}_1 \\
 New_3[[p]] \stackrel{\Delta}{=} \langle \tau_k \mid \tau_k = \Lambda \wedge 0 \leq k \leq p-1 \rangle & & 
 \end{array}$$

Formally, we are required to show that

$$New_0 = \mathfrak{R}_1 \circ New_3[[p]]$$

*Proof:* There is only the right-hand-side to consider:

$$\begin{aligned}
 & \mathfrak{R}_1 \circ New_3[[p]] \\
 &= \mathfrak{R}_1(New_3[[p]]) \\
 &= \mathfrak{R}_1(\langle \tau_k \mid \tau_k = \Lambda \wedge 0 \leq k \leq p-1 \rangle) \\
 &= \cup / \circ elems^*(\langle \tau_k \mid \tau_k = \Lambda \wedge 0 \leq k \leq p-1 \rangle) \\
 &= \cup / \langle elems(\tau_k) \mid \tau_k = \Lambda \wedge 0 \leq k \leq p-1 \rangle \\
 &= \cup / \langle s_k \mid s_k = \emptyset \wedge 0 \leq k \leq p-1 \rangle \\
 &\quad \text{which is } \mathfrak{R}_{10}(\mathfrak{R}_{31} \circ New_3[[p]]) = \mathfrak{R}_{10}(New_1[[p]]) \\
 &= \emptyset \\
 &= New_0 \quad \text{Q. E. D.}
 \end{aligned}$$

Note that we have also just shown that  $New_3$  is also a reification of  $New_1$  under  $\mathfrak{R}_{31} = elems^*$ , which is, strictly speaking, exact only in the case that the same number of pages/partitions  $p$  is chosen. Using  $\mathfrak{R}_2$ , we verify that  $\mathfrak{R}_2 \circ New_3 = New_0$ . The relevant details are

$$\begin{aligned}
 & \mathfrak{R}_2(\langle \tau_k \mid \tau_k = \Lambda \wedge 0 \leq k \leq p-1 \rangle) \\
 &= elems \circ \wedge / (\langle \tau_k \mid \tau_k = \Lambda \wedge 0 \leq k \leq p-1 \rangle) \\
 &= elems \Lambda \\
 &\quad \text{which is } \mathfrak{R}_{20}(\mathfrak{R}_{32} \circ New_3[[p]]) = \mathfrak{R}_{20}(New_2) \\
 &= \emptyset
 \end{aligned}$$

## 8.3. The Enter Operation

First we prove that  $Ent_3$  is a reification of  $New_0$  with respect to the retrieve function  $\mathfrak{R}_1 = \cup / \circ elems^*$ .

$$\begin{array}{ccc}
 Ent_0[[w]]\delta_3 \triangleq \delta_3 \cup \{w\} & & \\
 \mathcal{D}_0 & \xrightarrow{Ent_0[[w]]} & \mathcal{D}_0 \\
 \uparrow \mathfrak{R}_1 & & \uparrow \mathfrak{R}_1 \\
 \mathcal{D}_3 & \xrightarrow{Ent_3[[w]]} & \mathcal{D}_3 \\
 Ent_3[[w]](\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r) \triangleq \delta_l \wedge \langle \langle w \rangle \wedge \delta_{h(w)} \rangle \wedge \delta_r & & 
 \end{array}$$

Formally, we are required to show that

$$(Ent_0[[w]] \circ \mathfrak{R}_1)\delta_3 = (\mathfrak{R}_1 \circ Ent_3[[w]])\delta_3$$

*Proof:* Consideration of the left-hand-side immediately gives us

$$\begin{aligned}
 & (Ent_0[[w]] \circ \mathfrak{R}_1)\delta_3 \\
 &= Ent_0[[w]](\mathfrak{R}_1(\delta_3)) \\
 &= \{w\} \cup \mathfrak{R}_1(\delta_3)
 \end{aligned}$$

and now the right-hand-side:

$$\begin{aligned}
 & (\mathfrak{R}_1 \circ Ent_3[[w]])\delta_3 \\
 &= \mathfrak{R}_1(Ent_3[[w]]\delta_3) \\
 &= \mathfrak{R}_1(\text{let } \delta_3 = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \\
 &\quad \text{in } \delta_l \wedge \langle \langle w \rangle \wedge \delta_{h(w)} \rangle \wedge \delta_r) \\
 &= \text{let } \delta_3 = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \\
 &\quad \text{in } \mathfrak{R}_1(\delta_l \wedge \langle \langle w \rangle \wedge \delta_{h(w)} \rangle \wedge \delta_r)
 \end{aligned}$$

But

$$\begin{aligned}
 & \mathfrak{R}_1(\delta_l \wedge \langle \langle w \rangle \wedge \delta_{h(w)} \rangle \wedge \delta_r) \\
 &= \cup / \circ elems^*(\delta_l \wedge \langle \langle w \rangle \wedge \delta_{h(w)} \rangle \wedge \delta_r) \\
 &= \cup / (elems^* \delta_l \wedge elems^*(\langle \langle w \rangle \wedge \delta_{h(w)} \rangle) \wedge elems^* \delta_r) \\
 &= \cup / (elems^* \delta_l \wedge \langle elems(\langle w \rangle \wedge \delta_{h(w)}) \rangle \wedge elems^* \delta_r) \\
 &= \cup / (elems^* \delta_l \wedge \langle \{w\} \cup elems \delta_{h(w)} \rangle \wedge elems^* \delta_r) \\
 &\quad \text{which is } \mathfrak{R}_{10}(\mathfrak{R}_{31} \circ Ent_3[[w]]) = \mathfrak{R}_{10}(Ent_1[[w]] \circ \mathfrak{R}_{31}) \\
 &\quad \text{only if the partition function } h \text{ is the same.} \\
 &= \cup / \circ elems^* \delta_l \cup \cup / \langle \{w\} \cup elems \delta_{h(w)} \rangle \cup \cup / \circ elems^* \delta_r
 \end{aligned}$$

# THE DICTIONARY

$$\begin{aligned}
&= \cup / \circ elems^* \delta_l \cup \{w\} \cup elems \delta_{h(w)} \cup \cup / \circ elems^* \delta_r \\
&= \{w\} \cup (\cup / \circ elems^* \delta_l \cup elems \delta_{h(w)} \cup \cup / \circ elems^* \delta_r) \\
&= \{w\} \cup \cup / \circ elems^* (\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r) \\
&= \{w\} \cup \mathfrak{R}_1(\delta_3) \\
&\text{Q. E. D.}
\end{aligned}$$

Thus, we can assert that  $DICT_3$  is also a reification of  $DICT_1$  if the ‘magic’ function  $h$  used to assign words to a partition/page is the same. I use ‘same’ in a loose sense here. Strictly speaking, if we let  $h_1: WORD \longrightarrow \{1 \dots p\}$  and  $h_3: WORD \longrightarrow \{0 \dots p-1\}$  denote the two functions in question, subscripted appropriate to the model, then we require that  $plus[[1]] \circ h_3(w) = h_1(w)$ . But this gives us an opportunity to widen the conjecture according to the Lakatos’ method. First let us consider functions  $h_1$  and  $h_3$  which have the partition property, i.e.,  $h_1: WORD \longrightarrow \{1 \dots p\}$  and  $h_3: WORD \longrightarrow \{0 \dots p-1\}$ , such that  $plus[[1]] \circ h_3 = h_1$ , but not necessarily that for a given word  $w$ ,  $h_1$  and  $h_3$  map  $w$  into the ‘same’ partition, i.e.,  $plus[[1]] \circ h_3(w) \neq h_1(w)$ . Then what is the relationship between  $DICT_3$  and  $DICT_1$ ? Formally, we seek a function  $f$  such that  $f \circ h_3 = h_1$ . This may yet again be generalised to indexed families of functions  $\{h_1^i \mid i \in I\}$  and  $\{h_3^j \mid j \in J\}$ , where  $I$  and  $J$  are index sets, which have the partition property, but assign words to different partitions/pages. At the level of  $DICT_3$  one may interpret this as asking questions about the inter-relationships of dictionaries produced by different hashing functions. However, rather than explore solutions to these problems here, it has been sufficient to draw attention to the fact that by carrying out detailed proofs, interesting and important practical questions do arise.

Let us now turn to consider the problem of the retrieve function  $\mathfrak{R}_2 = elems \circ \wedge /$ . The relevant part of the proof is

$$\begin{aligned}
&\mathfrak{R}_2 \circ Ent_3[[w]]\delta_3 \\
&= \mathfrak{R}_2(\delta_l \wedge \langle \langle w \rangle \wedge \delta_{h(w)} \rangle \wedge \delta_r) \\
&= elems \circ \wedge / (\delta_l \wedge \langle \langle w \rangle \wedge \delta_{h(w)} \rangle \wedge \delta_r) \\
&= elems(\wedge / \delta_l \wedge \wedge / \langle \langle w \rangle \wedge \delta_{h(w)} \rangle \wedge \wedge / \delta_r) \\
&= elems(\wedge / \delta_l \wedge \langle w \rangle \wedge \delta_{h(w)} \wedge \wedge / \delta_r)
\end{aligned}$$

Clearly, we will obtain the desired result. But, I have terminated the proof at precisely that point where the argument to  $elems$  is  $\mathfrak{R}_{32} \circ Ent_3[[w]]\delta_3$ . Considering  $Ent_2[[w]]\delta_2 = \langle w \rangle \wedge \delta_2$ , it is clear that  $\mathfrak{R}_{32} \circ Ent_3[[w]] \neq Ent_2[[w]] \circ \mathfrak{R}_{32}$ . Hence,

$DICT_3$  is not a refinement of  $DICT_2$  under the retrieve function  $\mathfrak{R}_{32} = \wedge /$ . Where the retrieve function breaks down is clear: the problem is with the property that a sequence inherently has order. We can not even rescue the situation by positing the non-deterministic specification of  $Ent_2$  because the use of a hash function is deterministic. This does not mean that the approach is flawed. It merely demonstrates that there is a great deal of complexity involved in considering reification. What we have established here is that the composite,  $elems \circ \wedge /$ , is a valid retrieve function from  $DICT_3$  to  $DICT_0$  and although  $elems$  retrieves  $DICT_0$  from  $DICT_2$ , it is not true that  $\wedge /$  retrieves  $DICT_2$  from  $DICT_3$ . Looking ahead to the even more complex file system case study in Appendix C, where there are multiple levels of reification, one of our chief concerns will be examine carefully the compositionality of the given retrieve functions.

#### 8.4. The Remove Operation

Again we start by considering the proof that  $Re_1 = \cup / \circ elems^*$  retrieves  $DICT_0$  directly from  $DICT_3$ .

$$\begin{array}{ccc}
 & & \text{Rem}_0[[w]]\delta_3 \triangleq \{w\} \triangleleft \delta_3 \\
 & & \\
 \mathcal{D}_0 & \xrightarrow{\text{Rem}_0[[w]]} & \mathcal{D}_0 \\
 \uparrow \mathfrak{R}_1 & & \uparrow \mathfrak{R}_1 \\
 \mathcal{D}_3 & \xrightarrow{\text{Rem}_3[[w]]} & \mathcal{D}_3 \\
 \text{Rem}_3[[w]](\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r) \triangleq \delta_l \wedge \langle \{w\} \triangleleft \delta_{h(w)} \rangle \wedge \delta_r
 \end{array}$$

Formally, we are required to show that

$$(\text{Rem}_0[[w]] \circ \mathfrak{R}_1)\delta_3 = (\mathfrak{R}_1 \circ \text{Rem}_3[[w]])\delta_3$$

*Proof:* First we consider the left-hand-side:

$$\begin{aligned}
 & (\text{Rem}_0[[w]] \circ \mathfrak{R}_1)\delta_3 \\
 & = \text{Rem}_0[[w]](\mathfrak{R}_1(\delta_3)) \\
 & = \{w\} \triangleleft \mathfrak{R}_1(\delta_3)
 \end{aligned}$$

Now let us look at the right-hand-side

# THE DICTIONARY

$$\begin{aligned}
& (\mathfrak{R}_1 \circ \text{Rem}_3[[w]])\delta_3 \\
&= \mathfrak{R}_1(\text{Rem}_3[[w]]\delta_3) \\
&= \mathfrak{R}_1(\text{let } \delta_3 = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \\
&\quad \text{in } \delta_l \wedge \langle \{w\} \Leftarrow \delta_{h(w)} \rangle \wedge \delta_r) \\
&= \text{let } \delta_3 = \delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r \\
&\quad \text{in } \mathfrak{R}_1(\delta_l \wedge \langle \{w\} \Leftarrow \delta_{h(w)} \rangle \wedge \delta_r)
\end{aligned}$$

But

$$\begin{aligned}
& \mathfrak{R}_1(\delta_l \wedge \langle \{w\} \Leftarrow \delta_{h(w)} \rangle \wedge \delta_r) \\
&= \cup / \circ \text{elems}^*(\delta_l \wedge \langle \{w\} \rangle \wedge \delta_{h(w)} \rangle \wedge \delta_r) \\
&= \cup / (\text{elems}^* \delta_l \wedge \text{elems}^*(\langle \{w\} \Leftarrow \delta_{h(w)} \rangle) \wedge \text{elems}^* \delta_r) \\
&= \cup / (\text{elems}^* \delta_l \wedge \langle \text{elems}(\{w\} \Leftarrow \delta_{h(w)}) \rangle \wedge \text{elems}^* \delta_r) \\
&= \cup / (\text{elems}^* \delta_l \wedge \langle \{w\} \Leftarrow \text{elems } \delta_{h(w)} \rangle \wedge \text{elems}^* \delta_r) \\
&\quad \text{which is } \mathfrak{R}_{10}(\mathfrak{R}_{31} \circ \text{Rem}_3[[w]]) = \mathfrak{R}_{10}(\text{Rem}_1[[w]] \circ \mathfrak{R}_{31}) \\
&= \cup / \circ \text{elems}^* \delta_l \cup \cup / \langle \{w\} \Leftarrow \text{elems } \delta_{h(w)} \rangle \cup \cup / \circ \text{elems}^* \delta_r \\
&= \cup / \circ \text{elems}^* \delta_l \cup (\{w\} \Leftarrow \text{elems } \delta_{h(w)}) \cup \cup / \circ \text{elems}^* \delta_r \\
&= \{w\} \Leftarrow (\cup / \circ \text{elems}^* \delta_l \cup \text{elems } \delta_{h(w)} \cup \cup / \circ \text{elems}^* \delta_r) \\
&= \{w\} \Leftarrow \mathfrak{R}_1(\delta_3) \quad \text{Q. E. D.}
\end{aligned}$$

Thus passage from  $DICT_3$  through  $DICT_1$  to  $DICT_0$  is ‘safe’ with respect to removal. Let us now turn our attention to the other passage which failed in the case of the enter command. Specifically we wish to examine the conjecture that  $\mathfrak{R}_{32} \circ \text{Rem}_3[[w]] = \text{Rem}_2[[w]] \circ \mathfrak{R}_{32}$  by carrying out the proof that  $\mathfrak{R}_2 \circ \text{Rem}_3[[w]] = \text{Rem}_0[[w]] \circ \mathfrak{R}_2$ . The relevant stages of the proof are

$$\begin{aligned}
& \mathfrak{R}_2(\delta_l \wedge \langle \{w\} \Leftarrow \delta_{h(w)} \rangle \wedge \delta_r) \\
&= \text{elems} \circ \wedge / (\delta_l \wedge \langle \{w\} \Leftarrow \delta_{h(w)} \rangle \wedge \delta_r) \\
&= \text{elems}(\wedge / \delta_l \wedge \wedge / \langle \{w\} \Leftarrow \delta_{h(w)} \rangle \wedge \wedge / \delta_r) \\
&= \text{elems}(\wedge / \delta_l \wedge (\{w\} \Leftarrow \delta_{h(w)}) \wedge \wedge / \delta_r) \\
&= \text{elems}(\{w\} \Leftarrow (\wedge / \delta_l \wedge \delta_{h(w)} \wedge \wedge / \delta_r)) \\
&= \text{elems}(\{w\} \Leftarrow \mathfrak{R}_{32}(\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r)) \\
&= \text{elems}(\text{Rem}_2[[w]] \circ \mathfrak{R}_{32}(\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r))
\end{aligned}$$

and we can stop there. The passage is safe. Not only does  $\mathfrak{R}_2 = \text{elems} \circ \wedge /$  retrieve  $DICT_0$  from  $DICT_3$  but we can safely retrieve  $DICT_2$  first, using  $\mathfrak{R}_{32} = \wedge /$  and then retrieve  $DICT_0$ , using  $\mathfrak{R}_{20} = \text{elems}$ .

## 8.5. The Lookup Operation

Once again we first consider the proof of the direct route from  $DICT_3$  to  $DICT_0$  under the retrieve function  $\mathfrak{R}_1 = \cup / \circ elems^*$ :

$$\begin{array}{ccc}
 Lkp_0[[w]]\delta_0 & \stackrel{\Delta}{=} & \chi[[w]]\delta_0 \\
 \\
 \mathcal{D}_0 & \xrightarrow{Lkp_0[[w]]} & \mathbf{B} \\
 \uparrow \mathfrak{R}_1 & & \uparrow \mathcal{I} \\
 \mathcal{D}_3 & \xrightarrow{Lkp_3[[w]]} & \mathbf{B} \\
 Lkp_3[[w]](\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r) & \stackrel{\Delta}{=} & \chi[[w]]\delta_{h(w)}
 \end{array}$$

Formally, we are required to show that

$$(Lkp_0[[w]] \circ \mathfrak{R}_1)\delta_3 = (\mathcal{I} \circ Lkp_3[[w]])\delta_3$$

*Proof:* First we shall consider the left-hand-side. The characteristic function used in this part of the proof is strictly that for sets, i.e.,  $\chi[[w]]s \stackrel{\Delta}{=} w \in s$ :

$$\begin{aligned}
 & (Lkp_0[[w]] \circ \mathfrak{R}_1)\delta_3 \\
 &= Lkp_0[[w]](\mathfrak{R}_1(\delta_3)) \\
 &= Lkp_0[[w]](\cup / \circ elems^*(\delta_3)) \\
 &= \chi[[w]](\cup / \circ elems^*(\delta_3)) \\
 &= \chi[[w]](\cup / \circ elems^*(\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r)) \\
 &= \chi[[w]]\cup / (elems^* \delta_l \wedge elems^* \langle \delta_{h(w)} \rangle \wedge elems^* \delta_r) \\
 &= \chi[[w]]\cup / (elems^* \delta_l \wedge \langle elems \delta_{h(w)} \rangle \wedge elems^* \delta_r) \\
 &= \chi[[w]](\cup / \circ elems^* \delta_l \cup \cup / \langle elems \delta_{h(w)} \rangle \cup \cup / \circ elems^* \delta_r) \\
 &= \chi[[w]](\cup / \circ elems^* \delta_l \cup elems \delta_{h(w)} \cup \cup / \circ elems^* \delta_r) \\
 &= \chi[[w]]\cup / \circ elems^* \delta_l \vee \chi[[w]] elems \delta_{h(w)} \vee \chi[[w]]\cup / \circ elems^* \delta_r \\
 &= \chi[[w]] elems \delta_{h(w)}
 \end{aligned}$$

Now we consider the right-hand-side. In this case the characteristic function is that for sequences, i.e.,  $\chi[[w]]s \stackrel{\Delta}{=} w \in elems s$ :

$$\begin{aligned}
 & (\mathcal{I} \circ Lkp_3[[w]])\delta_3 \\
 &= Lkp_3[[w]]\delta_3 \\
 &= \chi[[w]](\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r) \\
 &= \chi[[w]]\delta_{h(w)} \\
 &= \chi[[w]] elems \delta_{h(w)} \quad \text{by definition}
 \end{aligned}$$

# THE DICTIONARY

Clearly, the result is established. But it is difficult to see the passage through  $DICT_1$  from the proof. Application of  $\mathfrak{R}_{31} = elems^*$  to  $\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r$ , gives  $elems^* \delta_l \wedge \langle elems \delta_{h(w)} \rangle \wedge elems^* \delta_r$ . Then  $Lkp_1[[w]]$  is an application of  $\chi[[w]]$  to this latter result to give  $\chi[[w]] elems \delta_{h(w)}$  directly. Similarly, it may be argued that the passage from  $DICT_3$  through  $DICT_2$  under  $\mathfrak{R}_2 = elems \circ \wedge /$  is also safe.

## 8.6. The Size Operation

In considering the proof using the retrieve function  $\mathfrak{R}_1 = \cup / \circ elems^*$ , it is important to note that the definition of  $Size_3$  used is of an analogous form:  $+ / len^*$ . In fact, for the retrieve function  $\mathfrak{R}_2 = elems \circ \wedge /$ , the analogous form for the definition of  $Size_3$  is  $len \circ \wedge /$ . In other words, the  $Size_3$  command is structurally isomorphic to the retrieve function.

$$\begin{array}{ccc}
 & & Size_0(\delta_0) \triangleq card \delta_0 \\
 & & \\
 \mathcal{D}_0 & \xrightarrow{Size_0} & \mathbf{N} \\
 \uparrow \mathfrak{R}_1 & & \uparrow \mathcal{I} \\
 \mathcal{D}_3 & \xrightarrow{Size_3} & \mathbf{N} \\
 Size_3(\delta_3) \triangleq + / \circ len^*(\delta_3) & & 
 \end{array}$$

Formally, we are required to prove that

$$(Size_0 \circ \mathfrak{R}_1)\delta_3 = (\mathcal{I} \circ Size_3)\delta_3$$

*Proof:* Considering the left-hand-side, the relevant details are

$$\begin{aligned}
 & (Size_0 \circ \mathfrak{R}_1)(\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r) \\
 &= card(\cup / \circ elems^*(\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r)) \\
 &= card(\cup / (elems^* \delta_l \wedge \langle elems \delta_{h(w)} \rangle \wedge elems^* \delta_r)) \\
 &= card(\cup / \circ elems^* \delta_l \cup elems \delta_{h(w)} \cup \cup / \circ elems^* \delta_r) \\
 &= |\cup / \circ elems^* \delta_l| + |elems \delta_{h(w)}| + |\cup / \circ elems^*| \\
 &\quad \text{because of the partitioning function } h
 \end{aligned}$$

and for the right-hand-side we have

$$\begin{aligned}
 (\mathcal{I} \circ \text{Size}_3)\delta_3 &= \text{Size}_3(\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r) \\
 &= +/\circ \text{len}^*(\delta_l \wedge \langle \delta_{h(w)} \rangle \wedge \delta_r) \\
 &= +/(\text{len}^*\delta_l \wedge \langle \text{len} \delta_{h(w)} \rangle \wedge \text{len}^*\delta_r) \\
 &= +/\circ \text{len}^*\delta_l + \text{len} \delta_{h(w)} + +/\circ \text{len}^*\delta_r
 \end{aligned}$$

and the result is established by structural induction. Taking the other passage through  $DICT_2$ , and using the corresponding definition of the size function, gives

$$|\text{elems} \circ \wedge / \delta_l| + |\text{elems} \delta_{h(w)}| + |\text{elems} \circ \wedge / \delta_r| = \text{len} \circ \wedge / \delta_l + \text{len} \delta_{h(w)} + \text{len} \circ \wedge / \delta_r$$

and we may conclude that the passage is safe.

That basically concludes my observations and analysis on this first collection of dictionary models. From the perspective of the practical application of formal specifications to real problem domains, I have demonstrated that great care needs to be taken in conducting proofs. The analysis which I presented with respect to the  $Ent_3$  command clearly highlights some of the difficulties that one ought to expect to find. That difficulties should arise in such simple models might at first seem disconcerting. However, it is important to keep in mind, that once the analysis has been carried out, the material then becomes a permanent repository of formal specification knowledge. We are establishing theorems and their proofs in much the same way as is done in mathematics.

9. Model 4 — Map

We now come to a breakpoint in our discussions on the various specifications of the dictionary. In this section I now look at the concept of *evolution* of a specification as distinct from reification. By evolution I mean that I now consider an *enrichment* of the original model  $DICT_0$  by incorporating more information at the same abstract level. Such evolution is a lateral development of the model. Fortunately, the technique of retrieve function application is still appropriate. The extra information I wish to consider is that of the definition of a word. Thus, I wish to consider the specification of a dictionary that may be used to lookup the definitions of words in contrast to the mere ‘spelling-checker’ dictionary that I have considered heretofore. The *usual* model of such a dictionary is

$$DICT = WORD \xrightarrow{m} \mathcal{P}DEF$$

which associates sets of definitions with each word. However, I consider this in the next section and look instead at an ‘artificial’ model of such a dictionary, artificial in the sense that only one definition is associated with each word:

$$DICT_4 = WORD \xrightarrow{m} DEF$$

There are two very good reasons for considering such an artificial model. First, I may use it as a contrast to the usual dictionary yet to be developed. Second, and perhaps more importantly, when I consider it abstractly

$$MODEL = X \xrightarrow{m} Y$$

I note that it is representative of many other specifications. For example, it is an abstract model of both the environment and the store of denotational semantics, to be discussed in Appendix B. It is also representative of part of the model of the file system in Appendix C. Before proceeding to the specification of the commands, I would like to introduce the retrieve function  $\mathfrak{R}_{40}$  which retrieves  $DICT_0$  from  $DICT_4$ :

$$\begin{aligned} \mathfrak{R}_{40}: DICT_4 &\longrightarrow DICT_0 \\ \mathfrak{R}_{40}(\delta) &\triangleq dom \delta \end{aligned}$$

Rather than dedicate a separate section to proofs that  $DICT_4$  is a valid evolution of  $DICT_0$ , I include them directly with the specification of the  $DICT_4$  operations.

## 9.1. The Operations

Just as in the case of the previous models, we have a command to create a new dictionary:  $New_4$ , a command to enter both words and their definitions into the dictionary:  $Ent_4$ , a remove command that deletes both a word and its definition from the dictionary:  $Rem_4$ , the all-important lookup command, the most significant one from the conceptual viewpoint of the user of the dictionary:  $Lkp_4$ , and the command which determines the size of the dictionary  $Size_4$ . In addition, I throw in an extra command which, given a definition as argument, returns the set of words that have the same definition:  $Assoc_4$ . This command has no counterpart, of course, in the  $DICT_0$  model. Each of these are now discussed in turn.

### 9.1.1. The New Command

Issuing of the new command gives me an empty dictionary just as  $New_0$  did in the case of  $DICT_0$ :

$$\begin{aligned} New_4 &: \longrightarrow DICT_4 \\ New_4 &\stackrel{\Delta}{=} \theta \end{aligned}$$

Clearly,  $\mathfrak{R}_{40} \circ New_4 = dom \theta = \emptyset = New_0$ .

### 9.1.2. The Enter Command

This command has the semantics that if a word  $w$  is not present in the dictionary, then the dictionary is extended with the new word together with its definition  $d_w$ ; otherwise, it is understood that the definition supplied  $d_w$ , is to replace an existing definition. Note in particular that there is no pre-condition:

$$\begin{aligned} Ent_4 &: WORD \times DEF \longrightarrow DICT_4 \longrightarrow DICT_4 \\ Ent_4[[w, d_w]]\delta &\stackrel{\Delta}{=} \\ &\neg\chi[[w]]\delta \\ &\quad \rightarrow \delta \cup [w \mapsto d_w] \\ &\quad \rightarrow \delta + [w \mapsto d_w] \end{aligned}$$

The proof that  $Ent_4$  is an evolution of  $Ent_0$  has two cases to consider. For convenience, I use a compact form of the McCarthy Conditional:

# THE DICTIONARY

$$\begin{aligned}
& (\mathfrak{R}_{40} \circ Ent_4[[w, d_w]])\delta \\
& = dom(\chi[[w]]\delta(\delta \cup [w \mapsto d_w], \delta + [w \mapsto d_w])) \\
& = \chi[[w]](dom(\delta \cup [w \mapsto d_w]), dom(\delta + [w \mapsto d_w])) \\
& = \chi[[w]](dom \delta \cup \{w\}, dom \delta) \\
& = dom \delta \cup \{w\} \\
& = (Ent_0[[w]] \circ \mathfrak{R}_{40})\delta
\end{aligned}$$

Note how *selection* by the characteristic function collapses due to the properties of set union: if  $w$  is already in the set  $dom \delta$ , then  $dom \delta = dom \delta \cup \{w\}$ .

A little diversion on the limitations of the enter command is in order. There is no mechanism in this model by which I am able to enter a word into the dictionary on its own. I must also give its definition. This is a little inflexible. One way in which I might cater for such an option, is to introduce the notion of a possible *nil* definition. This is accomplished by extending the original model to

$$DICT_4' = WORD \xrightarrow{m} (DEF \mid \{nil\})$$

### 9.1.3. The Remove Command

The command has no effect if the word supplied,  $w$ , is not in the dictionary:

$$\begin{aligned}
& Rem_4: WORD \longrightarrow DICT_4 \longrightarrow DICT_4 \\
& Rem_4[[w]]\delta \triangleq \\
& \quad \neg\chi[[w]]\delta \\
& \quad \rightarrow \delta \\
& \quad \rightarrow \{w\} \Leftarrow \delta
\end{aligned}$$

The proof that  $Rem_4$  is a valid evolution of  $Rem_0$  is similar to that of the enter command:

$$\begin{aligned}
& (\mathfrak{R}_{40} \circ Rem_4[[w]])\delta \\
& = dom(\neg\chi[[w]]\delta(\delta, \{w\} \Leftarrow \delta)) \\
& = \neg\chi[[w]]\delta(dom \delta, dom(\{w\} \Leftarrow \delta)) \\
& = \{w\} \Leftarrow dom \delta \\
& = (Rem_0[[w]] \circ \mathfrak{R}_{40})\delta
\end{aligned}$$

9.1.4. The Lookup Command

Upon issuing the lookup command, it returns the definition of a word if it occurs in the dictionary; otherwise I do not care to say what the operation does:

$$\begin{aligned}
 Lkp_4: WORD &\longrightarrow DICT_4 \longrightarrow DEF \\
 Lkp_4[[w]]\delta &\triangleq \\
 \chi[[w]]\delta & \\
 &\rightarrow \delta(w) \\
 &\rightarrow \perp
 \end{aligned}$$

Now let us apply the retrieve function in a ‘mechanical manner’, i.e., without thinking what the real meaning of the lookup operation is:

$$\begin{aligned}
 (\mathfrak{R}_{40} \circ Lkp_4[[w]])\delta & \\
 = dom(\chi[[w]]\delta(\delta(w), \perp)) & \\
 = \chi[[w]]\delta(dom \delta(w), dom \perp) &
 \end{aligned}$$

and we must stop here. Whereas, I can feel quite happy to interpret  $dom \perp$  as  $\perp$ , the expression  $dom \delta(w)$  is meaningless, at least in this model. Such a problem would have been detected immediately had we first attempted to draw a commuting diagram:

$$\begin{array}{ccc}
 \mathcal{D}_0 & \xrightarrow{Lkp_0[[w]]} & \mathbf{B} \\
 \uparrow \mathfrak{R}_{40} = dom & & \uparrow ? \\
 \mathcal{D}_4 & \xrightarrow{Lkp_4[[w]]} & DEF
 \end{array}$$

The required retrieve function which maps a definition into a boolean value can not possibly be  $\mathfrak{R}_{40}$ . But now we have another problem, indicated by the question mark: ‘?’. There is absolutely no possibility of retrieving something in  $DICT_0$  which corresponds to this lookup operation. In other words, this operation is really different in nature from all of the other operations in the model. It is meaningful only with respect to the evolved part of  $DICT_4$ .

# THE DICTIONARY

## 9.1.5. The Size Command

Finally, there is the command that determines the size of the dictionary. The size is determined by the number of words in the dictionary, not by the number of definitions:

$$\begin{aligned} \text{Size}_4: \text{DICT}_4 &\longrightarrow \mathbf{N} \\ \text{Size}_4(\delta_0) &\triangleq \text{card} \circ \text{dom } \delta \end{aligned}$$

The proof that the size command is valid is absolutely trivial:

$$(\text{Size}_0 \circ \mathfrak{R}_{40})\delta = \text{card} \circ \text{dom } \delta = \text{Size}_4(\delta)$$

## 9.1.6. The Associate Command

There may be occasions when one wishes to know which words have a given definition:

$$\begin{aligned} \text{Assoc}_4: \text{DEF} &\longrightarrow \text{DICT}_4 \longrightarrow \mathcal{P}\text{WORD} \\ \text{Assoc}_4[[d_w]]\delta &\triangleq \delta^{-1}(d_w) \end{aligned}$$

where  $\delta^{-1}(d_w) = \{w \mid \delta(w) = d_w\}$  is the inverse image of  $d_w$  under  $\delta$ . This operation has no counterpart whatsoever in the  $\text{DICT}_0$  model. Operations such as  $\text{Lkp}_4$  and  $\text{Assoc}_4$  effectively establish  $\text{DICT}_4$  as a new baseline model for subsequent reification.

## 10. Model 5 — Map

This is the *usual* model of a dictionary that associates definitions with words. It has already been discussed thoroughly with respect to a user's conceptual model in the Chapter on Communications and Behaviour. Formally, the dictionary is given by

$$DICT_5 = WORD \xrightarrow{m} \mathcal{P}DEF$$

In the abstract, the model is of the form

$$MODEL_5 = X \xrightarrow{m} \mathcal{P}Y$$

which is, of course, the model of a relation expressed as a map. Although I could give a retrieve function  $\mathfrak{R}_{50} = dom$  that maps  $DICT_5$  to  $DICT_0$  and carry out the corresponding proofs as for  $DICT_4$ , I do not do so, since the details are almost identical to those of the  $DICT_4$  in the previous section. This model, which is an evolution of  $DICT_0$ , is truly another new starting point for reification and I can now consider applying the sorts of operations to it as I did for  $DICT_4$ .

## 10.1. The Operations

The set of operations are very similar to those of  $DICT_4$ . However, due to the inherent flexibility and moderate complexity of this model, I introduce some new commands: the concept of an update command which is an extension of the enter command and the removal of definitions as well as that of words.

## 10.1.1. The New Command

The specification of the creation of a new dictionary is exactly the same as that for  $DICT_4$ :

$$\begin{aligned} New_5 &: \longrightarrow DICT_5 \\ New_5 &\triangleq \theta \end{aligned}$$

# THE DICTIONARY

## 10.1.2. The Enter Command

First, let us consider entering a new word together with a (possibly empty) set of definitions into the dictionary:

$$\begin{aligned} Ent_5: WORD \times \mathcal{P}DEF &\longrightarrow DICT_5 \longrightarrow DICT_5 \\ Ent_5[[w, ds]]\delta &\triangleq \\ \neg\chi[[w]]\delta(\delta_1 \cup [w \mapsto ds], \perp) \end{aligned}$$

Let us look at some uses of the command. To enter a new word  $w$  into the dictionary  $\delta$ , without committing oneself to supplying a definition at the same time, one uses  $Ent_5[[w, \emptyset]]\delta$ . If we want to enter a new word together with a single definition  $d_w$ , then we use  $Ent_5[[w, \{d_w\}]]\delta$ .

Now we consider the case of extending the set of definitions of a word which is already in the dictionary. This particular kind of enter command will be called an ‘update’:

$$\begin{aligned} Upd_5: WORD \times \mathcal{P}DEF &\longrightarrow DICT_5 \longrightarrow DICT_5 \\ Upd_5[[w, ds]]\delta &\triangleq \\ \chi w\delta(\delta + [w \mapsto \delta(w) \cup ds], \perp) \end{aligned}$$

## 10.1.3. The Remove Command

The usual command to remove words (together with their definitions) is given by:

$$\begin{aligned} Rem_5: WORD &\longrightarrow DICT_5 \longrightarrow DICT_5 \\ Rem_5[[w]]\delta &\triangleq \\ \neg\chi[[w]]\delta(\delta, \{w\} \triangleleft \delta) \end{aligned}$$

To remove the definitions of a word without removing the word itself I use a delete command:

$$\begin{aligned} Del_5: WORD \times \mathcal{P}DEF &\longrightarrow DICT_5 \longrightarrow DICT_5 \\ Del_5[[w, ds]]\delta &\triangleq \\ \chi[[w]]\delta(\delta + [w \mapsto ds \triangleleft \delta(w)], \perp) \end{aligned}$$

## 10.1.4. The Lookup Command

If I lookup a word which is in the dictionary then I obtain the set of definitions associated with that word:

$$\begin{aligned} Lkp_5: WORD &\longrightarrow DICT_5 \longrightarrow \mathcal{P}DEF \\ Lkp_5[w]\delta &\triangleq \\ \chi[w]\delta(\delta(w), \perp) \end{aligned}$$

## 10.1.5. The Size Command

To determine the size of the dictionary is similar to that for  $DICT_4$ :

$$\begin{aligned} Size_5: DICT_5 &\longrightarrow \mathbf{N} \\ Size_5(\delta_0) &\triangleq card \circ dom \delta \end{aligned}$$

## 10.1.6. The Associate Command

Given a definition  $d_w$ , we may wish to obtain the set of associated words. Consider a typical dictionary:

$$\delta = [w_1 \mapsto ds_1, w_2 \mapsto ds_2, \dots, w_j \mapsto ds_j, \dots, w_n \mapsto ds_n]$$

The range of  $\delta$  consists of sets of definitions  $\{ds_1, ds_2, \dots, ds_j, \dots, ds_n\}$ , i.e.,  $rng \delta \in \mathcal{P}PDEF$ . Those sets of definitions of which  $d_w$  is a member is determined by

$$D = \{ds_j \mid ds_j \in rng \delta, \chi[d_w]ds_j\}$$

and the corresponding set of associated words may be obtained by iterating over  $D$  with respect to the inverse image of the given definition  $d_w$ ,  $\mathcal{P}\delta^{-1}(d_w)$ . This will give the result, a set of words:

$$\begin{aligned} (\mathcal{P}\delta^{-1}(d_w))D &= \mathcal{P}\delta^{-1}(d_w)\{ds_{j_1}, ds_{j_2}, \dots, ds_{j_k}, \dots, ds_{j_m}\} \\ &= \{\delta^{-1}(d_w)ds_{j_1}, \delta^{-1}(d_w)ds_{j_2}, \dots, \delta^{-1}(d_w)ds_{j_k}, \dots, \delta^{-1}(d_w)ds_{j_m}\} \\ &= \{w_{j_1}, w_{j_2}, \dots, w_{j_k}, \dots, w_{j_m}\} \end{aligned}$$

Now the only difficulty from the point of view of an operator calculus is the implicit nature of  $D$ . I need an operator that effectively generalizes both ‘choice’,  $\in$ , and ‘membership’,  $\chi[-]$ , of an element with respect to a set. First, I give an algorithm  $op_{\triangle}$  to specify the meaning of such an operator:

$$\begin{aligned} op_{\triangle}: X \times \mathcal{P}PX &\longrightarrow \mathcal{P}PX \\ op_{\triangle}(x, S) &\triangleq op_{\triangle}[x, S]\emptyset \end{aligned}$$

# THE DICTIONARY

where the tail-recursive  $\text{op}_{\underline{\underline{\Delta}}}$  is specified by

$$\begin{aligned} \text{op}_{\underline{\underline{\Delta}}}: X \times \mathcal{P}\mathcal{P}X &\longrightarrow \mathcal{P}\mathcal{P}X \longrightarrow \mathcal{P}\mathcal{P}X \\ \text{op}_{\underline{\underline{\Delta}}}\llbracket x, \emptyset \rrbracket T &\stackrel{\Delta}{=} T \\ \text{op}_{\underline{\underline{\Delta}}}\llbracket x, \{A_j\} \uplus S \rrbracket T &\stackrel{\Delta}{=} \\ \chi\llbracket x \rrbracket A_j(\text{op}_{\underline{\underline{\Delta}}}\llbracket x, S \rrbracket T \uplus \{A_j\}, \text{op}_{\underline{\underline{\Delta}}}\llbracket x, S \rrbracket T) & \end{aligned}$$

Then, the specification of the  $\text{Assoc}_5$  command is

$$\begin{aligned} \text{Assoc}_5: \text{DEF} &\longrightarrow \text{DICT}_5 \longrightarrow \text{PWORD} \\ \text{Assoc}_5\llbracket d_w \rrbracket \delta &\stackrel{\Delta}{=} (\mathcal{P}\delta^{-1}(d_w)) \circ (\mathcal{P}(d_w \text{op}_{\underline{\underline{\Delta}}} -)) \circ \text{rng } \delta \end{aligned}$$

From this new baseline I may now proceed to apply the same sort of reifications that I used for  $\text{DICT}_0$ , provide appropriate retrieve functions, carry out the associated proofs and establish another body of formal specification knowledge.

I have supplied sufficient evidence of the method with respect to the models of *dictionary* and made salient remarks in passing to their relationship to the conceptual model of a dictionary. But what is it exactly which makes one think that a dictionary is a dictionary? To elaborate on this critical aspect of the relationship between *VDM* model and conceptual model, I now present another view of ‘dictionary’ in the context of ‘standard’ denotational semantics.

# Appendix B

## The Environment and Store

### 1. Introduction

The *VDM* has historically been associated with the provision of the denotational semantics of programming languages. Perhaps the most widely available account of its application to this domain is the standard reference, *Formal Specification and Software Development* (Bjørner and Jones, 1982). With respect to the Oxford style of denotational semantics, a standard reference is *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory* (Scott 1977). I have said very little about the denotational semantics of programming languages *per se* in this thesis and I do not intend to say much in this Appendix.

The definition of the formal semantics of a programming language is a very large complex specification. As in all major specifications one must find an entry point, a simple model, upon which the rest may be built. For me that starting point was the computer memory or store itself. For this starting point and the subsequent reification that I present here I must acknowledge the material and approach of Cliff B. Jones which I followed (Bjørner and Jones 1982, 85–139).

In itself, the material is of a very elementary nature, sufficiently simple that I might carry out detailed proofs. Just as the dictionary model in the previous Appendix was instrumental in suggesting to me an operator calculus for the *VDM Meta-IV*, the store and environment model was largely responsible for exhibiting the sort of complexity that lay behind pairs of composable maps and for that reason is included for the record. Indeed, I now use it as prerequisite material for the file system case study in Appendix C.

From a conceptual model perspective, the material focuses on the concept of ‘map-splitting’, whereby a single map or table is divided into two distinct maps or tables, which may be recombined under map composition to give the original.

## THE ENVIRONMENT AND STORE

In addition, the splitting of a many-to-one map,  $\mu \in X \xrightarrow{m} Y$ , gives a pair of composable maps  $\varrho \in X \xrightarrow{m} A$  and  $\varsigma \in A \xrightarrow{m} Y$ , such that  $\varrho$  is 1-1 and  $\varsigma$  is many-to-one. There is nothing really intrinsic to denotational semantics.

### 2. Model 0: The Store

To present the framework against which one might wish to build up a model (of the denotational semantics) of a programming language, it seems reasonable to begin with a simple model of a computer store:

$$\varsigma \in STORE_0 = Id \xrightarrow{m} VAL$$

$$i \in Id = \dots$$

$$v \in VAL = \dots$$

which is essentially a map from variable identifiers to values. There is no need to be specific about either the identifier domain or the value domain.

#### 2.1. The Invariant

$$inv\text{-}STORE_0: STORE_0 \longrightarrow \mathbf{B}$$

$$inv\text{-}STORE_0(\varsigma) \triangleq true$$

There are no restrictions on the model.

#### 2.2. The Operations

I propose the following set of operations which are in fact nothing more than names for basic map operators.

##### 2.2.1. New

$$New_0: \longrightarrow STORE_0$$

$$New_0 \triangleq \theta$$

2.2.2. Enter

$$\begin{aligned} Ent_0: Id \times VAL &\longrightarrow STORE_0 \longrightarrow STORE_0 \\ Ent_0[[i, v]]_\varsigma &\triangleq \varsigma \cup [i \mapsto v] \end{aligned}$$

subject to the pre-condition

$$\begin{aligned} pre-Ent_0: Id \times VAL &\longrightarrow STORE_0 \longrightarrow \mathbf{B} \\ pre-Ent_0[[i, v]]_\varsigma &\triangleq \neg \chi[[i]]_\varsigma \end{aligned}$$

This operation ‘suggests’ the declaration of variables to me.

2.2.3. Update

$$\begin{aligned} Upd_0: Id \times VAL &\longrightarrow STORE_0 \longrightarrow STORE_0 \\ Upd_0[[i, v]]_\varsigma &\triangleq \varsigma + [i \mapsto v] \end{aligned}$$

subject to the pre-condition

$$\begin{aligned} pre-Upd_0: Id \times VAL &\longrightarrow STORE_0 \longrightarrow \mathbf{B} \\ pre-Upd_0[[i, v]]_\varsigma &\triangleq \chi[[i]]_\varsigma \end{aligned}$$

The update operation is essentially assignment.

2.2.4. Remove

$$\begin{aligned} Rem_0: Id &\longrightarrow STORE \longrightarrow STORE \\ Rem_0[[i]]_\varsigma &\triangleq \{i\} \triangleleft \varsigma \end{aligned}$$

2.2.5. Is-Recorded

$$\begin{aligned} Rec_0: Id &\longrightarrow STORE_0 \longrightarrow \mathbf{B} \\ Rec_0[[i]]_\varsigma &\triangleq \chi[[i]]_\varsigma \end{aligned}$$

2.2.6. Lookup

$$\begin{aligned} Lkp_0: Id &\longrightarrow STORE_0 \longrightarrow VAL \\ Lkp_0[[i]]_\varsigma &\triangleq \varsigma(i) \end{aligned}$$

subject to the pre-condition

$$\begin{aligned} pre-Lkp_0: Id &\longrightarrow STORE_0 \longrightarrow \mathbf{B} \\ pre-Lkp_0[[i]]_\varsigma &\triangleq \chi[[i]]_\varsigma \end{aligned}$$

# THE ENVIRONMENT AND STORE

## 2.2.7. Size

$$\begin{aligned} \text{Size}_0: \text{STORE}_0 &\longrightarrow \mathbf{N} \\ \text{Size}_0(\varsigma) &\triangleq \text{card} \circ \text{dom} \varsigma \end{aligned}$$

It ought to be clear that the names of the domains and the names of the operations are the links between the purely abstract forms of the *VDM Meta-IV* and some interpretation in a real problem domain.

## 3. Model 1: Environment and Store

In developing a first reification of the original model,  $\text{STORE}_0$ , I have in mind the idea of ‘splitting’ it into two parts: a dictionary part, called the environment,  $\text{ENV}_1$ , and a slightly different model of store,  $\text{STORE}_1$ . In fact, the latter is isomorphic to  $\text{STORE}_0$  and, consequently, I am justified in considering this development step to be an evolution rather than a reification, though I will continue to employ the latter term throughout the section.

$$\begin{aligned} \varrho \in \text{ENV}_1 &= \text{Id} \xrightarrow[m]{} \text{LOC} \\ \varsigma \in \text{STORE}_1 &= \text{LOC} \xrightarrow[m]{} \text{VAL} \\ l \in \text{LOC} &= \mathbf{N} \end{aligned}$$

### 3.1. The Invariant

The conceptual model underlying the reification is that the pair of maps,  $\varrho$  and  $\varsigma$ , may be composed in the usual manner to give the original model. Consequently, the conditions under which such a pair may be composed is encapsulated by the invariant

$$\begin{aligned} \text{inv}: \text{ENV}_1 \times \text{STORE}_1 &\longrightarrow \mathbf{B} \\ \text{inv}(\varrho, \varsigma) &\triangleq \text{rng} \varrho = \text{dom} \varsigma \end{aligned}$$

An immediate corollary follows. The retrieve function must have the form

$$\begin{aligned} \mathfrak{R}_{10}: \text{ENV}_1 \times \text{STORE}_1 &\longrightarrow \text{STORE}_0 \\ \mathfrak{R}_{10}(\varrho, \varsigma) &\triangleq \varsigma \circ \varrho \end{aligned}$$

But suppose for the sake of argument that the invariant were to be violated. In other words, let us apply a Lakatos style of proof and refutation. There are two cases to consider:

- $l \in \text{rng } \varrho \wedge l \notin \text{dom } \varsigma$

This may be interpreted as the dangling pointer/reference problem.

- $l \notin \text{rng } \varrho \wedge l \in \text{dom } \varsigma$

One might consider this to model the omission of returning assigned storage to the free list.

It will become clear from the subsequent analysis that it is impossible to violate the invariant.

### 3.2. The Operations

The operations for this model are exactly the same as those of the baseline model,  $STORE_0$ . I give the operations without much introductory comment and then proceed to verify their correctness both with respect to the invariant and the retrieve function. The purpose in doing so is to elaborate on the properties of a pair of composable maps.

#### 3.2.1. New

$$\begin{aligned} New_1 &: \longrightarrow ENV_1 \times STORE_1 \\ New_1 &\triangleq (\theta, \theta) \end{aligned}$$

*Verify with respect to the invariant:*

HYPOTHESIS B.1. *true*

CONCLUSION B.1.  $\text{rng } \theta = \text{dom } \theta$

*Proof:* Trivial.

*Verify with respect to the retrieve function:*

$$\begin{aligned} \mathfrak{R}_{10} \circ New_1 & \\ &= \mathfrak{R}_{10}(\theta, \theta) \\ &= \theta \circ \theta \\ &= \theta \\ &= New_0 \end{aligned}$$

# THE ENVIRONMENT AND STORE

## 3.2.2. Enter

The enter operation is intended to model, in an approximate manner, the declaration of variables in a programming language. By ‘approximate’, I mean that the operation suggests the modelling of variable declaration.

$$\begin{aligned}
 Ent_1: Id \times VAL &\longrightarrow ENV_1 \times STORE_1 \longrightarrow ENV_1 \times STORE_1 \\
 Ent_1[[i, v]](\varrho, \varsigma) &\triangleq \\
 \text{let } l \in LOC \setminus rng \varrho \text{ in} & \\
 (\varrho \cup [i \mapsto l], \varsigma \cup [l \mapsto v]) &
 \end{aligned}$$

subject to the pre-condition

$$\begin{aligned}
 pre-Ent_1: Id \times VAL &\longrightarrow ENV_1 \times STORE_1 \longrightarrow \mathbf{B} \\
 pre-Ent_1[[i, v]](\varrho, \varsigma) &\triangleq \neg \chi[[i]]\varrho
 \end{aligned}$$

*Verify with respect to the invariant:*

HYPOTHESIS B.2.  $rng \varrho = dom \varsigma$

CONCLUSION B.2.  $rng(\varrho \cup [i \mapsto l]) = dom(\varsigma \cup [l \mapsto v])$

*Proof:*

$$\begin{aligned}
 &rng(\varrho \cup [i \mapsto l]) \\
 &= rng \varrho \cup \{l\} \quad \text{-- and } l \notin rng \varrho \text{ by construction}
 \end{aligned}$$

Also

$$\begin{aligned}
 &dom(\varsigma \cup [l \mapsto v]) \\
 &= dom \varsigma \cup \{l\} \quad \text{-- since } dom \text{ is a homomorphism with respect to } \Delta \\
 &\quad \text{-- and, of course, } l \notin dom \varsigma, \text{ by definition}
 \end{aligned}$$

and the result follows. Note carefully the justification for  $dom(\varsigma \cup [l \mapsto v]) = dom \varsigma \cup \{l\}$ . Since map extend does not give a monoid, then I had to move up to map symmetric difference and, of course, under domain-disjointness, the map extend operator may be regarded as a particularisation of symmetric difference.

Verify with respect to the retrieve function:

$$\begin{aligned}
 & \mathfrak{R}_{10} \circ \text{Ent}_1[[i, v]](\varrho, \varsigma) \\
 &= \text{let } l \in \text{LOC} \setminus \text{rng } \varrho \text{ in} \\
 & \quad \mathfrak{R}_{10}(\varrho \cup [i \mapsto l], \varsigma \cup [l \mapsto v]) \\
 &= \text{let } l \in \text{LOC} \setminus \text{rng } \varrho \text{ in} \\
 & \quad (\varsigma \cup [l \mapsto v]) \circ (\varrho \cup [i \mapsto l]) \\
 & \quad - - \text{ and it is quite evident that one **expects** the result} \\
 &= \text{let } l \in \text{LOC} \setminus \text{rng } \varrho \text{ in} \\
 & \quad (\varsigma \circ \varrho) \cup ([l \mapsto v] \circ [i \mapsto l]) \\
 &= (\varsigma \circ \varrho) \cup [i \mapsto v]
 \end{aligned}$$

Thus the verification depends on the equality

$$(\varsigma \cup [l \mapsto v]) \circ (\varrho \cup [i \mapsto l]) = (\varsigma \circ \varrho) \cup ([l \mapsto v] \circ [i \mapsto l])$$

The pairs of maps  $(\varrho \cup [i \mapsto l], \varsigma \cup [l \mapsto v])$  and  $(\varrho, \varsigma)$  are both composable by virtue of the invariant. Clearly,  $([i \mapsto l], [l \mapsto v])$  is also a composable pair of maps. Some simple sketches confirms our intuition of the correctness of the result. We may then propose a general lemma on composable maps.

LEMMA B.1. *Let  $(\varrho_j, \varsigma_j)$  and  $(\varrho_k, \varsigma_k)$  be pairs of composable maps such that  $\text{dom } \varrho_j \cap \text{dom } \varrho_k = \emptyset$  and  $\text{dom } \varsigma_j \cap \text{dom } \varsigma_k = \emptyset$ . Then*

$$(\varsigma_j \cup \varsigma_k) \circ (\varrho_j \cup \varrho_k) = (\varsigma_j \circ \varrho_j) \cup (\varsigma_k \circ \varrho_k)$$

Now I am much more interested in establishing an equivalent lemma for map override, since it is this operator that gives us a monoid of maps.

### 3.2.3. Update

The update operation essentially models the assignment statement.

$$\begin{aligned}
 \text{Upd}_1: \text{Id} \times \text{VAL} &\longrightarrow \text{ENV}_1 \times \text{STORE}_1 \longrightarrow \text{ENV}_1 \times \text{STORE}_1 \\
 \text{Upd}_1[[i, v]](\varrho, \varsigma) &\triangleq (\varrho, \varsigma + [\varrho(i) \mapsto v])
 \end{aligned}$$

subject to the pre-condition

$$\begin{aligned}
 \text{pre-Upd}_1: \text{Id} \times \text{VAL} &\longrightarrow \text{ENV}_1 \times \text{STORE}_1 \longrightarrow \mathbf{B} \\
 \text{pre-Upd}_1[[i, v]](\varrho, \varsigma) &\triangleq \chi[[i]]\varrho \wedge \chi[[\varrho(i)]]\varsigma
 \end{aligned}$$

## THE ENVIRONMENT AND STORE

*Verify with respect to the invariant:*

HYPOTHESIS B.3.  $\text{rng } \varrho = \text{dom } \varsigma$

CONCLUSION B.3.  $\text{rng } \varrho = \text{dom}(\varsigma + [\varrho(i) \mapsto v])$

*Proof:*

$$\begin{aligned} & \text{dom}(\varsigma + [\varrho(i) \mapsto v]) \\ &= \text{dom } \varsigma \cup \{\varrho(i)\} \quad \text{-- since } \text{dom} \text{ is a homomorphism with respect to override} \\ &= \text{dom } \varsigma \quad \text{-- by the pre-condition} \\ &= \text{rng } \varrho \quad \text{-- by the hypothesis} \end{aligned}$$

*Verify with respect to the retrieve function:*

$$\begin{aligned} & \mathfrak{R}_{10} \circ \text{Upd}_1[[i, v]](\varrho, \varsigma) \\ &= \mathfrak{R}_{10}(\varrho, \varsigma + [\varrho(i) \mapsto v]) \\ &= (\varsigma + [\varrho(i) \mapsto v]) \circ \varrho \\ &\quad \text{-- and performing the usual sort of mathematical trick} \\ &= (\varsigma + [\varrho(i) \mapsto v]) \circ (\varrho + [i \mapsto \varrho(i)]) \\ &\quad \text{-- one expects to obtain} \\ &= (\varsigma \circ \varrho) + ([\varrho(i) \mapsto v] \circ [i \mapsto \varrho(i)]) \\ &= (\varsigma \circ \varrho) + [i \mapsto v] \end{aligned}$$

I can still recall the first time that I discovered the ‘mathematical trick’ mentioned above. It may seem to be such a simple matter, hardly worth drawing attention to. Similar techniques abound in ‘conventional’ mathematics. That the same sort of technique would be necessary in carrying out proofs in the *VDM* came as a sort of surprise. I had always argued that ‘doing’ formal specifications was simply a matter of ‘doing’ mathematics, albeit in a discrete domain. I certainly did not realise the full import of my belief. It was just such simple discoveries that led to the conviction expressed throughout this thesis. There is an enormous wealth of basic mathematics in formal specifications just waiting to be discovered, a body of knowledge which **must** form the foundation for software engineering. That little progress has been made in this matter is unfortunately due to the over-emphasis on the supposedly important rôle that formal predicate logic has to play.

Just as in the case of the proof of the retrieve function with respect to the  $\text{Ent}_1$  operation, we have stumbled across another equality that needs to be verified:

$$(\varsigma + [\varrho(i) \mapsto v]) \circ (\varrho + [i \mapsto \varrho(i)]) = (\varsigma \circ \varrho) + ([\varrho(i) \mapsto v] \circ [i \mapsto \varrho(i)])$$

suggesting another general lemma. Since the override operation may be defined in terms of the extend operation, then we immediately have

$$\begin{aligned}
 & (\varsigma + [\varrho(i) \mapsto v]) \circ (\varrho + [i \mapsto \varrho(i)]) \\
 &= ((\{ \varrho(i) \} \triangleleft \varsigma) \cup [\varrho(i) \mapsto v]) \circ ((\{ i \} \triangleleft \varrho) \cup [i \mapsto \varrho(i)]) \\
 &= ((\{ \varrho(i) \} \triangleleft \varsigma) \circ (\{ i \} \triangleleft \varrho)) \cup ([\varrho(i) \mapsto v] \circ [i \mapsto \varrho(i)]) \\
 &\quad - - \text{by the lemma for map extend with respect to composition} \\
 &= (\{ i \} \triangleleft (\varsigma \circ \varrho)) \cup [i \mapsto v] \\
 &\quad - - \text{WHICH MUST BE JUSTIFIED!!!} \\
 &= (\varsigma \circ \varrho) + [i \mapsto v]
 \end{aligned}$$

The construction of this proof appeared to expose a serious flaw in the model which was first pointed out to me by Michael Butler in an undergraduate class on VDM. To illustrate the problem, let us consider a simple counter-example:

$$\varrho = [i \mapsto l, j \mapsto l], \varsigma = [l \mapsto v]$$

Then, the application of the removal operator to  $\varrho$  with respect to  $i$ , and to  $\varsigma$  with respect to  $\varrho(i) = l$ , gives

$$\begin{aligned}
 & (\{ l \} \triangleleft \varsigma) \circ (\{ i \} \triangleleft \varrho) \\
 &= \theta \circ [j \mapsto l] \\
 &= \theta
 \end{aligned}$$

whereas

$$\begin{aligned}
 & \{ i \} \triangleleft (\varsigma \circ \varrho) \\
 &= \{ i \} \triangleleft [i \mapsto v, j \mapsto v] \\
 &= [j \mapsto v]
 \end{aligned}$$

Clearly the difficulty arises only in the event that  $\varrho$  is not a 1-1 map. One might suppose that the remedy is to be found in tightening up the invariant to ensure that the environment is always a 1-1 map. But, then I would lose the possibility of interpreting the environment model in terms of identifier aliases. When I re-visit the operations on this model to investigate whether such aliasing can arise, I find that the pertinent operation is the  $Ent_1$  command which only allows the construction of 1-1  $\varrho$  maps. Hence, by carrying out proofs, even for simple models, I discover a certain amount of complexity with respect to interpretation of the model in a real problem domain. In this case, we see inter-relationships between the invariant, the retrieve function, the proof of the verification of the latter with respect to the  $Upd_1$  command, and the pre-condition of the  $Ent_1$  command. Where more complex

## THE ENVIRONMENT AND STORE

models are in question, such as the file system case study in the next Appendix, we ought to expect to encounter an even greater degree of inter-relationships between the various parts. Therefore, one must draw the conclusion that every model implies the existence of a corresponding theory.

Returning to the equality that triggered off these remarks, it is also evident that it is sufficient to constrain the  $Upd_1$  command to those identifiers in the domain of the environment which give rise to 1–1 submaps and to introduce a new operation into the model that permits the construction of many-to-one submaps. This is by far the most fruitful approach. The general proposed lemma is, therefore

LEMMA B.2. *Let  $(\varrho_j, \varsigma_j)$  and  $(\varrho_k, \varsigma_k)$  be pairs of composable maps such that  $\text{dom } \varrho_j \supseteq \text{dom } \varrho_k$ ,  $\text{dom } \varsigma_j \supseteq \text{dom } \varsigma_k$ , and  $\varrho_j, \varrho_k$  are 1–1 maps, i.e.,  $|\text{dom } \varrho_j| = |\text{rng } \varrho_j|$  and  $|\text{dom } \varrho_k| = |\text{rng } \varrho_k|$ . Then*

$$(\varsigma_j + \varsigma_k) \circ (\varrho_j + \varrho_k) = (\varsigma_j \circ \varrho_j) + (\varsigma_k \circ \varrho_k)$$

### 3.2.4. Remove

If we are prepared to agree that the environment/symbol table is extended upon the declaration of new variables, then we will also wish to provide for the possibility of removing said variables once the ‘program pointer’ leaves the scope of their declaration and use. The remove operation models this concept:

$$\begin{aligned} \text{Rem}_1: \text{Id} &\longrightarrow \text{ENV}_1 \times \text{STORE}_1 \longrightarrow \text{ENV}_1 \times \text{STORE}_1 \\ \text{Rem}_1[[i]](\varrho, \varsigma) &\triangleq (\{i\} \triangleleft \varrho, \{\varrho(i)\} \triangleleft \varsigma) \end{aligned}$$

*Verify with respect to the invariant:*

HYPOTHESIS B.4.  $\text{rng } \varrho = \text{dom } \varsigma$

CONCLUSION B.4.  $\text{rng}(\{i\} \triangleleft \varrho) = \text{dom}(\{\varrho(i)\} \triangleleft \varsigma)$

*Proof:* Recall that there are no pre-conditions for the remove operation. Hence, there are two cases to consider. Let us consider the first case:

case  $i \notin \text{dom } \varrho$

$$\text{rng}(\{i\} \triangleleft \varrho) = \text{rng } \varrho$$

and

$$\text{dom}(\{\varrho(i)\} \triangleleft \varsigma) = \dots \quad \text{-- which does not make any sense}$$

Although, the remove operation on the  $STORE_0$  model was a valid monoid endomorphism, the reification stage has introduced a complication. If  $i$  is not in the domain of  $\varrho$ , then it is not possible to refer to the element  $\varrho(i)$  in the domain of  $\varsigma$ . We must include a guard into the remove operation for this case, and the proof of verification with respect to the retrieve operation reduces to the trivial one. In the case that  $i \in \text{dom } \varrho$ , we immediately have

$$\begin{aligned} \text{dom}(\{\varrho(i)\} \triangleleft \varsigma) &= \{\varrho(i)\} \triangleleft \text{dom } \varsigma \\ \text{and} \\ \text{rng}(\{i\} \triangleleft \varrho) &= \{\varrho(i)\} \triangleleft \text{rng } \varrho \end{aligned}$$

and the conclusion follows.

*Verify with respect to the retrieve function:*

$$\begin{aligned} &\mathfrak{R}_{10} \circ \text{Rem}_1[[i]](\varrho, \varsigma) \\ &= \mathfrak{R}_{10}(\{i\} \triangleleft \varrho, \{\varrho(i)\} \triangleleft \varsigma) \\ &= (\{\varrho(i)\} \triangleleft \varsigma) \circ (\{i\} \triangleleft \varrho) \\ &\quad \text{-- with the expectation that} \\ &= \{i\} \triangleleft (\varsigma \circ \varrho) \end{aligned}$$

We have already observed, in the case of the  $Upd_1$  command, under what conditions we may validly deduce this result.

The remaining three operations do not involve a  $ENV_1 \times STORE_1$  to  $ENV_1 \times STORE_1$  transformation and, therefore, there is no question of verifying with respect to the invariant. The verification with respect to the retrieve function also takes on a slightly different form.

### 3.2.5. Is-Recorded

We may need to know whether or not a particular variable has been declared:

$$\begin{aligned} \text{Rec}_1: Id &\longrightarrow ENV_1 \times STORE_1 \longrightarrow \mathbf{B} \\ \text{Rec}_1[[i]](\varrho, \varsigma) &\triangleq \chi[[i]]\varrho \wedge \chi[[\varrho(i)]]\varsigma \end{aligned}$$

*Verify with respect to the retrieve function:* We are required to prove that

$$\text{Rec}_0[[i]] \circ \mathfrak{R}_{10} = \mathcal{I} \circ \text{Rec}_1[[i]] = \text{Rec}_1[[i]]$$

# THE ENVIRONMENT AND STORE

Taking the left-hand-side we have

$$\begin{aligned} & Rec_0[[i]] \circ \mathfrak{R}_{10}(\varrho, \varsigma) \\ &= Rec_0[[i]](\varsigma \circ \varrho) \\ &= \chi[[i]](\varsigma \circ \varrho) \end{aligned}$$

and the right-hand-side is clearly just a repetition of the definition of  $Rec_1$

$$\begin{aligned} & Rec_1[[i]](\varrho, \varsigma) \\ &= \chi[[i]]\varrho \wedge \chi[[\varrho(i)]]\varsigma \end{aligned}$$

What is one to infer from this proof? I have concluded that whenever one subjects a non-transformational operation to a retrieve function, one exhibits basic properties of the model. In this case, we have the property that if  $i$  is recorded in the domain of  $\varrho$ , then it must also be true that  $\varrho(i)$  is recorded in the domain of  $\varsigma$ , and vice-versa. This is precisely what the following equation asserts:

$$\chi[[i]](\varsigma \circ \varrho) = \chi[[i]]\varrho \wedge \chi[[\varrho(i)]]\varsigma$$

### 3.2.6. Lookup

Naturally, given a variable, one needs to be able to find the associated address,  $l \in LOC$ , in order to retrieve its value from the store:

$$\begin{aligned} & Lkp_1: Id \longrightarrow ENV_1 \times STORE_1 \longrightarrow VAL \\ & Lkp_1[[i]](\varrho, \varsigma) \triangleq \varsigma(\varrho(i)) \end{aligned}$$

subject to the pre-condition

$$\begin{aligned} & pre-Lkp_1: Id \longrightarrow ENV_1 \times STORE_1 \longrightarrow \mathbf{B} \\ & pre-Lkp_1[[i]](\varrho, \varsigma) \triangleq \chi[[i]]\varrho \wedge \chi[[\varrho(i)]]\varsigma \end{aligned}$$

*Verify with respect to the retrieve function:* Required to prove that

$$Lkp_0[[i]] \circ \mathfrak{R}_{10} = \mathcal{I} \circ Lkp_1[[i]] = Lkp_1[[i]]$$

Taking the left-hand-side we obtain

$$\begin{aligned} & Lkp_0[[i]] \circ \mathfrak{R}_{10}(\varrho, \varsigma) \\ &= Lkp_0[[i]](\varsigma \circ \varrho) \\ &= (\varsigma \circ \varrho)(i) \end{aligned}$$

and from the right-hand-side

$$\begin{aligned} Lkp_1[[i]](\varrho, \varsigma) \\ = \varsigma(\varrho(i)) \end{aligned}$$

giving the important equality

$$\varsigma(\varrho(i)) = (\varsigma \circ \varrho)(i)$$

This basic property may seem to be self-evident. That is certainly true in conventional mathematics. However, it is not always so obvious in *VDM* models. In fact, establishing such an equality proved to be one of the major issues in my analysis of the file system case study.

### 3.2.7. Size

To complete this section, I would like to demonstrate that one may employ the retrieve function to infer a ‘useful’ definition for an operation. Consider the following outline:

$$\begin{aligned} Size_1: ENV_1 \times STORE_1 &\longrightarrow \mathbf{N} \\ Size_1(\varrho, \varsigma) &\triangleq \dots \end{aligned}$$

*Verify with respect to the retrieve function:* Required to prove

$$Size_0 \circ \mathfrak{R}_{10} = \mathcal{I} \circ Size_1 = Size_1$$

From the left-hand-side we obtain

$$\begin{aligned} Size_0 \circ \mathfrak{R}_{10}(\varrho, \varsigma) \\ = Size_0(\varsigma \circ \varrho) \\ = card \circ dom(\varsigma \circ \varrho) \\ - - \text{and knowing what map composition entails I write} \\ = card \circ dom \varrho \end{aligned}$$

giving me the required definition for  $Size_1$ :

$$Size_1(\varrho, \varsigma) \triangleq card \circ dom \varrho$$

The store,  $\varsigma$ , plays no rôle in the definition.

## 4. Summary

I would not have become aware of some of the interesting, though ‘peculiar’, properties of composable maps had I not carried out verification proofs with respect to invariants and retrieve functions. The proofs I regard in much the same light as the old fashioned geometrical-construction style proofs. Most importantly, I began to look with suspicion on proposed invariants and retrieve functions. At all times my goal in developing formal specifications was to simplify, to convert verbosity into the symbolic, to make *VDM* specifications tractable to an operator calculus, to extract the æsthetic from the ugly. The next and final Appendix presents that problem domain upon which I tested my ideas over a period of three years.

# Appendix C

## The File System

### 1. Introduction

A detailed analysis of the classic case study example of the file system, taken from Bjørner and Jones (1982, 353–77), is given in this Appendix primarily in order to demonstrate further the applicability of the operator calculus of the Irish School of the *VDM* and to support the arguments presented in Chapter 5. It also complements the material in Appendix A on the dictionary and that of Appendix B on the environment and store. The example is sufficiently well-known within the circle of the *VDM* community and is, consequently, ideal for the exposition of my arguments.

The published text does not give the details of all the operations at every level of reification; nor are there any real demonstrations of the applications of the various retrieve functions. This offered me an opportunity to explain at length my method.

Particular attention is paid to the form and use of the stated invariants and their rôle both with respect to the formulation of the retrieve functions and the proof of the validity of the given operations. In many cases I present my theorems, “problems to prove”, in the classical form of hypothesis followed by conclusion (Pólya [1945] 1957,155), an approach which was particularly evident in the previous Appendix. I believe that their formulation and proof is much better than, say, the form of the following ‘theorem’ taken from the original text (Bjørner and Jones 1982, 363):

$$\begin{aligned} thm_1 \quad & (\forall c \in Cmd) \\ & (\forall \varphi_0 \in FS_0) \\ & (\forall \varphi_1 \in \varphi_1) \\ & (((inv-\varphi_1(\varphi_1) \wedge retr-FS_0(\varphi_1) = \varphi_0) \\ & \quad \wedge pre-Elab-Cmd[c](\varphi_0)) \\ & \quad \supset \\ & (retr-RES_1(Elab-Cmd_2[c](\varphi_2)) = Elab-Cmd_1[c](\varphi_0))) \end{aligned}$$

A few remarks on the particular choice of style, in which I have chosen to present

## THE FILE SYSTEM

the material, are in order. Rather than use the denotational style, I have decided on the simpler curried style that pervades the thesis. Thus, for example in version 0, instead of writing

$$\begin{aligned} \text{Int-Crea}_0: \text{Crea}_0 &\longrightarrow \text{FS}_0 \longrightarrow \text{FS}_0 \\ \text{Int-Crea}_0 \llbracket \text{mk-Crea}_0(fn) \rrbracket \varphi &\triangleq \dots \end{aligned}$$

where  $\text{Crea}_0 :: Fn$ , I preferred

$$\begin{aligned} \text{Crea}_0: Fn &\longrightarrow \text{FS}_0 \longrightarrow \text{FS}_0 \\ \text{Crea}_0 \llbracket fn \rrbracket \varphi &\triangleq \dots \end{aligned}$$

Again instead of using the constructor ‘*mk-*’ on trees, I have chosen to employ my ‘for *mk-Syn(...)* use ...’ clause. Thus, in the case of version 1, instead of writing  $\text{mk-}\varphi_1(\kappa, \tau, \varpi)$ , I use simply  $(\kappa, \tau, \varpi)$ , which leads me to remarks on my use of Greek characters.

I am ever mindful of the importance that is stressed on ‘giving meaningful names’ to things in computing. It has been customary to employ Greek characters in the Oxford style of denotational semantics and ‘meaningful names’ in the VDM style and I tend towards the latter, which I recommend strongly to initiates, rather than the former. However, in practice, especially in the application of the operator calculus, I have discovered the hard way that ‘meaningful names’ are more likely to obfuscate than clarify. In general, I use lower case Greek characters only to denote elements of a map domain. Although I have tried to tie the Greek characters to the objects in question by phonetic associations such as

$$\begin{aligned} \text{directory} &\longmapsto \delta \in \text{DIR} \\ \text{file system} &\longmapsto \varphi \in \text{FS} \\ \text{katalogue} &\longmapsto \kappa \in \text{CTLG} \end{aligned}$$

I have not always been successful. For example, for a directory system I use  $\tau$ . I might have used the Greek capital  $\Delta$ . But this would conflict with the symmetric difference operator. For a paging system I use  $\varpi$ , a variant of  $\pi$ , since the latter is my customary projection operator.

Finally recall that my use of the symbol,  $\perp$ , is to be interpreted as ‘let us not worry yet’ since its interpretation will depend on the context into which the specification will be embedded (see especially Chapter 8 on Communications and Behaviour). It is **not** to be interpreted as ‘error’ or ‘undefined’.

## 2. File System — Version 0

Although I am working from the case study of the file system, which I have already mentioned in the Introduction, is contained in Bjørner and Jones (1982), it is important to note that the highest level abstract model, to be discussed in this section, already appeared in the earliest widespread publication of the *VDM—The Vienna Development Method: The Meta-Language* (Bjørner and Jones 1978, 78; 81; 85–7; 91–2; 346). In particular, the *Meta-IV* on p.92 of the work cited contains much of the essentials of the semantic functions to be discussed. A more self-contained version of the model appeared later in (Bjørner 1980, 148–55). Its continued existence as a basic *Meta-IV* example of map domains seems to be ensured, since it occurs again in (Bjørner 1988, II: 452–6). A specification of a similar system in the  $\mathcal{Z}$  style is to be found in (Morgan and Suffrin 1984, 128–42). Let us now consider the domain of discourse, the principal part of which is the set of semantic domain equations.

### 2.1. Semantic Domains

A file system,  $FS_0$ , is considered to be a map from file names,  $F_n$ , to files,  $FILE$ . In turn, a file is modelled as a map from page names,  $P_n$ , to pages,  $PG$ . In summary

$$FS_0 = F_n \xrightarrow{m} FILE$$

$$FILE = P_n \xrightarrow{m} PG$$

The most significant point that we need note here is that this model is of the form

$$MODEL_0 = X \xrightarrow{m} (Y \xrightarrow{m} Z)$$

and recalls a curried function. Without even advancing any further, the formal ‘methodist’ ought to be able to develop the theory of  $MODEL_0$  completely in the abstract. For example, for all  $\mu \in MODEL_0$ , we have

$$|rng \mu| \leq |dom \mu|$$

with strict equality only in the case that  $\mu$  is 1–1. Rather than work in the abstract, we will present the theory in the concrete, i.e., for  $MODEL_0 = FS_0$ , and *that* in the context of a discussion of the invariant which follows. In order that this Appendix

# THE FILE SYSTEM

might be reasonably self-contained I have been forced to repeat some of the earlier remarks of Chapter 5.

## 2.2. The Invariant

The invariant stated in the original text is simply

$$\begin{aligned} \text{inv-}FS_0: FS_0 &\longrightarrow \mathbf{B} \\ \text{inv-}FS_0(\varphi) &\stackrel{\Delta}{=} \text{true} \end{aligned}$$

Taken literally, this means that there are no restrictions on the domain of discourse  $FS_0$ . Whatever satisfies the domain equations must be invariant at any subsequent level of reification. There is a tendency among formal methodists with a logical bent to take an invariant such as this at face value and hurry on to the real material. On the other hand, the suspicions of those, with a mathematical bent of the ‘discovery’ or ‘proof and refutation’ kind, are immediately aroused. There *are* non-trivial invariants here which have not been expressed, ‘hidden’ lemmas, if you will. Everything that follows, the whole process of reification, is going to depend on the validity of this first abstract model. Therefore, it behoves us to identify its properties, its characteristics.

As stated in Chapter 5, there is a non-trivial invariant which constrains  $FS_0$ , “*every page belongs to exactly one file*”, an invariant which is forced upon the model as a result of considering the invariant of the first level of reification,  $FS_1$ , and arguing that invariants must be preserved with respect to the retrieve function,  $\mathfrak{R}_{10}$ . Here I wish to demonstrate how such an invariant is constructed.

For definiteness, I shall consider a simple example taken from the original text and extended, in the sense of the map extend operator, to make it more interesting:

$$\begin{aligned} \varphi &= [fn_1 \mapsto \mu_1, fn_2 \mapsto \mu_2, fn_3 \mapsto \mu_3] \cup [fn_4 \mapsto \mu_1, fn_5 \mapsto \mu_3, fn_6 \mapsto \mu_4] \\ \mu_1 &= [pn_1 \mapsto pg_1, pn_2 \mapsto pg_2] \\ \mu_2 &= [pn_3 \mapsto pg_3] \\ \mu_3 &= \theta \\ \mu_4 &= \mu_1 \cup \mu_2 \cup \mu_3 \cup [pn_4 \mapsto pg_1, pn_5 \mapsto pg_3] \end{aligned}$$

If someone objects that this example is not correct with respect to a real file system, for which the model is a specification, then we have found a counter-example. Whether or not there is such a person who can make a judgment of this sort, we are

fully justified in using the example to ‘test’ the specification. Now let me elaborate on the reasons for my choice of example.

1. Both  $fn_3$  and  $fn_5$  name the empty file. This corresponds quite naturally to issuing two successive ‘create file’ commands. There does not appear to be anything extraordinary in this. But when we come to consider the first level of reification, a pair of 1–1 map domains are introduced,  $CTLG_1$  and  $DIRS_1$ , which have the desirable property that they are composable  $DIRS_1 \circ CTLG_1$ , i.e., an invariant is fixed, such that for all  $\kappa \in CTLG$  and  $\tau \in DIRS$ , the composite  $\tau \circ \kappa$  is well-defined. So far, so good. In addition, because both  $\kappa$  and  $\tau$  are 1–1, then so is the composite  $\tau \circ \kappa$ . But, the overall intention of the reification is that under the retrieve function,  $Crea_1$  should correspond to  $Crea_0$  and this is where the problem arises. For supposing that two successive  $Crea_1$  commands are issued, which would correspond to our example, then the partial system

$$\kappa = [fn_i \mapsto dn_i, fn_j \mapsto dn_j]$$

$$\tau = [dn_i \mapsto \theta, dn_j \mapsto \theta]$$

would be constructed, giving  $\tau \circ \kappa = [fn_i \mapsto \theta, fn_j \mapsto \theta]$ . Technically,  $\tau \circ \kappa$  is no longer a 1–1 map! The problem does not lie in the abstract specification of  $FS_0$  but rather in the reification from general maps to 1–1 maps!

2. The other feature of my example is  $\mu_4$  which not only incorporates the notion of distinct page names mapping to the same page, but also distinct page names from *different* files mapping to the same page. I believe that this is not intended in the model. But there is nothing to exclude it! (In fact when we come to consider the semantics of the commands, the only other place where I might be prohibited from arriving at my example, we note that their well-formedness is always true)!

Now we will continue with our consideration of the test example. Since we are dealing with map domains, the first stage in development will be to apply the domain and range operators to the first domain equation:

$$dom \varphi = \{fn_1, fn_2, fn_3, fn_4, fn_5, fn_6\}$$

$$rng \varphi = \{\mu_1, \mu_2, \mu_3, \mu_4\}$$

giving the particular invariant  $|rng \varphi| < |dom \varphi|$  for this example. For any example

## THE FILE SYSTEM

of a file system,  $\varphi_j$ , we always have the invariant

$$|rng \varphi_j| \leq |dom \varphi_j|$$

with strict equality only in the case that  $\varphi$  is 1-1. Consideration of the second domain equation gives

$$\begin{aligned} dom \mu_1 &= \{pn_1, pn_2\} & rng \mu_1 &= \{pg_1, pg_2\} \\ dom \mu_2 &= \{pn_3\} & rng \mu_2 &= \{pg_3\} \\ dom \mu_3 &= \emptyset & rng \mu_3 &= \emptyset \\ dom \mu_4 &= \{pn_1, pn_2, pn_3, pn_4, pn_5\} & rng \mu_4 &= \{pg_1, pg_2, pg_3\} \end{aligned}$$

This may be written compactly using the powerset functor

$$\begin{aligned} \mathcal{P} dom \circ rng \varphi &= \mathcal{P} dom \{\mu_1, \mu_2, \mu_3, \mu_4\} \\ &= \{dom \mu_1, dom \mu_2, dom \mu_3, dom \mu_4\} \\ &= \{\{pn_1, pn_2\}, \{pn_3\}, \emptyset, \{pn_1, pn_2, pn_3, pn_4, pn_5\}\} \\ \mathcal{P} rng \circ rng \varphi &= \mathcal{P} rng \{\mu_1, \mu_2, \mu_3, \mu_4\} \\ &= \{rng \mu_1, rng \mu_2, rng \mu_3, rng \mu_4\} \\ &= \{\{pg_1, pg_2\}, \{pg_3\}, \emptyset, \{pg_1, pg_2, pg_3\}\} \end{aligned}$$

giving us the invariants

$$\begin{aligned} |dom \mu_1| &= |rng \mu_1|, \\ |dom \mu_2| &= |rng \mu_2|, \\ |dom \mu_3| &= |rng \mu_3|, \\ |dom \mu_4| &> |rng \mu_4| \end{aligned}$$

Now, if we take the distributed union of  $\mathcal{P} dom \circ rng \varphi$  and  $\mathcal{P} rng \circ rng \varphi$ , we will obtain the set of all page names and the set of all pages used in the file system,  $\varphi$ , respectively:

$$\begin{aligned} \cup / \circ \mathcal{P} dom \circ rng \varphi &= \{pn_1, pn_2, pn_3, pn_4, pn_5\} \\ \cup / \circ \mathcal{P} rng \circ rng \varphi &= \{pg_1, pg_2, pg_3\} \end{aligned}$$

Another very important primitive development step is to take inverse images:

$$\begin{aligned} \varphi^{-1} \mu_1 &= \{fn_1, fn_4\} \\ \varphi^{-1} \mu_2 &= \{fn_2\} \\ \varphi^{-1} \mu_3 &= \{fn_3, fn_5\} \\ \varphi^{-1} \mu_4 &= \{fn_6\} \end{aligned}$$

which gives the partition of  $\text{dom } \varphi$  which may be expressed as

$$\mathcal{P}\varphi^{-1} \circ \text{rng } \varphi = \{\{fn_1, fn_4\}, \{fn_2\}, \{fn_3, fn_5\}, \{fn_6\}\}$$

and, similarly for each file  $\mu_j$ , we obtain

$$\mu_1^{-1} \circ \text{rng } \mu_1 = \{pn_1, pn_2\}$$

$$\mu_2^{-1} \circ \text{rng } \mu_2 = \{pn_3\}$$

$$\mu_3^{-1} \circ \text{rng } \mu_3 = \emptyset$$

$$\mu_4^{-1} \circ \text{rng } \mu_4 = \{pn_1, pn_4, pn_2, pn_3, pn_5\}$$

which give partitions of the respective file domains,  $\text{dom } \mu_1$ ,  $\text{dom } \mu_2$ ,  $\text{dom } \mu_3$  and  $\text{dom } \mu_4$ . One might wish to express this collection of partitions in the form

$$\mathcal{P}((-)^{-1} \circ \text{rng } -) \circ \text{rng } \varphi$$

To check the validity of the expression, one may apply it to the example:

$$\begin{aligned} & \mathcal{P}((-)^{-1} \circ \text{rng } -) \circ \text{rng } \varphi \\ &= \mathcal{P}((-)^{-1} \circ \text{rng } -)(\{\mu_1, \mu_2, \mu_3, \mu_4\}) \\ &= \{((-)^{-1} \circ \text{rng } -)\mu_1, ((-)^{-1} \circ \text{rng } -)\mu_2, ((-)^{-1} \circ \text{rng } -)\mu_3, ((-)^{-1} \circ \text{rng } -)\mu_4\} \\ &= \{\mu_1^{-1} \circ \text{rng } \mu_1, \mu_2^{-1} \circ \text{rng } \mu_2, \mu_3^{-1} \circ \text{rng } \mu_3, \mu_4^{-1} \circ \text{rng } \mu_4\} \\ &= \{\{pn_1, pn_2\}, \{pn_3\}, \emptyset, \{pn_1, pn_4, pn_2, pn_3, pn_5\}\} \end{aligned}$$

In formalising the invariant for  $FS_0$ , I decided to try to capture the notion that

- 1) the set of all pages in the file system was partitioned

$$\Delta / \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi = \cup / \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi$$

$$\wedge |\oplus / \mathcal{P} j \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi| = |\cup / \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi|$$

where  $j: e \mapsto [e \mapsto 1]$  is the injection operator that takes elements of a set into singleton bags, and  $\oplus$  is the bag addition operator.

- 2) the set of all page names in the file system was partitioned

$$\Delta / \circ \mathcal{P} \text{dom} \circ \text{rng } \varphi = \cup / \circ \mathcal{P} \text{dom} \circ \text{rng } \varphi$$

$$\wedge |\oplus / \mathcal{P} j \circ \mathcal{P} \text{dom} \circ \text{rng } \varphi| = |\cup / \circ \mathcal{P} \text{dom} \circ \text{rng } \varphi|$$

- 3) and it must necessarily be the case that there is a 1–1 correspondence between these partitions

$$\wedge / \circ (\text{dom } - = (-)^{-1} \circ \text{rng } -)^* \circ \diamond' / \circ \mathcal{P} j \circ \text{rng } \varphi$$

where  $j: e \mapsto \langle e \rangle$  is the injection operator that takes elements of a set into singleton sequences,  $\diamond'$  is the unique concatenation operator of sequences intro-

## THE FILE SYSTEM

duced in Chapter 3, and the overall reduction is with respect to the logical and operator.

These formalisms were then wrapped up as a single invariant:

$$\begin{aligned}
 \text{inv-}FS_0: FS_0 &\longrightarrow \mathbf{B} \\
 \text{inv-}FS_0(\varphi) &\triangleq \\
 &\Delta / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi = \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi \\
 &\wedge |\oplus / \mathcal{P}_J \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi| = |\cup / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi| \\
 &\wedge \Delta / \circ \mathcal{P} \text{ dom} \circ \text{rng} \varphi = \cup / \circ \mathcal{P} \text{ dom} \circ \text{rng} \varphi \\
 &\wedge |\oplus / \mathcal{P}_J \circ \mathcal{P} \text{ dom} \circ \text{rng} \varphi| = |\cup / \circ \mathcal{P} \text{ dom} \circ \text{rng} \varphi| \\
 &\wedge \wedge / \circ (\text{dom} - = (-)^{-1} \circ \text{rng} -)^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng} \varphi
 \end{aligned}$$

A word of explanation on the last line of the invariant is probably in order. First, for any file,  $\mu$ , we require that the inverse image of its range be equal to its domain:

$$Q(\mu) \triangleq \text{dom} \mu = \mu^{-1} \circ \text{rng} \mu$$

where, for convenience I introduce the predicate name,  $Q$ . Next, I wish to take a set of files,  $\text{rng} \varphi = \{\mu_1, \dots, \mu_k\}$ , and transform it into a sequence of files,  $\langle \mu_1, \dots, \mu_k \rangle$ . The standard way in which this may be achieved is by applying the composite operator  $\wedge / \circ \mathcal{P}_J$ . However, when we come to consider proofs of correctness with respect to this invariant, we will find that we need to consider expressions of the form,  $\wedge / \circ \mathcal{P}_J(S_j \cup S_k)$ , where  $S_j$  and  $S_k$  are sets. A simple counter-example will show that, in general

$$\wedge / \circ \mathcal{P}_J(S_j \cup S_k) \neq \wedge / \circ \mathcal{P}_J(S_j) \wedge \wedge / \circ \mathcal{P}_J(S_k)$$

Only by working within the monoid of unique sequences under the particular operator,  $\diamond'$ , will the operator calculus proof go through. Having formed the sequence of files, I may then apply the  $Q$  predicate to every element in the sequence

$$Q^* \langle \mu_1, \dots, \mu_k \rangle$$

This effectively gives me a sequence of boolean values, all of which must be true. Conceptually, this is the standard way in which to model a bit vector. The final step is the reduction of this sequence with respect to the logical and operator. In one sense, the last line of the invariant is very similar to a one line APL program, a similarity which should not cause any surprise, since both APL and the Irish School of the *VDM* are operator based. Having elaborated to some extent on my chosen

example with which I shall ‘test’ the specification, it is time to consider the latter in some detail.

### 2.3. Syntactic Domains

We have established the domain of discourse, i.e., presented the semantic domains. One then turns one’s attention to the presentation of the syntactic domains. There are five commands, i.e., operations, to be performed:

$$Cmd_0 = Crea_0 \mid Eras_0 \mid Put_0 \mid Get_0 \mid Del_0$$

$$Crea_0 :: Fn \quad \text{— create a new file with no pages}$$

$$Eras_0 :: Fn \quad \text{— erase an existing file}$$

$$Put_0 :: Fn \times Pn \times PG \quad \text{— put a page into a file}$$

$$Get_0 :: Fn \times Pn \quad \text{— get a page from a file}$$

$$Del_0 :: Fn \times Pn \quad \text{— delete a page from a file}$$

In practice, when developing real specifications, one does not proceed in such a neat fashion. Rather, semantic domains, syntactic domains *and* semantic functions emerge from discussion and analysis in a rather haphazard manner. We are effectively studying a ‘polished’ result.

### 2.4. Semantic Functions

#### 2.4.1. The Crea Command

This is the command that is ‘creative’ in the sense that from it everything else arises. Note that I have chosen to use the structure of  $FS_0$  in place of the name  $FS_0$  in the signature for a reason which shall be made apparent. But I have not replaced the name *FILE* with its structure since the latter is not particularly relevant for this command. The specification of  $Crea_0$  is then:

$$Crea_0: Fn \longrightarrow (Fn \xrightarrow{m} FILE) \longrightarrow (Fn \xrightarrow{m} FILE)$$

$$\begin{aligned} Crea_0 \llbracket fn \rrbracket \varphi &\triangleq \\ \neg \chi \llbracket fn \rrbracket \varphi & \\ \rightarrow \varphi \cup [fn \mapsto \theta] & \\ \rightarrow \perp & \end{aligned}$$

The use of the map extend operator in the expression,  $\varphi \cup [fn \mapsto \theta]$ , is defined only because of the guard,  $\neg \chi \llbracket fn \rrbracket \varphi$ , i.e.,  $fn \notin dom \varphi$ . An alternative formulation using

## THE FILE SYSTEM

the map symmetric difference operator is,  $\varphi \Delta [fn \mapsto \theta]$ .

Now let us verify that this specification does indeed preserve an invariant. One of the simplest invariants is that for all  $\varphi \in FS_0$ , the cardinality of the range can never exceed that of the domain, i.e., that  $|rng \varphi| \leq |dom \varphi|$  should be preserved. This may seem too trivial and obvious to worth bothering to prove. However, it is just such proofs that serve as a ‘mnemotechnic system’ to assist in rembering basic facts (Pólya [1945] 1957, 218). I formulate the problem as hypothesis, followed by conclusion, i.e., ‘If the cardinality of the range of a file system does not exceed that of its domain (hypothesis), then after invocation of a create command, the cardinality of the range of the new file system will not exceed that of the cardinality of the domain of the new file system (conclusion)’. More formally:

HYPOTHESIS C.1.  $|rng \varphi| \leq |dom \varphi|$ .

CONCLUSION C.1.  $|rng(\varphi \cup [fn \mapsto \theta])| \leq |dom(\varphi \cup [fn \mapsto \theta])|$ .

Those familiar with the English School of the *VDM* will recognise here the usual sort of pre- and post-conditions. The proof itself is quite elementary.

*Proof:*

$$\begin{aligned} |rng(\varphi \cup [fn \mapsto \theta])| &= \begin{cases} |rng \varphi|, & \text{if } \theta \in rng \varphi; \\ |rng \varphi| + 1, & \text{otherwise.} \end{cases} \\ |dom(\varphi \cup [fn \mapsto \theta])| &= |dom \varphi| + 1 \end{aligned}$$

and, by the hypothesis,

$$|rng \varphi| < |rng \varphi| + 1 \leq |dom \varphi| + 1$$

giving the desired result. Next I verify the correctness of the  $Crea_0$  command with respect to the new invariant. I only give the proofs for two subparts. The others are similarly constructed.

HYPOTHESIS C.2.  $\Delta / \circ \mathcal{P} rng \circ rng \varphi = \cup / \circ \mathcal{P} rng \circ rng \varphi$

CONCLUSION C.2.  $\Delta / \circ \mathcal{P} rng \circ rng(\varphi \cup [fn \mapsto \theta]) = \cup / \circ \mathcal{P} rng \circ rng(\varphi \cup [fn \mapsto \theta])$

*Proof:* Simplifying the right-hand-side gives

$$\begin{aligned}
& \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng}(\varphi \cup [fn \mapsto \theta]) \\
&= \cup / \circ \mathcal{P} \text{ rng}(\text{rng} \varphi \cup \text{rng} [fn \mapsto \theta]) \\
&= \cup / \circ \mathcal{P} \text{ rng}(\text{rng} \varphi \cup \{\theta\}) \\
&= \cup / (\mathcal{P} \text{ rng} \circ \text{rng} \varphi \cup \mathcal{P} \text{ rng} \{\theta\}) \\
&= \cup / (\mathcal{P} \text{ rng} \circ \text{rng} \varphi \cup \{\text{rng} \theta\}) \\
&= \cup / (\mathcal{P} \text{ rng} \circ \text{rng} \varphi \cup \{\emptyset\}) \\
&= \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi \cup \cup / \{\emptyset\} \\
&= \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi \cup \emptyset \\
&= \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi
\end{aligned}$$

Similarly, the left-hand-side reduces to

$$\begin{aligned}
& \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng}(\varphi \Delta [fn \mapsto \theta]) \\
&= \dots \\
&= \Delta / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi
\end{aligned}$$

where the choice of the alternate form of the  $\text{Crea}_0$  command is employed to facilitate the application of the calculus and the result is established. Now we will consider the ‘final line’ of the invariant.

$$\text{HYPOTHESIS C.3. } \wedge / \circ (\text{dom} - = (-)^{-1} \circ \text{rng} -)^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng} \varphi$$

$$\text{CONCLUSION C.3. } \wedge / \circ (\text{dom} - = (-)^{-1} \circ \text{rng} -)^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng}(\varphi \cup [fn \mapsto \theta])$$

*Proof:* For convenience, I will use the  $Q$  predicate introduced earlier, and be a little more brief. Without loss of generality, I may assume that  $\theta \notin \text{rng} \varphi$ , otherwise the expression involving the  $\diamond'$  operator will be invalid:

$$\begin{aligned}
& \wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng}(\varphi \cup [fn \mapsto \theta]) \\
&= \wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J(\text{rng} \varphi \cup \{\theta\}) \\
&= \wedge / \circ Q^* \circ \diamond' / (\mathcal{P}_J \circ \text{rng} \varphi \cup \{\langle \theta \rangle\}) \\
&= \wedge / \circ Q^*(\diamond' / \circ \mathcal{P}_J \circ \text{rng} \varphi \diamond' \langle \theta \rangle), \text{ -- I need the } \diamond' \text{ for this step} \\
&= \wedge / (Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng} \varphi \diamond' \langle Q(\theta) \rangle) \\
&= \wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng} \varphi \wedge Q(\theta)
\end{aligned}$$

and since  $Q(\theta) = (\text{dom} \theta = (\theta)^{-1} \circ \text{rng} \theta) = \text{true}$ , which may readily be verified, the conclusion is established.

But the very act of proving basic facts is not all that should be done with such a specification. Again, as in the case of the invariant, one should consider the method

## THE FILE SYSTEM

of specialisation and generalisation. One might suppose that the abstract syntax of the command models a ‘user operation’—that of creating a file. In concrete syntax, both a create command and a file name would be provided by the user. But let us consider a different syntactic model— one in which the system chooses file names. Then, for the abstract syntax

$$Crea_0 :: nil$$

we may write the specification in the form

$$Crea_0: nil \longrightarrow (Fn \xrightarrow{m} FILE) \longrightarrow (Fn \xrightarrow{m} FILE)$$

$$Crea_0 \llbracket \varphi \rrbracket \triangleq$$

let  $fn \in (dom \varphi \leftarrow Fn)$  in

$$\varphi \cup [fn \mapsto \theta]$$

Now the reason for the original signature is clear. I did not wish a reference to  $Fn$  to appear like a *deus ex machina* within the specification of  $Crea_0$ . Although the point seems a little forced here, the issue does arise frequently in specifications and will be further noted subsequently in this Appendix. Note moreover that the guard has disappeared as well as the ‘do not care’ expression  $\perp$ . Another very fruitful path to follow is generalisation. Successive calls to the create command may be modelled precisely by

$$Crea_0: \mathcal{P}Fn \longrightarrow (Fn \xrightarrow{m} FILE) \longrightarrow (Fn \xrightarrow{m} FILE)$$

$$Crea_0 \llbracket fns \rrbracket \varphi \triangleq$$

$$fns \neq \emptyset$$

$$\rightarrow Crea_0 \llbracket \{fn\} \leftarrow fns \rrbracket \circ Crea_0 \llbracket fn \rrbracket \varphi$$

$$\rightarrow \varphi$$

This specification is it not exactly equivalent to the original in so far that  $\perp$  has been replaced with  $\varphi$ . Of course, the same specification may be stated more succinctly using the powerset functor, and reduction with respect to composition:

$$(\circ / \circ \mathcal{P}Crea_0 \llbracket - \rrbracket) \varphi$$

Recall that it is precisely the counterpart of this generalisation that will fail in the subsequent reification process, a point which was raised in Chapter 5.

## 2.4.2. The Erase Command

The erase command is, in a sense, the inverse of the create command:

$$Eras_0: Fn \longrightarrow (Fn \xrightarrow{m} FILE) \longrightarrow (Fn \xrightarrow{m} FILE)$$

$$\begin{aligned} Eras_0 \llbracket fn \rrbracket \varphi &\triangleq \\ \chi \llbracket fn \rrbracket \varphi & \\ \rightarrow \{fn\} \triangleleft \varphi & \\ \rightarrow \perp & \end{aligned}$$

Since we are using the guard,  $\chi \llbracket fn \rrbracket \varphi$ , then an equivalent expression using the map symmetric difference operator is given by,  $\varphi \Delta [fn \mapsto \dots]$ , where it is of no interest what the image of  $fn$  under  $\varphi$  is.

Let us now check that  $Eras_0$  is correct with respect to the two subparts of the invariant that we used above for  $Crea_0$ .

$$\text{HYPOTHESIS C.4. } \Delta / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi = \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi$$

$$\text{CONCLUSION C.4. } \Delta / \circ \mathcal{P} \text{ rng} \circ \text{rng}(\{fn\} \triangleleft \varphi) = \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng}(\{fn\} \triangleleft \varphi)$$

*Proof:* In the Irish School of the VDM doing proofs is essential, not only for the usual purpose of ensuring correctness, but also to obtain insight into the problem domain. As has been frequently stated, it is in the course of doing such proofs that many interesting properties of the *Meta-IV* domains and operators emerge. The problem in hand is typical. Here, one needs to analyse the distribution of the *rng* operator over the map remove operator. Specifically

$$\text{rng}(\{fn\} \triangleleft \varphi) = \begin{cases} \{\varphi(fn)\} \triangleleft \text{rng} \varphi, & \text{if } |\varphi^{-1} \circ \varphi(fn)| = 1 \\ \text{rng} \varphi, & \text{otherwise} \end{cases}$$

In other words, if  $\varphi$  is 1-1 with respect to  $fn$ , i.e.,  $|\varphi^{-1} \circ \varphi(fn)| = 1$ , then removal of  $\varphi$  with respect to  $fn$ , is the same as removing the file,  $\varphi(fn)$ , from the image set  $\text{rng} \varphi$ . In the many-to-one case, i.e., where there is some other  $fn' \in \text{dom} \varphi$ , or, equivalently,  $fn' \in \varphi^{-1} \circ \varphi(fn)$ , then the image set is unchanged.

We may assume, without loss of generality, that  $|\varphi^{-1} \circ \varphi(fn)| = 1$ , otherwise there is nothing to prove. Reducing the right-hand-side gives

## THE FILE SYSTEM

$$\begin{aligned}
& \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng}(\{fn\} \triangleleft \varphi) \\
& = \cup / \circ \mathcal{P} \text{ rng}(\{\varphi(fn)\} \triangleleft \text{rng } \varphi) \\
& = \cup / (\{\text{rng } \varphi(fn)\} \triangleleft \mathcal{P} \text{ rng} \circ \text{rng } \varphi) \\
& = \{\text{rng } \varphi(fn)\} \triangleleft \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng } \varphi
\end{aligned}$$

Similarly, the left-hand-side reduces to

$$\Delta / \circ \mathcal{P} \text{ rng} \circ \text{rng}(\{fn\} \triangleleft \varphi) = \{\text{rng } \varphi(fn)\} \triangleleft \Delta / \circ \mathcal{P} \text{ rng} \circ \text{rng } \varphi$$

giving the result. Let us now look at the other clause to be checked, where I have already abbreviated the expressions, using the predicate  $Q$ .

HYPOTHESIS C.5.  $\wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng } \varphi$

CONCLUSION C.5.  $\wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng}(\{fn\} \triangleleft \varphi)$

*Proof:* Again we may assume, without loss of generality, that  $|\varphi^{-1} \circ \varphi(fn)| = 1$ .

$$\begin{aligned}
& \wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng}(\{fn\} \triangleleft \varphi) \\
& = \wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J(\{\varphi(fn)\} \triangleleft \text{rng } \varphi) \\
& = \wedge / \circ Q^* \circ \diamond' / (\{\varphi(fn)\} \triangleleft \mathcal{P}_J \circ \text{rng } \varphi) \\
& = \wedge / \circ Q^*(\{\varphi(fn)\} \triangleleft \diamond' / \circ \mathcal{P}_J \circ \text{rng } \varphi) \\
& = \wedge / (\{Q(\varphi(fn))\} \triangleleft Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng } \varphi)
\end{aligned}$$

By hypothesis, the last expression reduces to  $\wedge / \Lambda$ , which we may take to be true *in this case*.

### 2.4.3. The Put Command

Whereas both the create and erase commands operate at the ‘front end’ of the file system, the put command operates on the ‘back end’, i.e., on individual files. From a conceptual model point of view, there *is* an important distinction between front and back end commands, a distinction which is not clear from the initial specification of the command set. Moreover, this distinction has an important rôle to play in the way that we will view proofs with respect to the file system. Since we are dealing with a back end command, I use the structure of a file instead of the name, *FILE*.

$$Put_0: Fn \times Pn \times PG \longrightarrow (Fn \xrightarrow{m} (Pn \xrightarrow{m} PG)) \longrightarrow (Fn \xrightarrow{m} (Pn \xrightarrow{m} PG))$$

$$\begin{aligned} Put_0[[fn, pn, pg]]\varphi &\triangleq \\ \chi[[fn]]\varphi \wedge \chi[[pn]]\varphi(fn) & \\ \rightarrow \varphi + [fn \mapsto \varphi(fn) + [pn \mapsto pg]] & \\ \chi[[fn]]\varphi \wedge \neg\chi[[pn]]\varphi(fn) & \\ \rightarrow \varphi + [fn \mapsto \varphi(fn) \cup [pn \mapsto pg]] & \\ \rightarrow \perp & \end{aligned}$$

Note that in the original text the definition is given simply as

$$\varphi + [fn \mapsto \varphi(fn) + [pn \mapsto pg]]$$

where advantage is taken of the fact that the override operator overloads the extend operator. However, this confusion is quickly brought to light when the next stage in reification is applied. Although such an overloading may appear to be convenient, I am inclined to insist that, wherever possible, one should use the override operator in a strict manner. Again the guards permit us to use symmetric difference in place of extend to give  $\varphi + [fn \mapsto \varphi(fn) \Delta [pn \mapsto pg]]$ .

To demonstrate, in the absence of the new invariant, that it is possible to construct two files  $\mu_1$  and  $\mu_2$  such that distinct page names  $pn_1 \in dom \mu_1$  and  $pn_2 \in dom \mu_2$  both map to the same page  $pg$ , consider the following example:

$$\begin{aligned} Put_0[[fn_2, pn_2, pg]][fn_1 \mapsto [pn_1 \mapsto pg], fn_2 \mapsto \theta] & \\ = [fn_1 \mapsto [pn_1 \mapsto pg], fn_2 \mapsto \theta] + [fn_2 \mapsto \theta \cup [pn_2 \mapsto pg]] & \\ = [fn_1 \mapsto [pn_1 \mapsto pg], fn_2 \mapsto \theta] + [fn_2 \mapsto [pn_2 \mapsto pg]] & \\ = [fn_1 \mapsto [pn_1 \mapsto pg], fn_2 \mapsto [pn_2 \mapsto pg]] & \end{aligned}$$

Consequently, my ‘strange’ counter-example to ‘test’ the specification may be constructed within the model. In other words, even though the original invariant was trivially true, there are no subsequent hidden constraints, such as might be provided by a pre-condition, that rules out the example. In order to constrain the  $Put_0$  command such that it does not violate the new invariant, we need the pre-condition (see Chapter 5) which asserts that the new page,  $pg$ , must not belong to any of the *other* files in the file system:

$$\begin{aligned} pre\text{-}Put_0: Fn \times Pn \times PG \longrightarrow (Fn \xrightarrow{m} (Pn \xrightarrow{m} PG)) \longrightarrow \mathbf{B} & \\ pre\text{-}Put_0[[fn, pn, pg]]\varphi \triangleq \chi[[fn]]\varphi \wedge pg \notin \cup / \circ \mathcal{P} \text{rng}(\{\varphi(fn)\}) \triangleleft \text{rng } \varphi & \end{aligned}$$

## THE FILE SYSTEM

From our proof that  $Eras_0$  is correct with respect to ‘first line’ of the invariant, we constructed the equality

$$\cup / \circ \mathcal{P} \text{rng}(\{\varphi(fn)\}) \triangleleft \text{rng } \varphi = \{\text{rng } \varphi(fn)\} \triangleleft \cup / \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi$$

and the corresponding page restriction condition may be given in the form

$$pg \notin (\{\text{rng } \varphi(fn)\} \triangleleft \cup / \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi)$$

Strictly speaking, I am only considering the special case,  $|\varphi^{-1} \circ \varphi(fn)| = 1$ , in using this result. For a complete treatment of the semantics of the  $Put_0$  command, we must also deal with file aliases, i.e., all file names in the set  $\varphi^{-1} \circ \varphi(fn)$ . Naturally none of this is mentioned in the original published text.

Turning our attention to verifying that  $Put_0$  is correct with respect to the new invariant, we will again just consider the same two subparts. For simplicity, I will focus only on the case

$$Put_0[[fn, pn, pg]]\varphi = \varphi + [fn \mapsto \varphi(fn) + [pn \mapsto pg]]$$

.

HYPOTHESIS C.6.  $\Delta / \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi = \cup / \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi$

CONCLUSION C.6.  $\Delta / \circ \mathcal{P} \text{rng} \circ \text{rng}(\varphi + [fn \mapsto \varphi(fn) + [pn \mapsto pg]]) = \cup / \circ \mathcal{P} \text{rng} \circ \text{rng}(\varphi + [fn \mapsto \varphi(fn) + [pn \mapsto pg]])$

*Proof:* Considering the right-hand-side first, and applying the rule that  $\text{rng}(\mu_j + \mu_k) \subseteq \text{rng } \mu_j \cup \text{rng } \mu_k$ , gives:

$$\begin{aligned} & \cup / \circ \mathcal{P} \text{rng} \circ \text{rng}(\varphi + [fn \mapsto \varphi(fn) + [pn \mapsto pg]]) \\ & \subseteq \cup / \circ \mathcal{P} \text{rng}(\text{rng } \varphi \cup \text{rng } [fn \mapsto \varphi(fn) + [pn \mapsto pg]]) \\ & = \cup / \circ \mathcal{P} \text{rng}(\text{rng } \varphi \cup \{\varphi(fn) + [pn \mapsto pg]\}) \\ & = \cup / (\mathcal{P} \text{rng} \circ \text{rng } \varphi \cup \mathcal{P} \text{rng } \{\varphi(fn) + [pn \mapsto pg]\}) \\ & = \cup / (\mathcal{P} \text{rng} \circ \text{rng } \varphi \cup \{\text{rng}(\varphi(fn) + [pn \mapsto pg])\}) \\ & \subseteq \cup / (\mathcal{P} \text{rng} \circ \text{rng } \varphi \cup \{\text{rng } \varphi(fn) \cup \text{rng } [pn \mapsto pg]\}) \\ & = \cup / (\mathcal{P} \text{rng} \circ \text{rng } \varphi \cup \{\text{rng } \varphi(fn) \cup \{pg\}\}) \\ & = \cup / \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi \cup \cup / \{\text{rng } \varphi(fn) \cup \{pg\}\}) \\ & = \cup / \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi \cup (\text{rng } \varphi(fn) \cup \{pg\}) \\ & = (\cup / \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi \cup \text{rng } \varphi(fn)) \cup \{pg\} \\ & = \cup / \circ \mathcal{P} \text{rng} \circ \text{rng } \varphi \cup \{pg\} \end{aligned}$$

which reduces to  $\cup / \circ \mathcal{P} \text{rng} \circ \text{rng} \varphi$ , if the page  $pg$  already belongs to the file  $\varphi(fn)$ . Now there is the matter of demonstrating that the subset inclusion operators may be strengthened to strict equalities. Consider the first inclusion operator. We wish to show that

$$\text{rng}(\varphi + [fn \mapsto \varphi(fn) + [pn \mapsto pg]]) = \text{rng} \varphi \cup \text{rng} [fn \mapsto \varphi(fn) + [pn \mapsto pg]]$$

This is only possible if none of the elements in the set  $\text{rng} [fn \mapsto \varphi(fn) + [pn \mapsto pg]]$  belongs to  $\text{rng} \varphi$ , i.e., none of the elements in  $\{\varphi(fn) + [pn \mapsto pg]\}$  belongs to  $\text{rng} \varphi$ . Suppose to the contrary that there is indeed some  $fn'$  such that

$$[fn' \mapsto \varphi(fn) + [pn \mapsto pg]] \in \varphi$$

But this would violate the added pre-condition of the  $Put_0$  command that the page,  $pg$ , was not in any of the other files of the file system,  $pg \notin \cup / \circ \mathcal{P} \text{rng}(\{\varphi(fn)\} \leftarrow \text{rng} \varphi)$ . Consequently, we do have strict equality. But what about the second inclusion operator? Again strict equality is justified if and only if there is no element of  $\text{rng} [pn \mapsto pg] = \{pg\}$  which is in the set  $\text{rng} \varphi(fn)$ , i.e., if and only if  $pg \notin \text{rng} \varphi(fn)$ . But we *do* permit the possibility that  $pg \in \text{rng} \varphi(fn)$ . In the former case, we have strict equality and the conclusion follows. To resolve the latter case, we may proceed in the usual manner of rewriting the override operator in terms of the extend operator:

$$\begin{aligned} & \cup / \circ \mathcal{P} \text{rng} \circ \text{rng}(\varphi + [fn \mapsto \varphi(fn) + [pn \mapsto pg]]) \\ &= \cup / (\mathcal{P} \text{rng} \circ \text{rng} \varphi \cup \{\text{rng}(\varphi(fn) + [pn \mapsto pg])\}) \\ &= \cup / (\mathcal{P} \text{rng} \circ \text{rng} \varphi \cup \{\text{rng}(\{pn\} \leftarrow \varphi(fn)) \cup [pn \mapsto pg]\}) \\ &= \cup / (\mathcal{P} \text{rng} \circ \text{rng} \varphi \cup \{\text{rng}(\{pn\} \leftarrow \varphi(fn)) \cup \text{rng} [pn \mapsto pg]\}) \\ &= \cup / (\mathcal{P} \text{rng} \circ \text{rng} \varphi \cup \{\{(\varphi(fn))(pn)\} \leftarrow \text{rng} \varphi(fn) \cup \{pg\}\}) \\ &= \cup / (\mathcal{P} \text{rng} \circ \text{rng} \varphi \cup \{\{(\varphi(fn))(pn)\} \leftarrow \text{rng} \varphi(fn)\}) \\ &= \cup / \circ \mathcal{P} \text{rng} \circ \text{rng} \varphi \cup \{(\varphi(fn))(pn)\} \leftarrow \text{rng} \varphi(fn) \\ &= \{(\varphi(fn))(pn)\} \leftarrow (\cup / \circ \mathcal{P} \text{rng} \circ \text{rng} \varphi \cup \text{rng} \varphi(fn)) \\ &= \{(\varphi(fn))(pn)\} \leftarrow \cup / \circ \mathcal{P} \text{rng} \circ \text{rng} \varphi \end{aligned}$$

and we have no difficulty in establishing the result, well *almost* no difficulty. The distribution of the  $\text{rng}$  operator over map removal also occurs in the body of this proof. The detailed arguments and justification are omitted. For the second subpart of the invariant, I may be more brief in my demonstration.

# THE FILE SYSTEM

HYPOTHESIS C.7.  $\wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng } \varphi$

CONCLUSION C.7.  $\wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng}(\varphi + [fn \mapsto \varphi(fn) + [pn \mapsto pg]])$

*Proof:*

$$\begin{aligned}
 & \wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng}(\varphi + [fn \mapsto \varphi(fn) + [pn \mapsto pg]]) \\
 &= \wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J(\text{rng } \varphi \cup \{\varphi(fn) + [pn \mapsto pg]\}) \\
 &= \wedge / \circ Q^* \circ \diamond' / (\mathcal{P}_J \circ \text{rng } \varphi \cup \{\langle \varphi(fn) + [pn \mapsto pg] \rangle\}) \\
 &= \wedge / \circ Q^*(\diamond' / \circ \mathcal{P}_J \circ \text{rng } \varphi \diamond' \langle \varphi(fn) + [pn \mapsto pg] \rangle) \\
 &= \wedge / (Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng } \varphi \diamond' \langle Q(\varphi(fn) + [pn \mapsto pg]) \rangle) \\
 &= \wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng } \varphi \wedge Q(\varphi(fn) + [pn \mapsto pg])
 \end{aligned}$$

and the conclusion depends on the truth value of  $Q(\varphi(fn) + [pn \mapsto pg])$ . This may readily be established. Let  $\mu = \varphi(fn) + [pn \mapsto pg]$ . Then

$$\begin{aligned}
 \text{rng } \mu &= \text{rng}(\varphi(fn) + [pn \mapsto pg]) \\
 &= \text{rng}(\{pn\} \triangleleft \varphi(fn) \cup [pn \mapsto pg]) \\
 &= \{(\varphi(fn))(pn)\} \triangleleft \text{rng } \varphi(fn) \cup \{pg\}
 \end{aligned}$$

But

$$\begin{aligned}
 \mu^{-1} \circ \text{rng } \mu &= \mu^{-1}(\{(\varphi(fn))(pn)\} \triangleleft \text{rng } \varphi(fn) \cup \{pg\}) \\
 &= \mu^{-1}(\{(\varphi(fn))(pn)\} \triangleleft \text{rng } \varphi(fn)) \cup \mu^{-1}\{pg\} \\
 &= \{pn\} \triangleleft \text{dom } \varphi(fn) \cup \{pn\} \\
 &= \text{dom } \varphi(fn)
 \end{aligned}$$

and, of course

$$\begin{aligned}
 \text{dom } \mu &= \text{dom}(\varphi(fn) + [pn \mapsto pg]) \\
 &= \text{dom } \varphi
 \end{aligned}$$

and the result is established, at least for the case  $|\mu^{-1} \circ \mu(pn)| = 1$ , which governs the distributivity of the *rng* operator over the map removal operator. The establishment of the other case is similar.

## 2.4.4. The Get Command

This is a basic lookup command. It certainly seems innocuous. However, it is the only command that does not cause a file system transformation. Consequently, it may be used to establish a property of the file system with respect to a retrieve function, a point brought out in Appendix B. In fact, it did play a key rôle in untangling some of the complexities in the sequence of published reifications.

$$Get_0: Fn \times Pn \longrightarrow (Fn \xrightarrow{m} (Pn \xrightarrow{m} PG)) \longrightarrow PG$$

$$\begin{aligned} Get_0[[fn, pn]]\varphi &\triangleq \\ \chi[[fn]]\varphi \wedge \chi[[pn]]\varphi(fn) & \\ \rightarrow (\varphi(fn))(pn) & \\ \rightarrow \perp & \end{aligned}$$

Naturally, there is nothing to verify with respect to the invariant.

#### 2.4.5. The Del Command

This is essentially the inverse of the put command.

$$Del_0: Fn \times Pn \longrightarrow (Fn \xrightarrow{m} (Pn \xrightarrow{m} PG)) \longrightarrow (Fn \xrightarrow{m} (Pn \xrightarrow{m} PG))$$

$$\begin{aligned} Del_0[[fn, pn]]\varphi &\triangleq \\ \chi[[fn]]\varphi \wedge \chi[[pn]]\varphi(fn) & \\ \rightarrow \varphi + [fn \mapsto \{pn\} \leftarrow \varphi(fn)] & \\ \rightarrow \perp & \end{aligned}$$

The expression may be rewritten  $\varphi + [fn \mapsto \varphi(fn) \Delta [pn \mapsto \dots]]$ .

The verification that  $Del_0$  is correct with respect to the new invariant is straightforward. In carrying out the proof, we will be more interested in the kind of expressions that need resolution in the operator calculus, than in the mere result itself, which is obvious. Note once more that we have to deal with the properties of the *rng* operator with respect to map override and map removal. I present the formal details without comment. Checking the proof will disclose the assumptions with respect to which the proof is valid. For convenience, the ‘suspect’ portions are marked with the symbol  $\nabla$ . First, we will consider

$$\text{HYPOTHESIS C.8. } \Delta / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi = \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi$$

$$\text{CONCLUSION C.8. } \Delta / \circ \mathcal{P} \text{ rng} \circ \text{rng}(\varphi + [fn \mapsto \{pn\} \leftarrow \varphi(fn)]) = \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng}(\varphi + [fn \mapsto \{pn\} \leftarrow \varphi(fn)])$$

# THE FILE SYSTEM

*Proof:* The right-hand-side gives

$$\begin{aligned}
& \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng}(\varphi + [fn \mapsto \{pn\} \triangleleft \varphi(fn)]) \\
&= \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng}(\{fn\} \triangleleft \varphi \cup [fn \mapsto \{pn\} \triangleleft \varphi(fn)]) \\
&= \cup / \circ \mathcal{P} \text{ rng}(\text{rng}(\{fn\} \triangleleft \varphi) \cup \text{rng} [fn \mapsto \{pn\} \triangleleft \varphi(fn)]) \\
&= \cup / \circ \mathcal{P} \text{ rng}(\{\varphi(fn)\} \triangleleft \text{rng} \varphi \cup \{\{pn\} \triangleleft \varphi(fn)\}) \text{ -- } \nabla \\
&= \cup / (\mathcal{P} \text{ rng}(\{\varphi(fn)\} \triangleleft \text{rng} \varphi) \cup \mathcal{P} \text{ rng} \{\{pn\} \triangleleft \varphi(fn)\}) \\
&= \cup / (\text{rng} \varphi(fn) \triangleleft \mathcal{P} \text{ rng} \circ \text{rng} \varphi \cup \{\text{rng}(\{pn\} \triangleleft \varphi(fn))\}) \\
&= \cup / (\text{rng} \varphi(fn) \triangleleft \mathcal{P} \text{ rng} \circ \text{rng} \varphi \cup \{\{(\varphi(fn))(pn)\} \triangleleft \text{rng} \varphi(fn)\}) \text{ -- } \nabla \\
&= \text{rng} \varphi(fn) \triangleleft \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi \cup \{(\varphi(fn))(pn)\} \triangleleft \text{rng} \varphi(fn) \\
&= \{(\varphi(fn))(pn)\} \triangleleft (\text{rng} \varphi(fn) \triangleleft \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi \cup \text{rng} \varphi(fn)) \\
&= \{(\varphi(fn))(pn)\} \triangleleft \cup / \circ \mathcal{P} \text{ rng} \circ \text{rng} \varphi
\end{aligned}$$

The reduction of the left-hand-side will give a similar result. Finally we take a look at the other subpart of the invariant.

HYPOTHESIS C.9.  $\wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_j \circ \text{rng} \varphi$

CONCLUSION C.9.  $\wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_j \circ \text{rng}(\varphi + [fn \mapsto \{pn\} \triangleleft \varphi(fn)])$

*Proof:*

$$\begin{aligned}
& \wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_j \circ \text{rng}(\varphi + [fn \mapsto \{pn\} \triangleleft \varphi(fn)]) \\
&= \wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_j \circ \text{rng}(\{fn\} \triangleleft \varphi \cup [fn \mapsto \{pn\} \triangleleft \varphi(fn)]) \\
&= \wedge / \circ Q^* \circ \diamond' / \circ \mathcal{P}_j(\{\varphi(fn)\} \triangleleft \text{rng} \varphi \cup \{\{pn\} \triangleleft \varphi(fn)\}) \text{ -- } \nabla \\
&= \wedge / \circ Q^* \circ \diamond' / (\{\varphi(fn)\} \triangleleft \mathcal{P}_j \circ \text{rng} \varphi \cup \{\langle \{pn\} \triangleleft \varphi(fn) \rangle\}) \\
&= \wedge / \circ Q^*(\{\varphi(fn)\} \triangleleft \diamond' / \circ \mathcal{P}_j \circ \text{rng} \varphi \diamond' \langle \{pn\} \triangleleft \varphi(fn) \rangle) \\
&= \wedge / (\{Q(\varphi(fn))\} \triangleleft Q^* \circ \diamond' / \circ \mathcal{P}_j \circ \text{rng} \varphi \diamond' \langle Q(\{pn\} \triangleleft \varphi(fn)) \rangle) \\
&= \wedge / \langle Q(\{pn\} \triangleleft \varphi(fn)) \rangle
\end{aligned}$$

where we have relied on the arguments presented in the case of the verification of the  $Eras_0$  command with respect to this part of the invariant. Hence, the result is established if  $Q(\{pn\} \triangleleft \varphi(fn)) = true$ , which is intuitively correct. The details are omitted. Recall that the check mark,  $\nabla$ , in the proof above indicates that we also need to consider the case of file aliases, i.e.,  $|\varphi^{-1} \circ \varphi(fn)| > 1$ .

## 2.5. Summary

A critical analysis of the file system case study has exposed some defects in the baseline model,  $FS_0$ , defects which have been overcome by constructing a non-trivial invariant, one which is derived as a direct consequence of trying to prove that the baseline model is indeed retrieved from the first level reification model to be presented in the next section. Given an invariant, the method of the Irish School of the *VDM* dictates that all operations must be verified with respect to it. I have not given the details of all parts of the proofs. Those which I did present led to some further interesting results, the most important of which is undoubtedly the extra pre-condition needed for the  $Put_0$  command. Indeed without such a pre-condition, it is impossible to do the proof. Other results chiefly concerned the properties of the ‘ubiquitous’ *rng* operator.

I would like to make one other observation which has great significance. Even for such a ‘simple’ formal model, the amount of effort required in carrying out the proofs of correctness is considerable. Either one does it via formal logic which is the customary style of other Schools, or one must use an operator calculus such as is employed in the Irish School. Ultimately, the proof must be reconstructed/checked by others. I firmly believe that the operator calculus approach, which is in the *mainstream* of mathematics, is the better of the two.

### 3. File System — Version 1

This is the model which provoked my deepest dissatisfaction with the *VDM* styles of other Schools and which proved to be the major testing ground of the operator calculus of the Irish School. I have already recounted at some length, in Chapter 5, of the difficulties which both the invariant and retrieve function presented me and in what manner I eventually resolved those difficulties, the construction of a new first level reification model,  $FS'_1$ , isomorphic to the old,  $FS_1$ . I give the details of the *current state* of the new model here. The emphasis is important. There is every reason to believe that further improvement of the model may be attained as a result of more work on the operator calculus.

#### 3.1. Semantic Domains

In the original text the domains  $CTLG_1$ ,  $DIRS_1$ , and  $DIR_1$  were constructed with the 1-1 map functor,  $- \xleftrightarrow[m]{-} -$ . I prefer to use the ordinary  $- \xrightarrow[m]{-} -$  functor and add in the 1-1 constraints given.

$$\begin{aligned} \varphi \in FS_1 &:: CTLG_1 \times DIRS_1 \times PGS_1 \\ \kappa \in CTLG_1 &= Fn \xrightarrow[m]{-} Dn \quad -- |dom \kappa| = |rng \kappa| \\ \tau \in DIRS_1 &= Dn \xrightarrow[m]{-} DIR_1 \quad -- |dom \tau| = |rng \tau| \\ \varpi \in PGS_1 &= Pa \xrightarrow[m]{-} PG \\ \delta \in DIR_1 &= Pn \xrightarrow[m]{-} Pa \quad -- |dom \delta| = |rng \delta| \end{aligned}$$

Such constraints are, of course, specific domain invariants. The new model is specified by the domain equations

$$\begin{aligned} \varphi \in FS'_1 &= CTLG_1 \times DIRS'_1 \\ \kappa \in CTLG_1 &= Fn \xrightarrow[m]{-} Dn \quad -- |dom \kappa| = |rng \kappa| \\ \tau \in DIRS'_1 &= Dn \xrightarrow[m]{-} (DIR_1 \times PGS'_1) \quad -- |dom \tau| = |rng \tau| \\ \delta \in DIR_1 &= Pn \xrightarrow[m]{-} Pa \quad -- |dom \delta| = |rng \delta| \\ \varpi \in PGS'_1 &= Pa \xrightarrow[m]{-} PG \end{aligned}$$

## 3.1.1. Invariant

The original invariant in the published text for  $FS_1$  is given by:

$$\begin{aligned}
& \text{inv-}FS_1: FS_1 \longrightarrow \mathbf{B} \\
& \text{inv-}FS_1(\kappa, \tau, \varpi) \triangleq \\
& \quad (\text{rng } \kappa = \text{dom } \tau) \\
& \quad \wedge \cup / \{ \text{rng } \delta \mid \delta \in \text{rng } \tau \} = \text{dom } \varpi \\
& \quad \wedge (\forall pa \in \text{dom } \varpi)(\exists! dn \in \text{dom } \tau)(pa \in \text{rng } \tau(dn))
\end{aligned}$$

That for the new model,  $FS'_1$ , is

$$\begin{aligned}
& \text{inv-}FS'_1: FS_1 \longrightarrow \mathbf{B} \\
& \text{inv-}FS'_1(\kappa, \tau) \triangleq \\
& \quad (\text{rng } \kappa = \text{dom } \tau) \\
& \quad \wedge \wedge / \circ ((\pi_1 - = \pi_2 -) \circ (\text{rng}, \text{dom}))^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng}(\tau \circ \kappa)
\end{aligned}$$

For the purpose of developing proofs, this invariant may be considered to consist of two subinvariants

$$\text{rng } \kappa = \text{dom } \tau$$

which addresses the composability of the pair of maps  $(\kappa, \tau)$  and, consequently, justifies the use of  $\tau \circ \kappa$ , and

$$\wedge / \circ ((\pi_1 - = \pi_2 -) \circ (\text{rng}, \text{dom}))^* \circ \diamond' / \circ \mathcal{P}_J \circ \text{rng}(\tau \circ \kappa)$$

which addresses the composability of maps  $(\delta, \varpi) \in \text{rng}(\tau \circ \kappa)$ . As in the case of the baseline model, I will make use of the predicate

$$Q(\delta, \varpi) \triangleq ((\pi_1 - = \pi_2 -) \circ (\text{rng}, \text{dom}))(\delta, \varpi)$$

I may not have captured all of the constraints needed for  $FS'_1$ . Such constraints should emerge as a result of verifying the commands with respect to the invariant. Of more immediate concern was the demonstration that the baseline model could be retrieved from the new model.

# THE FILE SYSTEM

## 3.2. The Retrieve Function

In the original text the retrieve function  $retr\text{-}FS_0$ , denoted here by  $\mathfrak{R}_{10}$ , is essentially expressed in the form:

$$\begin{aligned} \mathfrak{R}_{10}: FS_1 &\longrightarrow FS_0 \\ \mathfrak{R}_{10}(\kappa, \tau, \varpi) &\triangleq \\ &[fn \mapsto [pn \mapsto \varpi((\tau(\kappa(fn)))(pn)) \mid pn \in \text{dom } \tau(\kappa(fn))] \mid fn \in \text{dom } \kappa] \end{aligned}$$

The author(s) of the original text claimed “that given [certain inputs], one can devise automatic means for transforming semantic functions  $[Crea_1[[fn], \dots]]$  into  $[Crea_0[[fn], \dots]]$ ”. It is difficult to understand this on the basis of the complexity of the retrieve function given. Indeed, even in the simpler case of a manual proof that  $[Crea_1[[fn], \dots]]$  correctly implements  $[Crea_0[[fn], \dots]]$ , use of the retrieve function is difficult. The claim may or may not ultimately prove to be correct. What *is* clear is that the value of such a transformation is questionable, given the defects that I have unearthed.

For the new model I propose the retrieve function:

$$\begin{aligned} \mathfrak{R}'_{10}: FS'_1 &\longrightarrow FS_0 \\ \mathfrak{R}'_{10}(\kappa, \tau) &\triangleq (- \xrightarrow{m} \circ)(\tau \circ \kappa) \end{aligned}$$

and from Chapter 5, the bijection which mapped  $FS_1$  to  $FS'_1$  was given in the form of a retrieve function

$$\mathfrak{R}'_{11}(c, ds, ps) \triangleq (c, (ds \bowtie (- \xrightarrow{m} \Leftarrow ps) \circ (- \xrightarrow{m} \text{rng}) ds))$$

where I have resorted to the Roman letters  $c \in CTLG_1$ ,  $ds \in DIRS_1$ , and  $ps \in PGS_1$  to avoid confusion. Consequently, the original retrieve function is given by the composition

$$\mathfrak{R}_{10}(c, ds, ps) = \mathfrak{R}'_{10} \circ \mathfrak{R}'_{11}(c, ds, ps) = (- \xrightarrow{m} \circ)((ds \bowtie (- \xrightarrow{m} \Leftarrow ps) \circ (- \xrightarrow{m} \text{rng}) ds) \circ c)$$

## 3.3. Syntactic Domains

The specification of the syntactic domain for each command is the same as for version 0.

## 3.4. Semantic Functions

In the remainder of this section I confine myself to verifying that each command of model  $FS'_1$  is a correct reification of the corresponding command for  $FS_0$  with respect to  $\mathfrak{R}'_{10}$ . This establishes the validity of my new model. Then I demonstrate that each command of the original reification  $FS_1$  is correct with respect to my new model under  $\mathfrak{R}'_{11}$ . Since I have already addressed the semantics of the  $Put_0$ ,  $Put'_1$  and  $Put_1$  commands in Chapter 5, I will not repeat the material here. Wishing to be as brief as possible, I have limited my observations on the actual proofs and have taken the liberty of signalling special conditions for validity by marking ‘interesting’ lines with  $\nabla$ .

## 3.4.1. The Crea Command

$$\begin{aligned}
& Crea_1: Fn \longrightarrow FS_1 \longrightarrow FS_1 \\
& Crea_1[[fn]](c, ds, ps) \triangleq \\
& \quad \neg\chi[[fn]]c \\
& \quad \rightarrow \text{let } dn \in (dom\ ds \Leftarrow Dn) \text{ in} \\
& \quad \quad (c \cup [fn \mapsto dn], ds \cup [dn \mapsto \theta], ps) \\
& \quad \rightarrow \perp
\end{aligned}$$

$$\begin{aligned}
& Cred'_1: Fn \longrightarrow FS'_1 \longrightarrow FS'_1 \\
& Cred'_1[[fn]](\kappa, \tau) \triangleq \\
& \quad \neg\chi[[fn]]\kappa \\
& \quad \rightarrow \text{let } dn \in (dom\ \tau \Leftarrow Dn) \text{ in} \\
& \quad \quad (\kappa \cup [fn \mapsto dn], \tau \cup [dn \mapsto (\theta, \theta)]) \\
& \quad \rightarrow \perp
\end{aligned}$$

*Verification with respect to the retrieve function  $\mathfrak{R}'_{10}$*

$$\begin{array}{ccc}
\varphi & \xrightarrow{Crea_0[[fn]]} & \varphi \cup [fn \mapsto \theta] \\
\uparrow \mathfrak{R}'_{10} & & \uparrow \mathfrak{R}'_{10} \\
(\kappa, \tau) & \xrightarrow{Cred'_1[[fn]]} & (\kappa \cup [fn \mapsto dn], \tau \cup [dn \mapsto (\theta, \theta)])
\end{array}$$

# THE FILE SYSTEM

*RHS*

$$\begin{aligned}
& \mathfrak{R}'_{10} \circ Eras'_1 \llbracket fn \rrbracket (\kappa, \tau) \\
&= \mathfrak{R}'_{10} (\kappa \cup [fn \mapsto dn], \tau \cup [dn \mapsto (\theta, \theta)]) \\
&= (- \xrightarrow{m} \circ) ((\tau \cup [dn \mapsto (\theta, \theta)]) \circ (\kappa \cup [fn \mapsto dn])) \\
&= (- \xrightarrow{m} \circ) ((\kappa \circ \tau) \cup ([dn \mapsto (\theta, \theta)] \circ [fn \mapsto dn])) \text{ -- } \nabla \\
&= (- \xrightarrow{m} \circ) ((\kappa \circ \tau) \cup [fn \mapsto (\theta, \theta)]) \\
&= (- \xrightarrow{m} \circ) (\kappa \circ \tau) \cup (- \xrightarrow{m} \circ) [fn \mapsto (\theta, \theta)] \text{ -- } \nabla \\
&= (- \xrightarrow{m} \circ) (\kappa \circ \tau) \cup [fn \mapsto \theta]
\end{aligned}$$

*LHS*

$$\begin{aligned}
& Crea_0 \llbracket fn \rrbracket \circ \mathfrak{R}'_{10} (\kappa, \tau) \\
&= Crea_0 \llbracket fn \rrbracket (- \xrightarrow{m} \circ) (\kappa \circ \tau) \\
&= (- \xrightarrow{m} \circ) (\kappa \circ \tau) \cup [fn \mapsto \theta] \\
&\text{Q.E.D.}
\end{aligned}$$

A generalisation of this command is possible. Essentially we wish to consider the merge of two file systems  $(\kappa_1, \tau_1)$  and  $(\kappa_2, \tau_2)$ .

HYPOTHESIS C.10.  $(\kappa_1, \tau_1)$  and  $(\kappa_2, \tau_2)$  are pairs of composable maps.

CONCLUSION C.10.  $\mathfrak{R}'_{10}(\kappa_1 \cup \kappa_2, \tau_1 \cup \tau_2) = \mathfrak{R}'_{10}(\kappa_1, \tau_1) \cup \mathfrak{R}'_{10}(\kappa_2, \tau_2)$

*Proof:*

$$\begin{aligned}
& \mathfrak{R}'_{10}(\kappa_1 \cup \kappa_2, \tau_1 \cup \tau_2) \\
&= (- \xrightarrow{m} \circ) ((\tau_1 \cup \tau_2) \circ (\kappa_1 \cup \kappa_2)) \\
&= (- \xrightarrow{m} \circ) ((\tau_1 \circ \kappa_1) \cup (\tau_2 \circ \kappa_2)) \text{ -- } \nabla \\
&= (- \xrightarrow{m} \circ) (\tau_1 \circ \kappa_1) \cup (- \xrightarrow{m} \circ) (\tau_2 \circ \kappa_2) \text{ -- } \nabla \\
&= \mathfrak{R}'_{10}(\kappa_1, \tau_1) \cup \mathfrak{R}'_{10}(\kappa_2, \tau_2)
\end{aligned}$$

*Verification with respect to the retrieve function  $\mathfrak{R}'_{11}$*

$$\begin{array}{ccc}
(\kappa, \tau) & \xrightarrow{Cred'_1 \llbracket fn \rrbracket} & (\kappa \cup [fn \mapsto dn], \tau \cup [dn \mapsto (\theta, \theta)]) \\
\uparrow \mathfrak{R}'_{11} & & \uparrow \mathfrak{R}'_{11} \\
(c, ds, ps) & \xrightarrow{Crea_1 \llbracket fn \rrbracket} & (c \cup [fn \mapsto dn], ds \cup [dn \mapsto \theta], ps)
\end{array}$$

RHS

$$\begin{aligned}
& \mathfrak{R}'_{11} \circ \text{Crea}_1 \llbracket fn \rrbracket (c, ds, ps) \\
&= \mathfrak{R}'_{11} (c \cup [fn \mapsto dn], ds \cup [dn \mapsto \theta], ps) \\
&= (c \cup [fn \mapsto dn], (ds \cup [dn \mapsto \theta]) \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} \text{rng})(ds \cup [dn \mapsto \theta])) \\
&= (c \cup [fn \mapsto dn], ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} \text{rng}) ds \\
&\quad \cup [dn \mapsto \theta] \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} \text{rng}) [dn \mapsto \theta]) \\
&= (c \cup [fn \mapsto dn], ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} \text{rng}) ds \\
&\quad \cup [dn \mapsto \theta] \bowtie (- \xrightarrow{m} \triangleleft ps) [dn \mapsto \emptyset]) \\
&= (c \cup [fn \mapsto dn], ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} \text{rng}) ds \\
&\quad \cup [dn \mapsto \theta] \bowtie [dn \mapsto \theta]) \\
&= (c \cup [fn \mapsto dn], ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} \text{rng}) ds \cup [dn \mapsto (\theta, \theta)])
\end{aligned}$$

LHS

$$\begin{aligned}
& \text{Crea}'_1 \llbracket fn \rrbracket \circ \mathfrak{R}'_{11} (c, ds, ps) \\
&= \text{Crea}'_1 \llbracket fn \rrbracket (c, ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} \text{rng}) ds) \\
&= (c \cup [fn \mapsto dn], ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} \text{rng}) ds \cup [dn \mapsto (\theta, \theta)]) \\
&\text{Q.E.D.}
\end{aligned}$$

## 3.4.2. The Eras Command

$$\begin{aligned}
& \text{Eras}_1: Fn \longrightarrow FS_1 \longrightarrow FS_1 \\
& \text{Eras}_1 \llbracket fn \rrbracket (c, ds, ps) \triangleq \\
& \quad \chi \llbracket fn \rrbracket c \\
& \quad \rightarrow (\{fn\} \triangleleft c, \{c(fn)\} \triangleleft ds, (\text{rng } ds \circ c(fn)) \triangleleft ps) \\
& \quad \rightarrow \perp
\end{aligned}$$

$$\begin{aligned}
& \text{Eras}'_1: Fn \longrightarrow FS'_1 \longrightarrow FS'_1 \\
& \text{Eras}'_1 \llbracket fn \rrbracket (\kappa, \tau) \triangleq \\
& \quad \chi \llbracket fn \rrbracket \kappa \\
& \quad \rightarrow (\{fn\} \triangleleft \kappa, \{\kappa(fn)\} \triangleleft \tau) \\
& \quad \rightarrow \perp
\end{aligned}$$

# THE FILE SYSTEM

Verification with respect to the retrieve function  $\mathfrak{R}'_{10}$

$$\begin{array}{ccc}
 & \varphi & \xrightarrow{Eras_0 \llbracket fn \rrbracket} & \{fn\} \triangleleft \varphi \\
 & \uparrow \mathfrak{R}'_{10} & & \uparrow \mathfrak{R}'_{10} \\
 RHS & (\kappa, \tau) & \xrightarrow{Eras'_1 \llbracket fn \rrbracket} & (\{fn\} \triangleleft \kappa, \{\kappa(fn)\} \triangleleft \tau)
 \end{array}$$

$$\begin{aligned}
 & \mathfrak{R}'_{10} \circ Eras'_1 \llbracket fn \rrbracket (\kappa, \tau) \\
 &= \mathfrak{R}'_{10} (\{fn\} \triangleleft \kappa, \{\kappa(fn)\} \triangleleft \tau) \\
 &= (- \xrightarrow{m} \circ) (\{\kappa(fn)\} \triangleleft \tau) \circ (\{fn\} \triangleleft \kappa) \\
 &= (- \xrightarrow{m} \circ) (\{fn\} \triangleleft \tau \circ \kappa) \text{ --- } \nabla \\
 &= \{fn\} \triangleleft (- \xrightarrow{m} \circ) (\tau \circ \kappa) \text{ --- } \nabla
 \end{aligned}$$

LHS

$$\begin{aligned}
 & Eras_0 \llbracket fn \rrbracket \circ \mathfrak{R}'_{10} (\kappa, \tau) \\
 &= Eras_0 \llbracket fn \rrbracket (- \xrightarrow{m} \circ) (\tau \circ \kappa) \\
 &= \{fn\} \triangleleft (- \xrightarrow{m} \circ) (\tau \circ \kappa)
 \end{aligned}$$

Q.E.D.

Verification with respect to the retrieve function  $\mathfrak{R}'_{11}$

$$\begin{array}{ccc}
 & (\kappa, \tau) & \xrightarrow{Eras'_1 \llbracket fn \rrbracket} & (\{fn\} \triangleleft \kappa, \{\kappa(fn)\} \triangleleft \tau) \\
 & \uparrow \mathfrak{R}'_{11} & & \uparrow \mathfrak{R}'_{11} \\
 RHS & (c, ds, ps) & \xrightarrow{Eras_1 \llbracket fn \rrbracket} & (\{fn\} \triangleleft c, \{c(fn)\} \triangleleft ds, (rng ds \circ c(fn)) \triangleleft ps)
 \end{array}$$

$$\begin{aligned}
 & \mathfrak{R}'_{11} \circ Eras_1 \llbracket fn \rrbracket (c, ds, ps) \\
 &= \mathfrak{R}'_{11} (\{fn\} \triangleleft c, \{c(fn)\} \triangleleft ds, (rng ds \circ c(fn)) \triangleleft ps) \\
 &= (\{fn\} \triangleleft c, \\
 & \quad (\{c(fn)\} \triangleleft ds) \bowtie (- \xrightarrow{m} \triangleleft (rng ds \circ c(fn)) \triangleleft ps) \circ (- \xrightarrow{m} rng) (\{c(fn)\} \triangleleft ds)) \\
 &= \dots
 \end{aligned}$$

LHS

$$\begin{aligned}
& Eras'_1 \llbracket fn \rrbracket \circ \mathfrak{R}'_{11}(c, ds, ps) \\
&= Eras'_1 \llbracket fn \rrbracket (c, ds \bowtie (- \xrightarrow{m} ps) \circ (- \xrightarrow{m} rng) ds) \\
&= (\{fn\} \triangleleft c, \{c(fn)\} \triangleleft (ds \bowtie (- \xrightarrow{m} ps) \circ (- \xrightarrow{m} rng) ds)) \\
&= \dots
\end{aligned}$$

where the ellipsis in the proof signals that we have a small subproblem that needs to be tackled. First, I introduce some auxillary notation. Let  $dn = c(fn)$ , and  $\delta = ds \circ c(fn)$ . Then to complete the proof we need only show that

$$\begin{aligned}
& (\{dn\} \triangleleft ds) \bowtie (- \xrightarrow{m} \triangleleft (rng \delta) \triangleleft ps) \circ (- \xrightarrow{m} rng) (\{dn\} \triangleleft ds) \\
&= \{dn\} \triangleleft (ds \bowtie (- \xrightarrow{m} ps) \circ (- \xrightarrow{m} rng) ds)
\end{aligned}$$

Let  $\{dn\} \triangleleft ds = [\dots, dn_j \mapsto \delta_j, \dots]$  be the directory system in question. Then

$$\begin{aligned}
& (\{dn\} \triangleleft ds) \bowtie (- \xrightarrow{m} \triangleleft (rng \delta) \triangleleft ps) \circ (- \xrightarrow{m} rng) (\{dn\} \triangleleft ds) \\
&= [\dots, dn_j \mapsto \delta_j, \dots] \bowtie (- \xrightarrow{m} \triangleleft (rng \delta) \triangleleft ps) \circ (- \xrightarrow{m} rng) [\dots, dn_j \mapsto \delta_j, \dots] \\
&= [\dots, dn_j \mapsto \delta_j, \dots] \bowtie (- \xrightarrow{m} \triangleleft (rng \delta) \triangleleft ps) [\dots, dn_j \mapsto rng \delta_j, \dots] \\
&= [\dots, dn_j \mapsto \delta_j, \dots] \bowtie [\dots, dn_j \mapsto rng \delta_j \triangleleft (rng \delta \triangleleft ps), \dots] \\
&= [\dots, dn_j \mapsto (\delta_j, rng \delta_j \triangleleft (rng \delta \triangleleft ps)), \dots]
\end{aligned}$$

On the other hand we have

$$\begin{aligned}
& \{dn\} \triangleleft (ds \bowtie (- \xrightarrow{m} ps) \circ (- \xrightarrow{m} rng) ds) \\
&= \{dn\} \triangleleft [\dots, dn_j \mapsto (\delta_j, rng \delta_j \triangleleft ps), \dots]
\end{aligned}$$

The only way in which both the left-hand-side and the right-hand-side equations will agree is if map removal of the page system,  $ps$ , with respect to the directory,  $\delta$ , has no effect on the rest of the file system, i.e.,

$$rng \delta_j \triangleleft (rng \delta \triangleleft ps) = rng \delta_j \triangleleft ps$$

This is possible if and only if we are indeed dealing with a partitioned page system. Now it is quite clear wherein lies the complexity of the published reification model, the hidden assumption of partitioning.

One observation is in order. This particular proof demonstrates quite clearly the limits of a ‘straight’ application of the operator calculus. To complete the proof, I had to resort to the kind of argument one finds in conventional mainstream mathematics.

# THE FILE SYSTEM

## 3.4.3. The Put Command

This has already been dealt with in Chapter 5.

## 3.4.4. The Get Command

Since this is a ‘back end’ command of the file system, I will adopt the same approach to verification that I used for the put commands—four commuting squares.

$$\begin{aligned}
 &Get_1: Fn \times Pn \longrightarrow FS_1 \longrightarrow PG \\
 &Get_1\llbracket fn, pn \rrbracket(c, ds, ps) \triangleq \\
 &\quad \chi\llbracket fn \rrbracket\kappa \wedge \chi\llbracket pn \rrbracket ds \circ c(fn) \\
 &\quad \rightarrow ps\left((ds \circ c(fn))(pn)\right) \\
 &\quad \rightarrow \perp
 \end{aligned}$$

$$\begin{aligned}
 &Get_1: Fn \times Pn \longrightarrow FS_1 \longrightarrow PG \\
 &Get_1\llbracket fn, pn \rrbracket(\kappa, \tau) \triangleq \\
 &\quad \chi\llbracket fn \rrbracket\kappa \wedge \chi\llbracket pn \rrbracket\tau \circ \kappa(fn) \\
 &\quad \rightarrow \text{let } (\delta, \varpi) = \tau \circ \kappa(fn) \text{ in } \varpi \circ \delta(pn) \\
 &\quad \rightarrow \perp
 \end{aligned}$$

Verification with respect to the retrieve function  $\mathfrak{R}'_{10}$

$$\begin{array}{ccccc}
 \varphi & \xrightarrow{Lkp_0\llbracket fn \rrbracket} & \varphi(fn) = \mu & \xrightarrow{Lkp_0\llbracket pn \rrbracket} & \mu(pn) \\
 \uparrow \mathfrak{R}'_{10} & & \uparrow \mathfrak{R}_1 & & \uparrow \mathcal{I} \\
 (\kappa, \tau) & \xrightarrow{Lkp'_1\llbracket fn \rrbracket} & \tau \circ \kappa(fn) = (\delta, \varpi) & \xrightarrow{Lkp'_1\llbracket pn \rrbracket} & \varpi \circ \delta(pn)
 \end{array}$$

First, we will consider the commuting square on the right.

*RHS*

$$\begin{aligned}
 &\mathcal{I} \circ Lkp'_1\llbracket pn \rrbracket(\delta, \varpi) \\
 &= Lkp'_1\llbracket pn \rrbracket(\delta, \varpi) \\
 &= \varpi \circ \delta(pn)
 \end{aligned}$$

*LHS*

$$\begin{aligned}
 &Lkp_0\llbracket pn \rrbracket \circ \mathfrak{R}_1(\delta, \varpi) \\
 &= \mathfrak{R}_1(\delta, \varpi)(pn)
 \end{aligned}$$

This establishes the definition of the retrieve function

$$\mathfrak{R}_1: (\delta, \varpi) \mapsto \varpi \circ \delta$$

Now we will look at the square on the left.

*RHS*

$$\begin{aligned} & \mathfrak{R}_1 \circ Lkp'_1 \llbracket fn \rrbracket (\kappa, \tau) \\ &= \mathfrak{R}_1 (\tau \circ \kappa (fn)) \end{aligned}$$

*LHS*

$$\begin{aligned} & Lkp_0 \llbracket fn \rrbracket \circ \mathfrak{R}'_{10} (\kappa, \tau) \\ &= Lkp_0 \llbracket fn \rrbracket (- \xrightarrow{m} \circ) (\tau \circ \kappa) \\ &= ((- \xrightarrow{m} \circ) (\tau \circ \kappa)) (fn) \end{aligned}$$

and we have a second definition of the retrieve function

$$\mathfrak{R}_1: (\tau \circ \kappa (fn)) \mapsto ((- \xrightarrow{m} \circ) (\tau \circ \kappa)) (fn)$$

It was precisely the attempted verification of this command with respect to the original published retrieve function that led to the construction of the new model and, in particular, the apparently strange retrieve function,  $\mathfrak{R}'_{10}: (\kappa, \tau) \mapsto (- \xrightarrow{m} \circ) (\tau \circ \kappa)$ .

*Verification with respect to the retrieve function  $\mathfrak{R}'_{11}$*

$$\begin{array}{ccccc} (\kappa, \tau) & \xrightarrow{Lkp'_1 \llbracket fn \rrbracket} & \tau \circ \kappa (fn) = (\delta, \varpi) & \xrightarrow{Lkp'_1 \llbracket pn \rrbracket} & \varpi \circ \delta (pn) \\ \uparrow \mathfrak{R}'_{11} & & \uparrow \mathfrak{R}_2 & & \uparrow \mathcal{I} \\ (c, ds, ps) & \xrightarrow{Lkp_1 \llbracket fn \rrbracket} & ds \circ c (fn) = (\delta, ps) & \xrightarrow{Lkp_1 \llbracket pn \rrbracket} & ps(\delta(pn)) \end{array}$$

where  $\mathfrak{R}_2: (\delta, ps) \mapsto (\delta, \text{rng } \delta \triangleleft ps)$ . First, we will consider the commuting square on the right.

*RHS*

$$\begin{aligned} & \mathcal{I} \circ Lkp_1 \llbracket pn \rrbracket (\delta, ps) \\ &= Lkp_1 \llbracket pn \rrbracket (\delta, ps) \\ &= ps(\delta(pn)) \end{aligned}$$

*LHS*

$$\begin{aligned} & Lkp_1 \llbracket pn \rrbracket \circ \mathfrak{R}_2 (\delta, ps) \\ &= Lkp_1 \llbracket pn \rrbracket (\delta, \text{rng } \delta \triangleleft ps) \\ &= (\text{rng } \delta \triangleleft ps) \circ \delta(pn) \end{aligned}$$

## THE FILE SYSTEM

and this establishes more precisely the real meaning of the get command in the published first level reification model

$$ps(\delta(pn)) = (rng \delta \triangleleft ps) \circ \delta(pn)$$

Now we will consider the square on the left.

*RHS*

$$\begin{aligned} & \mathfrak{R}_2 \circ Lkp_1 \llbracket fn \rrbracket (c, ds, ps) \\ &= \mathfrak{R}_2(ds \circ c(fn), ps) \\ &= (ds \circ c(fn), rng ds \circ c(fn) \triangleleft ps) \end{aligned}$$

*LHS*

$$\begin{aligned} & Lkp'_1 \llbracket fn \rrbracket \circ \mathfrak{R}'_1(c, ds, ps) \\ &= Lkp'_1 \llbracket fn \rrbracket (c, ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} rng) ds) \\ &= (ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} rng) ds) \circ c(fn) \end{aligned}$$

That both the left-hand-side and the right-hand-side are equal may be demonstrated in a manner similar to that of the erase command above. Let  $ds = [\dots, c(fn) \mapsto ds \circ c(fn), \dots]$ , a valid expression by virtue of the pre-condition, or guards, on the get command. Then we have

$$\begin{aligned} & (ds \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} rng) ds) \circ c(fn) \\ &= ([\dots, c(fn) \mapsto ds \circ c(fn), \dots] \\ & \quad \bowtie (- \xrightarrow{m} \triangleleft ps) \circ (- \xrightarrow{m} rng)[\dots, c(fn) \mapsto ds \circ c(fn), \dots]) \circ c(fn) \\ &= ([\dots, c(fn) \mapsto ds \circ c(fn), \dots] \bowtie (- \xrightarrow{m} \triangleleft ps)[\dots, c(fn) \mapsto rng ds \circ c(fn), \dots]) \circ c(fn) \\ &= ([\dots, c(fn) \mapsto ds \circ c(fn), \dots] \bowtie [\dots, c(fn) \mapsto rng ds \circ c(fn) \triangleleft ps, \dots]) \circ c(fn) \\ &= [\dots, c(fn) \mapsto (ds \circ c(fn), rng ds \circ c(fn) \triangleleft ps), \dots] \circ c(fn) \\ &= (ds \circ c(fn), rng ds \circ c(fn) \triangleleft ps) \end{aligned}$$

Q.E.D.

justifying yet again my choice of retrieve function that maps  $FS_1$  to  $FS'_1$ .

## 3.4.5. The Del Command

$$\begin{aligned}
Del_1: Fn \times Pn &\longrightarrow FS_1 \longrightarrow FS_1 \\
Del_1[[fn, pn]](c, ds, ps) &\triangleq \\
&\chi[[fn]]c \wedge \chi[[pn]]ds \circ c(fn) \\
&\rightarrow (c, ds + [c(fn) \mapsto \{pn\} \triangleleft ds \circ c(fn)], \{(ds \circ c(fn))(pn)\} \triangleleft ps-1) \\
&\rightarrow \perp
\end{aligned}$$

$$\begin{aligned}
Del'_1: Fn \times Pn &\longrightarrow FS'_1 \longrightarrow FS'_1 \\
Del'_1[[fn, pn]](\kappa, \tau) &\triangleq \\
&\chi[[fn]]\kappa \wedge \chi[[pn]]\tau \circ \kappa(fn) \\
&\rightarrow \text{let } (\delta, \varpi) = \tau \circ \kappa(fn) \text{ in } (\kappa, \tau + [\kappa(fn) \mapsto (\{pn\} \triangleleft \delta, \{\delta(pn)\} \triangleleft \varpi)]) \\
&\rightarrow \perp
\end{aligned}$$

In this particular case, I feel that, for the purpose of demonstrating the method, it suffices to establish correctness with respect to the two retrieve functions,  $\mathfrak{R}'_{10}$ , and  $\mathfrak{R}'_{11}$ , for ‘pure’ deletion at the back end, omitting proofs for the update of the front end of the file system. Consequently, I factor the given delete command into a lookup command that gives me access to the back end and then a delete operation that is effectively a back end transformation.

*Verification with respect to the retrieve function  $\mathfrak{R}'_{10}$*

$$\begin{array}{ccccc}
\varphi & \xrightarrow{Lkp_0[[fn]]} & \varphi(fn) = \mu & \xrightarrow{Del_0[[pn]]} & \{pn\} \triangleleft \mu \\
\uparrow \mathfrak{R}'_{10} & & \uparrow \mathfrak{R}_1 & & \uparrow \mathfrak{R}_1 \\
(\kappa, \tau) & \xrightarrow{Lkp'_1[[fn]]} & \tau \circ \kappa(fn) = (\delta, \varpi) & \xrightarrow{Del'_1[[pn]]} & (\{pn\} \triangleleft \delta, \{\delta(pn)\} \triangleleft \varpi)
\end{array}$$

Quite clearly, we need only consider the commuting square on the right.

*RHS*

$$\begin{aligned}
&\mathfrak{R}_1 \circ Del'_1[[pn]](\delta, \varpi) \\
&= \mathfrak{R}_1(\{pn\} \triangleleft \delta, \{\delta(pn)\} \triangleleft \varpi) \\
&= \{\delta(pn)\} \triangleleft \varpi \circ (\{pn\} \triangleleft \delta) \\
&= \{pn\} \triangleleft (\varpi \circ \delta)
\end{aligned}$$

Note that the latter step is valid precisely because  $\delta \in DIR_1$  is 1–1 by definition.

# THE FILE SYSTEM

*LHS*

$$\begin{aligned}
 & Del_0[[pn]] \circ \mathfrak{R}_1(\delta, \varpi) \\
 &= Del_0[[pn]](\varpi \circ \delta) \\
 &= \{pn\} \triangleleft (\varpi \circ \delta) \\
 & \text{Q.E.D.}
 \end{aligned}$$

*Verification with respect to the retrieve function  $\mathfrak{R}'_{11}$*

$$\begin{array}{ccccc}
 (\kappa, \tau) & \xrightarrow{Lkp'_1[[fn]]} & \tau \circ \kappa(fn) = (\delta, \varpi) & \xrightarrow{Del'_1[[pn]]} & (\{pn\} \triangleleft \delta, \{\delta(pn)\} \triangleleft \varpi) \\
 \uparrow \mathfrak{R}'_{11} & & \uparrow \mathfrak{R}_2 & & \uparrow \mathfrak{R}_2 \\
 (c, ds, ps) & \xrightarrow{Lkp_1[[fn]]} & ds \circ c(fn) = (\delta, ps) & \xrightarrow{Del_1[[pn]]} & (\{pn\} \triangleleft \delta, \{\delta(pn)\} \triangleleft ps)
 \end{array}$$

Again, we need only consider the commuting square on the right.

*RHS*

$$\begin{aligned}
 & \mathfrak{R}_2 \circ Del_1[[pn]](\delta, ps) \\
 &= \mathfrak{R}_2(\{pn\} \triangleleft \delta, \{\delta(pn)\} \triangleleft ps) \\
 &= (\{pn\} \triangleleft \delta, \text{rng}(\{pn\} \triangleleft \delta) \triangleleft (\{\delta(pn)\} \triangleleft ps))
 \end{aligned}$$

*LHS*

$$\begin{aligned}
 & Del'_1[[pn]] \circ \mathfrak{R}_2(\delta, ps) \\
 &= Del'_1[[pn]](\delta, \text{rng } \delta \triangleleft ps) \\
 &= (\{pn\} \triangleleft \delta, \{\delta(pn)\} \triangleleft (\text{rng } \delta \triangleleft ps))
 \end{aligned}$$

and the result follows if we can prove that

$$\text{rng}(\{pn\} \triangleleft \delta) \triangleleft (\{\delta(pn)\} \triangleleft ps) = \{\delta(pn)\} \triangleleft (\text{rng } \delta \triangleleft ps)$$

Two simple transformations are required. First, since  $|\delta^{-1} \circ \delta_1(pn)| = 1$ , because  $\delta \in DIR_1$  is 1-1, we have  $\text{rng}(\{pn\} \triangleleft \delta) = \{\delta(pn)\} \triangleleft \text{rng } \delta$ . Second, we have an instance of a simple lemma

LEMMA C.1. *Let  $S$  and  $T$  be two sets in the domain  $\mathcal{PX}$ . Then for every map  $\mu \in X \xrightarrow{m} Y$ ,*

$$S \triangleleft (T \triangleleft \mu) = (S \triangleleft T) \triangleleft (S \triangleleft \mu)$$

which concludes the proof. Yet once more I observe that it was in the course of constructing proofs that VDM lemmas such as the above emerged.

## 4. Conclusion

The published file system case study contains many more levels of reification, all of which are interesting in their own right. However, detailed proofs of correctness using the operator calculus with respect to both invariants and retrieve functions have still to be carried out and published in full. I deliberately chose to insert my new model of the first level of reification *between* the two existing models  $FS_0$  and  $FS_1$  in order not to interfere with the subsequent development. Since the new model,  $FS'_1$ , is isomorphic to  $FS_1$  then another fruitful line of development is suggested—a different development path via successive reifications from  $FS'_1$ .

The case study presented me with the opportunity to demonstrate the effectiveness of the operator calculus for the construction of proofs of correctness. In turn, the very act of attempting to carry out the proofs led to a domain of discrete classical mathematics replete with lemmas and theorems that does not yet seem to have been adequately explored. There is more to formal specifications than formal logic and deadline-driven software products.



## References

- Ackermann, Wilhelm. [1928] 1967. On Hilbert's construction of the real numbers. In *From Frege to Gödel, A Source Book in Mathematical Logic, 1879–1931*, 493–507. Edited by Jean van Heijenoort. Cambridge, Massachusetts: Harvard University Press.
- Ackrill, J. L., ed. 1987. *A New Aristotle Reader*. Oxford: Clarendon Press.
- ALRM. 1983. *Reference Manual for the Ada Programming Language*. 29, Avenue de Versailles, 78170 La Celle-Saint-Cloud, France: Alsys.
- Aho, A. V., J. E. Hopcroft and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Reading, Massachusetts: Addison-Wesley.
- Alagić, Suad and Michael A. Arbib. 1978. *The Design of Well-Structured and Correct Programs*. Berlin: Springer-Verlag.
- Altmann, Simon L. 1986. *Rotations, Quaternions, and Double Groups*. Oxford: Clarendon Press.
- Anderson, A. R., and N. D. Belnap Jr. 1975. *Entailment—The Logic of Relevance and Necessity*. Vol. 1. Princeton: Princeton University Press.
- Andrews, D. 1988. Report from the BSI Panel for the Standardisation of VDM (IST/5/50). In *VDM'88, VDM—The Way Ahead*. Lecture Notes in Computer Science, vol. 328:74–8. Edited by R. Bloomfield, L. Marshall and R. Jones. Berlin: Springer-Verlag.
- ANSI/EIA/TIA-553. 1989. *Mobile Station—Land Station Compatibility Specification*. Electronic Industries Association, Engineering Department, 2001 Pennsylvania Ave. N.W., Washington, D.C. 20006.
- ANSI/IEEE Std 830. 1984. *IEEE Guide to Software Requirements Specifications*. The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017, USA.
- Arbib, Michael A. and Ernest G. Manes. 1975. *Arrows, Structures, and Functors, The Categorical Imperative*. London: Academic Press.

- Arsac, J. 1985. *Foundations of Programming*. Translated by F. Duncan. London: Academic Press Inc. Originally published as *Les Bases de la Programmation*. Paris: Dunod, 1983.
- Arsac, J. and Y. Kodratoff. 1982. Some Techniques for Recursion Removal from Recursive Functions. *ACM Transactions on Programming Languages and Systems* 4(2): 295–322.
- Astesiano, E., C. Bendix Nielsen, A. Fantechi, A. Giovini, E. W. Karlsen, F. Mazzanti, G. Reggio and E. Zucca. 1987. *The Draft Formal Definition of Ada, The Dynamic Semantics Definition*. Dansk Datamatik Center/CRAI/IEI/University of Genoa.
- Astesiano, Egidio and Gianna Reggio. 1987. The SMoLCS approach to the Formal Semantics of Programming Languages—A Tutorial Introduction. In *System Development and Ada*. Lecture Notes in Computer Science, vol. 275: 81–116. Edited by A. N. Habermann and U. Montanari. Berlin: Springer-Verlag.
- Backhouse, Roland C. 1979. *Syntax of Programming Languages, Theory and Practice*. Princeton, New Jersey: Prentice-Hall International.
- Backus, John. 1978. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM* 21(8): 613–41.
- Backus, J. W. *et al.* 1963. Revised Report on the Algorithmic Language Algol 60. *Numerische Mathematik* 4(5).
- Baker, G. P. and P. M. S. Hacker. 1983. *Essays on the Philosophical Investigations*. Vol. 1. *Wittgenstein, Meaning and Understanding*. Oxford: Basil Blackwell.
- Barstow, David R., Howard E. Shrobe and Erik Sandewall. 1984. *Interactive Programming Environments*. New York: McGraw-Hill Book Company.
- Berkeley, George. [1710] 1962. *The Principles of Human Knowledge*. Edited with an Introduction by G. J. Warnock. Fontana Press.
- Bézier, Pierre. 1986. *The Mathematical Basis of the UNISURF CAD System*. London: Butterworths.

- Bird, Richard. 1986. *An Introduction to the Theory of Lists*. Working Material for the lectures of Richard Bird, International Summer School on Logic of Programming and Calculi of Discrete Design, Marktoberdorf, West Germany.
- Bird, Richard and Philip Wadler. 1988. *Introduction to Functional Programming*. London: Prentice-Hall International.
- Birkhoff, Garrett and Saunders MacLane. 1965. *A Survey of Modern Algebra*. Third Edition. New York: The Macmillan Company.
- Birrell, N. D. and M. A. Ould. 1985. *A Practical Handbook for Software Development*. Cambridge: Cambridge University Press.
- Bjørner, Dines. 1980. Formalization of Data Base Models. In *Abstract Software Specifications*. Lecture Notes in Computer Science, vol. 86:144–215. Edited by D. Bjørner. Berlin: Springer-Verlag.
- . 1988. *Software Architectures and Programming Systems Design*. Vol. II. *Basic Abstraction Principles*. Lyngby, Denmark: Department of Computer Science, Technical University of Denmark. The material constituted the Lecture Notes of Dines Bjørner for the CEC sponsored VDM'88 Symposium Tutorial, Trinity College, Dublin, Ireland, 12th September 1988. It is a draft of a publication to appear.
- Bjørner, D. and C. B. Jones., eds. 1978. *The Vienna Development Method: The Meta-Language*, Lecture Notes in Computer Science, vol. 61. Berlin: Springer-Verlag.
- . 1982. *Formal Specification and Software Development*. Englewood Cliffs, New Jersey: Prentice-Hall International.
- Bjørner, D. and O. N. Oest, eds. 1980. *Towards a Formal Description of Ada*. Lecture Notes in Computer Science vol. 98. Berlin: Springer-Verlag.
- Bjørner, D., C. A. R. Hoare and H. Langmaack, eds. *VDM'90, VDM and Z—Formal Methods in Software Development*. Lecture Notes in Computer Science vol. 428. Berlin: Springer-Verlag.
- Bjørner, D., C. B. Jones, Mícheál Mac an Airchinnigh and Erik J. Neuhold, eds. 1987. *VDM'87, VDM—A Formal Method at Work*. Lecture Notes in Computer Science vol. 252. Berlin: Springer-Verlag.

- Blikle, Andrzej. 1987. Denotational Engineering or From Denotations to Syntax. In *VDM'87, VDM—A Formal Method at Work*. Lecture Notes in Computer Science, vol. 252:151–209. Edited by D. Bjørner, C. B. Jones, M. Mac an Airchinnigh and E. J. Neuhold. Berlin: Springer-Verlag.
- . 1988. Three-valued Predicates for Software Specification. In *VDM'88, VDM—The Way Ahead*. Lecture Notes in Computer Science, vol. 328:243–66. Edited by R. Bloomfield, L. Marshall and R. Jones. Berlin: Springer-Verlag.
- . 1990. On Conservative Extensions of Syntax in the Process of System Development. In *VDM'90, VDM and Z—Formal Methods in Software Development*. Lecture Notes in Computer Science, vol. 428:504–525. Edited by D. Bjørner, C. A. R. Hoare and H. Langmaack. Berlin: Springer-Verlag.
- Bolt, R. A. 1980. Put-That-There: Voice and Gesture at the Graphics Interface. *ACM SIGGRAPH* 14(3): 262–70.
- Boyer, C. B. 1968. *A History of Mathematics*, New York: John Wiley & Sons, Inc.
- Brady, J. M. 1977. *The Theory of Computer Science, A Programming Approach*. London: Chapman and Hall.
- Brock, Simon and Chris George. 1990. *RAISE Method Manual*. Bregnerødvej 144, DK-3460 Birkerød, Denmark: Computer Resources International A/S. Lecture Notes of the VDM'90 Symposium Tutorial, Kiel, F. R. Germany, April.
- Brooks Jr., Frederick P. 1986. No Silver Bullet, Essence and Accidents of Software Engineering. In *Information Processing 86*, 1069–1076. Edited by Hans-Jürgen Kugler. Amsterdam: North-Holland.
- Burstall, R. M. and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *Journal of the ACM* 24(1): 44–67.
- Burstall, R. and D. Rydeheard. 1986. Computing with Categories. In *Category Theory and Computer Programming*. Lecture Notes in Computer Science, vol. 240:506–19. Edited by David Pitt, Samson Abramsky, Axel Poigné and David Rydeheard. Berlin: Springer-Verlag.

- Ceri, Stefano. 1986. Requirements Collection and Analysis in Information Systems Design. In *Proceedings of IFIP 86*, 205–14. Edited by H.-J. Kugler. Amsterdam: North-Holland.
- Chaitin, Gregory J., 1988. *An Algebraic Equation for the Halting Probability*. In *The Universal Turing Machine, A Half-Century Survey*, 279–283. Edited by Rolf Herken. Oxford: Oxford University Press.
- The Chicago Manual of Style*. [1906] 1982. Thirteenth Edition, Revised and Expanded. Chicago: University of Chicago Press.
- Church, Alonzo. 1956. *Introduction to Mathematical Logic*. Vol. 1. Princeton, New Jersey: Princeton University Press.
- Clark, James. 1980. A VLSI Geometry Processor for Graphics. *IEEE Computer* 13(7): 59–68.
- Clocksin, W. F. and C. S. Mellish. 1987. *Programming in Prolog*. Third, Revised and Extended Edition. Berlin: Springer-Verlag.
- Crow, Frank. 1987. Displays on Display—The Origins of the Teapot. *Computer Graphics & Applications* 7(1): 8–19.
- Davis, P. J. and R. Hersch. 1980. *The Mathematical Experience*, Boston: Birkhäuser. Harmondsworth, Middelsex, England: Penguin Books Ltd, 1983.
- DeRose, Tony D. 1988. Composing Bézier Simplices. *ACM Transactions on Graphics* 7(3): 198–221.
- Diekert, Volker, ed. 1990. *Proceedings of the Workshop of the EBRA-Working-Group No. 3166 “Algebraic and Syntactic Methods in Computer Science (ASMICS)”*. München: Mathematisches Institut und Institut für Informatik der Technischen Universität München.
- Dieudonné, J. 1972. The Historical Development of Algebraic Geometry. *American Mathematical Monthly* 79:826–66.
- Dijkstra, E. W. 1989. On the Cruelty of Really Teaching Computing Science. *Communications of the ACM* 32(12): 1398–404. Note that this is the text of a lecture given at the ACM Computer Science Conference, February 1989.
- Diller, Antoni. 1990. *Z, An Introduction to Formal Methods*. Chichester, England: John Wiley & Sons.
- Dromey, R. G. 1982. *How to Solve it by Computer*. Englewood Cliffs, New Jersey: Prentice-Hall Inc.

- Eilenberg, Samuel. 1974. *Automata, Languages, and Machines*. Vol. A. New York: Academic Press, Inc.
- . 1976. *Automata, Languages, and Machines*. Vol. B. New York: Academic Press, Inc.
- Eiselt, Horst A. and Helmut von Frajer. 1977. *Operations Research Handbook, Standard Algorithms and Methods with Examples*. London: The Macmillan Press, Ltd.
- Enderton, H. B. 1972. *A Mathematical Introduction to Logic*. New York: Academic Press, Inc.
- Falkoff, Adin D. and Kenneth E. Iverson. 1981. *The Evolution of APL*. In *History of Programming Languages*, 661–674. Edited by Richard L. Wexelblat. London: Academic Press.
- Field, Anthony J. and Peter G. Harrison. 1988. *Functional Programming*. Wokingham, England: Addison-Wesley.
- Flew, A. 1979. *A Dictionary of Philosophy*. London: Pan Books Ltd. Note the 1984 edition is cited.
- Foley, James D. and Andries van Dam. 1982. *Fundamentals of Interactive Computer Graphics*. Reading, Massachusetts: Addison-Wesley.
- Gödel, Kurt. [1931] 1967. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems I*. In *From Frege to Gödel, A Source Book in Mathematical Logic, 1879–1931*, 596–617. Edited by Jean van Heijenoort. Cambridge, Massachusetts: Harvard University Press.
- Godement, Roger. 1969. *Algebra*. London: Kershaw Publishing Company Ltd. Translated from the French *Cours d'Algèbre*. Paris: Hermann, 1963.
- Goguen, Joseph A. 1990. *An Algebraic Approach to Refinement*. In *VDM'90, VDM and Z—Formal Methods in Software Development*. Lecture Notes in Computer Science, vol. 428:12–28. Edited by D. bjoerner, C. A. R. Hoare, and H. Langmaack. Berlin: Springer-Verlag.

- Goldwurm, Massimiliano. 1990. *Evaluations of the number of prefixes of traces: an application to algorithm analysis*. In *Proceedings of the Workshop of the EBRA-Working-Group No. 3166 "Algebraic and Syntactic Methods in Computer Science (ASMICS)"*, 68–80. Edited by Volker Diekert. München: Mathematisches Institut und Institut für Informatik der Technischen Universität München.
- Gotlieb, C. C. and Leo R. Gotlieb. 1978. *Data Types and Structures*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.
- Hadamard, J. [1945] 1954. *An Essay on The Psychology of Invention in the Mathematical Field*. Princeton: Princeton University Press. The 1954 Dover edition of an enlarged 1949 Princeton University Press edition is cited.
- Harrison, M. and H. Thimbleby, eds. 1990. *Formal Methods in Human-Computer Interaction*. Cambridge: Cambridge University Press.
- Havelund, Klaus. 1990. *An RSL Tutorial*. Bregnerødvej 144, DK-3460 Birkerød, Denmark: Computer Resources International A/S. Lecture Notes of the VDM'90 Symposium Tutorial, Kiel, Germany, April.
- Havelund, Klaus and Anne Haxthausen. 1990. *RSL Reference Manual*. Bregnerødvej 144, DK-3460 Birkerød, Denmark: Computer Resources International A/S. Lecture Notes of the VDM'90 Symposium Tutorial, Kiel, Germany, April.
- Hawking, Stephen W. 1988. *A Brief History of Time, From the Big Bang to Black Holes*. London: Bantam Press.
- Haxthausen, Anne E. 1990. *A Tutorial on RAISE*. Bregnerødvej 144, DK-3460 Birkerød, Denmark: Computer Resources International A/S. Lecture Notes of the VDM'90 Symposium Tutorial, Kiel, Germany, April.
- Hayes, Ian, ed. 1987. *Specification Case Studies*. Englewood Cliffs, New Jersey: Prentice/Hall International.
- Hearn, Donald, and M. Pauline Baker. 1986. *Computer Graphics*. London: Prentice-Hall International (UK) Limited.
- Henrici, Peter. 1964. *Elements of Numerical Analysis*. New York: John Wiley & Sons, Inc.

- Hermes, Hans. 1969. *Enumerability, Decidability, Computability, An Introduction to the Theory of Recursive Functions*. Berlin: Springer-Verlag. Translated from the German *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit (Grundlehren der mathematischen Wissenschaften, Band 109)* by G. T. Hermann and O. Plassmann. Second revised Edition. 1965.
- Hoare, C. A. R. 1985. *Communicating Sequential Processes*. Englewood Cliffs, New Jersey: Prentice/Hall International.
- Hogger, Christopher John. 1984. *Introduction to Logic Programming*. London: Academic Press.
- Hutton, James. 1982. *Aristotle's Poetics*. Translated, with an Introduction and Notes. Preface by Gordon M. Kirkwood. New York: W. W. Norton & Company.
- Ichbiah, Jean *et al.* 1979. Rationale for the Design of the Ada Programming Language. *SIGPLAN Notices* 14(6), part A.
- Iverson, Kenneth E. 1962. *A Programming Language*. New York: John Wiley & Sons, Inc.
- . 1966. *Elementary Functions: an algorithmic treatment*, Chicago: Science Research Associates, Inc.
- . 1979. Operators. *ACM Transactions on Programming Languages and Systems* 1(2): 161–76.
- . 1980. Notation as a Tool of Thought. *Communications of the ACM* 23(8): 444–65. Note that this is the text of the ACM Turing Award Lecture given by Iverson on 29th October 1979 at ACM'79.
- Jacobson, Nathan. 1974. *Basic Algebra I*. San Francisco: W. H. Freeman and Company.
- Johnson-Laird, P. N. 1983. *Mental Models—Towards a Cognitive Science of Language, Inference, and Consciousness*. Cambridge: Cambridge University Press.
- Johnson, Lee W. and R. Dean Riess. 1977. *Numerical Analysis*. Reading, Massachusetts: Addison-Wesley.
- Jones, Cliff B. 1986. *Systematic Software Development Using VDM*. Englewood Cliffs, New Jersey: Prentice/Hall International.

- Jonkers, H. B. M. 1989. An Introduction to COLD-K. In *Algebraic Methods: Theory, Tools and Applications*. Lecture Notes in Computer Science, vol. 394:139–205. Edited by M. Wirsing and J. A. Bergstra. Berlin: Springer-Verlag.
- Kant, Immanuel. [1781; 1787] 1929. *Critique of Pure Reason*. Translated by Norman Kemp Smith. London: Macmillan Education, Ltd.
- Klir, George J. and Tina A. Folger. 1988. *Fuzzy Sets, Uncertainty, and Information*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.
- Knuth, Donald E. 1973. *The Art of Computer Programming*. Vol. 1. *Fundamental Algorithms*. 2d ed. Reading, Massachusetts: Addison-Wesley.
- . 1973. *The Art of Computer Programming*. Vol. 3. *Sorting and Searching*. Reading, Massachusetts: Addison-Wesley.
- . 1981. *The Art of Computer Programming*. Vol. 2. *Seminumerical Algorithms*. 2d ed. Reading, Massachusetts: Addison-Wesley.
- Korfhage, Robert R. 1966. *Logic and Algorithms, With Applications to the Computer and Information Sciences*. New York: John Wiley & Sons, Inc.
- Kuhn, Thomas S. [1962] 1970. *The Structure of Scientific Revolutions*. Second Edition, Enlarged. Chicago: The University of Chicago Press.
- . 1977. *The Essential Tension, Selected Studies in Scientific Tradition and Change*. Chicago: The University of Chicago Press.
- Lakatos, Imre. 1976. *Proofs and Refutations, The Logic of Mathematical Discovery*. Edited by John Worrall and Elie Zahar. Cambridge: Cambridge University Press.
- Lloyd, G. E. R. 1968. *Aristotle: The Growth and Structure of his Thought*. Cambridge: Cambridge University Press.
- Lösch, Friedrich. 1966. *Tables of Higher Functions*. Seventh (revised) Edition. Stuttgart: B. G. Teubner Verlagsgesellschaft.

- Mac an Airchinnigh, M. 1984a. Ada Packages and the User's Conceptual Model. *ACM SIGPLAN AdaTEC Ada Letters* 3(4): 70–7.
- . 1984b. Some Educational Principles Relating to the Teaching and Use of Ada. In *Proceedings of the Third Joint Ada Europe/AdaTEC Conference*, 201–10. Edited by J. Teller. Cambridge: Cambridge University Press.
- . 1985a. Report on the User's Conceptual Model. In *User Interface Management Systems*, 31–40. Edited by G. E. Pfaff. Berlin: Springer-Verlag.
- . 1985b. A Model of the User's Conceptual Model of . . . In *User Interface Management Systems*, 203–24. Edited by G. E. Pfaff. Berlin: Springer-Verlag.
- . 1986. Conceptual Models and Command Languages. In *Foundation for Human-Computer Communication*, 99–123. Edited by K. Hopper and I. A. Newman. Amsterdam: North-Holland.
- McGettrick, A. D. 1982. *Program Verification using Ada*. Cambridge: Cambridge University Press.
- Mac Lane, S. and G. Birkhoff. 1979. *Algebra*. New York: Macmillan Publishing Co., Inc.
- Manna, Zohar and Richard Waldinger. 1985. *The Logical Basis for Computer Programming*. Vol. 1. *Deductive Reasoning*. Reading, Massachusetts: Addison-Wesley.
- Mazurkiewicz, Antoni. 1987. Trace Theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Lecture Notes in Computer Science, vol. 255:279–324. Edited by W. Brauer, W. Reisig and G. Rozenberg. Berlin: Springer-Verlag.
- Milner, Robin. 1989. *Communication and Concurrency*. New York: Prentice-Hall.
- Minsky, Marvin. 1967. *Computation, Finite and Infinite Machines*. Englewood Cliffs: New Jersey: Prentice-Hall, Inc.
- Morgan, Carroll and Bernard Suffrin. 1984. Specification of the UNIX Filing System. *IEEE Transactions on Software Engineering* 10(2): 128–42. This was reprinted in Hayes 1987, 91–140.

- Newman, William M. and Robert F. Sproull. 1979. *Principles of Interactive Computer Graphics*. Second Edition. New York: McGraw-Hill Book Company.
- Norman, D. A. 1983. Some Observations on Mental Models. In: *Mental Models*, 7–14. Edited by D. Gentner and A. L. Stevens. Hillsdale, New Jersey: Lawrence Erlbaum Associates.
- Papy, Georges. 1964. *Groups*. London: Macmillan & Company Ltd. Translated from the French by Mary Warner. Paris: Dunod, 1961.
- Penrose, Roger. 1990. *The Emperor's New Mind, Concerning Computers, Minds, and The Laws of Physics*. Oxford: Oxford University Press.
- Pereira, Fernando. 1986. *C-Prolog User's Manual*. Version 1.5. University of Edinburgh: Department of Architecture, EDCAAD, Edinburgh Computer Aided Architectural Design.
- Péter, Rózsa. 1967. *Recursive Functions*. Third Revised Edition. New York: Academic Press.
- Pfaff, G. E. 1985. *User Interface Management Systems*. Berlin: Springer-Verlag.
- Piaget, J. 1971. *Biology and Knowledge—An Essay on the Relations between Organic Regulations and Cognitive Processes*. Edinburgh: Edinburgh University Press, Chicago: The University of Chicago.
- Piegl, L. 1986. Representations of rational Bézier curves and surfaces by recursive algorithms. *Computer-Aided Design*, 18(7): 361–6.
- Pólya, George. [1945] 1957. *How to Solve It, A New Aspect of Mathematical Method*. Second Edition. Princeton, New Jersey: Princeton University Press.
- . [1962] 1981. *Mathematical Discovery, On understanding, learning, and teaching problem solving*. Combined Edition. 2 vols. New York: John Wiley & Sons.
- Popper, K. R. 1972. *Objective Knowledge—An Evolutionary Approach* Oxford: Oxford University Press. Note the 1979 edition is cited.
- Prehn, Søren. 1987. From VDM to RAISE. In *VDM '87, VDM—A Formal Method at Work*. Lecture Notes in Computer Science, vol. 252:141–50. Edited by D. Bjørner, C. B. Jones, M. Mac an Airchinnigh and E. J. Neuhold. Berlin: Springer-Verlag.

- Ralston, Anthony. 1971. *Introduction to Programming and Computer Science*. Tokyo: McGraw-Hill Kogakusha, Ltd.
- Rotman, Joseph E. 1973. *The Theory of Groups, An Introduction*. Second Edition. Boston: Allyn and Bacon, Inc.
- Royden, H. L. 1968. *Real Analysis*. London: The Macmillan Company.
- Rydeheard, David E. 1986. Functors and Natural Transformations. In *Category Theory and Computer Programming*. Lecture Notes in Computer Science, vol. 240:43–50. Edited by David Pitt, Samson Abramsky, Axel poigné and David E. Rydeheard. Berlin: Springer-Verlag.
- Salmon, Rod and Mel Slater. 1987. *Computer Graphics, Systems & Concepts*. Wokingham, England: Addison-Wesley.
- Sandewall, Erik. 1977. Some Observations on Conceptual Programming. In *Machine Intelligence*, Vol. 8: 223–65. Edited by E. W. Elcock and D. Michie. New York: John Wiley & Sons.
- Schmidt, David A. 1986. *Denotational Semantics, A Methodology for Language Development*. Boston: Allyn and Bacon, Inc.
- Schmidt, Uwe and Reinhard Völler. 1987. Experience with VDM in NORSK DATA. In Lecture Notes in Computer Science, vol. 252:49–62. Edited by Dines Bjørner, Cliff B. Jones, Mícheál Mac an Airchinnigh and Erik J. Neuhold. Berlin: Springer-Verlag.
- Shneiderman, Ben. 1980. *Software Psychology, Human Factors in Computer and Information Systems*. Cambridge, Massachusetts: Winthrop Publishers Inc.
- Schönfinkel, Moses. [1924] 1967. *On the building blocks of mathematical logic*. In *From Frege to Gödel, A Source Book in Mathematical Logic, 1879–1931*, 355–66. Edited by Jean van Heijenoort. Cambridge, Massachusetts: Harvard University Press. According to W. V. Quine, Schönfinkel’s ideas were presented in 1920 but were written up by Heinrich Behmann in 1924.
- Sen, D. 1987. *Objectives of the British Standardisation of a Language to support the Vienna Development Method*. In *VDM’87, VDM—A Formal Method at Work*. Lecture Notes in Computer Science, vol. 252:321–23. Edited by D. Bjørner, C. B. Jones, M. Mac an Airchinnigh and E. J. Neuhold. Berlin: Springer-Verlag.

- Sowa, J. F. 1984. *Conceptual Structures—Information Processing in Mind and Machine*. Reading, Massachusetts: Addison-Wesley.
- Sproull, Robert F. 1982. Using Program Transformations to Derive Line-Drawing Algorithms. *ACM Transactions on Graphics* 1(4): 259–273.
- STARTS. 1986. *The STARTS Purchasers' Handbook*. Manchester: NCC Publications.
- Sterling, Leon and Ehud Shapiro. 1986. *The Art of Prolog*. Cambridge, Massachusetts: The MIT Press.
- Storbank Pedersen, Jan. 1987. VDM in Three Generations of Ada Formal Descriptions. In *VDM '87, VDM—A Formal Method at Work*. Lecture Notes in Computer Science, vol. 252:33–48. Edited by D. Bjørner, C. B. Jones, M. Mac an Airchinnigh and E. J. Neuhold. Berlin: Springer-Verlag.
- . 1990. *A Tutorial on the BSI/VDM Specification Language*. Bregnerødvej 144, DK-3460 Birkerød, Denmark: Computer Resources International A/S. Lecture Notes of the VDM'90 Symposium Tutorial, Kiel, Germany, April.
- Stoy, J. E. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, Massachusetts: The MIT Press.
- . 1980. Foundations of Denotational Semantics. In *Abstract Software Specifications*. Lecture Notes in Computer Science, vol. 86:43–99. Edited by D. Bjørner. Berlin: Springer-Verlag.
- . 1982. Mathematical Foundations. In *Formal Specification & Software Development*, 47–81. Edited by Dines Bjørner and Cliff B. Jones. Englewood Cliffs, New Jersey: Prentice-Hall International.
- Sufrin, Bernard and Jifeng He. 1990. *Specification, Analysis and Refinement of Interactive Processes*. In *Formal Methods in Human-Computer Interaction*, 153–200. Edited by M. Harrison and H. Thimbleby. Cambridge: Cambridge University Press.
- Tomiyama, T., D. Xue, and Y. Ishida. 1989. An Experience of Developing a Design Knowledge Representation Language. In *Draft Proceedings of the Third Eurographics Workshop on Intelligent CAD Systems—Practical Experience and Evaluation*, 165–92. Amsterdam: CWI.

- Took, Roger. 1990. Putting Design into Practice: Formal Specification and the User Interface. In *Formal Methods in Human-Computer Interaction*, 63–96. Edited by M. Harrison and H. Thimbleby. Cambridge: Cambridge University Press.
- Ulam, S. 1976. *Adventures of a Mathematician*. New York: Charles Scribner's Sons.
- Veth, B. 1987. An Integrated Data Description Language for Coding Design Knowledge. In *Intelligent CAD Systems I*, 295–313. Edited by P. J. W. ten Hagen and T. Toyima. Berlin: Springer-Verlag.
- Vollmann, Thomas E., William L. Berry and D. Clay Whybark. 1984. *Manufacturing Planning and Control Systems*. Homewood, Illinois: Richard D. Irwin, Inc.
- Vuillemin, Jean. 1980. A Unifying Look at Data Structures. *Communications of the ACM* 23(4): 229–39.
- Walsh, G. R. 1975. *Methods of Optimization*. London: John Wiley & Sons.
- Wegner, Peter. 1971. *Programming Languages, Information Structures, and Machine Organization*. London: McGraw-Hill.
- Whitehead, Alfred North. [1911] 1978. *An Introduction to Mathematics*. Oxford: Oxford University Press.
- Wing, Jeannette M. 1988. A Study of 12 Specifications of the Library Problem. *IEEE Software* 5(4): 66–76.
- Winograd, Terry. 1983. *Language as a Cognitive Process*. Vol. 1. *Syntax*. Reading, Massachusetts: Addison-Wesley.
- Wittgenstein, Ludwig. 1974. *Philosophical Grammar*. Edited by Rush Rhees. Translated by Anthony Kenney. Oxford: Basil Blackwell. I used the 1978 edition of the University of California Press, Berkeley, Los Angeles.
- Yadav, Surya B., Ralph R. Bravoco, Akemi T. Chatfield and T. M. Rajkumar. 1988. Comparison of Analysis Techniques for Information Requirement Determination. *Communications of the ACM* 31(9): 1090–115.
- Zilles, Stephen N. 1984. Types, Algebras, and Modelling. In *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, 441–50. Edited by Michael L. Brodie, John Mylopoulos and Joachim W. Schmidt. Berlin: Springer-Verlag.