The Logic of $U\cdot(TP)^2$ [*]

Andrew Butterfield[1]

Lero@TCD, Trinity College Dublin, {butrfeld}@tcd.ie

Abstract. $U\cdot(TP)^2$ is a theorem prover developed to support the Unifying Theories of Programming (UTP) framework. Its primary design goal was to support the higher-order logic, alphabets, equational reasoning and "programs as predicates" style that is prevalent in much of the UTP literature, from the seminal work by Hoare & He onwards. In this paper we focus on the underlying logic of the prover, emphasising those aspects that are tailored to support the style of proof so often used for UTP foundational work. These aspects include support for alphabets, type-inferencing, explicit substitution notation, and explicit meta-notation for general variable-binding lists in quantifiers. The need for these features is illustrated by a running example that develops a theory of UTP designs. We finish with a discussion of issues regarding the soundness of the proof tool, and linkages to existing "industrial strength" provers such as Isabelle, PVS or CoQ.

1 Introduction

Unifying Theories of Programming (UTP) [HH98], is a framework that uses alphabetised predicates to define language semantics in a relational calculus style, in a way that facilitates the unification of otherwise disjoint semantic theories, either by merging them, or using special linking predicates that form a Galois connection. The framework is designed to cover the spectrum from abstract specifications all the way down to near-machine level descriptions, and as a consequence the notion of refinement plays a key role.

Typically the development of a UTP theory involves determining the key observational variables, so fixing the alphabet, then defining healthiness conditions to characterise the predicates that describe feasible behaviour, introducing the language under study as a signature, and giving meaning to that signature using healthy predicates. Algebraic laws of the language can then be developed.

In [But10] we gave an overview of the Unifying Theories of Programming Theorem Prover ($U\cdot(TP)^2$) that we are developing to support such theory development work[1]. The prover is an interactive tool, with a graphical user-interface, designed to make it easy to define a UTP theory and to experiment and perform the key foundational proofs. The motivation for developing this tool, rather

[*] This research was supported by grants 07-RFP-CMSF186 and 08-RFP-CMS1277 from Science Foundation Ireland, as well as partial support from Lero, the Irish Software Engineering Research Centre

[1] In that paper it was called SAOITHÍN, but the name has since changed to $U\cdot(TP)^2$

than using an existing one has been discussed in some detail in [But10]. We do not repeat it here in the introduction, but this paper effectively gives a technical underpinning to that motivation. In this paper we describe the logic behind $U\cdot(TP)^2$, starting from 1st-order equational logic [Tou01], and gradually exposing the extensions required to facilitate the kind of reasoning we require for foundational work. In effect this paper explores the proof infrastructure needed to reason about a theory of a simple imperative language (*While*), built upon a theory of "Designs", itself layered on top of a generic UTP base theory. We start at the bottom looking at the logic and work up until we can see what is needed for the *While* language.

This paper assumes that the reader is familiar with the basic ideas behind UTP, and does not give an introduction to the subject. A good introduction is the key textbook written by C.A.R. Hoare and He Jifeng [HH98], which is free to download from `unifyingtheories.org`.

In the rest of this paper, we use the term "user" to refer to a UTP practitioner involved in the development of new UTP theories, and not a software developer who might want to employ a formal method whose underlying semantics derive from UTP.

1.1 Structure of this paper

Section 2 talks about theories, and gives a visual outline of much of this paper in Figure 1. Section 3 introduces the logic of $U\cdot(TP)^2$, and Section 4 gives us an introduction to definitions common to most theories. In Section 5, and Section 6, we describe how `Theory`s can be layered up to present a UTP Theory of Designs, as well as a theory for a simple While programming language built as an extension on top of Designs. Section 7 and Section 8 discuss issues to do with the trustworthiness and usefulness of $U\cdot(TP)^2$, and finally, Section 9 concludes. A collection of relevant rules can be found in Appendix A.

2 Theories

A UTP theory is a coherent collection of the following items: an alphabet defining the observations that can be made; a set of healthiness conditions that characterise predicates that describe realistic/feasible systems; a signature that defines the abstract syntax of the language being defined; definitions of the language constructs as healthy predicates; and laws that relate the behaviours of the various language components. In $U\cdot(TP)^2$ we use the term "`Theory`" to refer to such collections, along with various other pieces of ancillary information, as well as subsets of a full theory. The ancillary information includes components to support language parsing, local and temporary definitions, as well as proof support in the form of conjectures, theorems and laws. In effect a UTP theory may be constructed in $U\cdot(TP)^2$ as a layering of `Theory` "slices", each looking at a small part of the whole.

As an example, consider Figure 1. Here we see theory slices organised as an

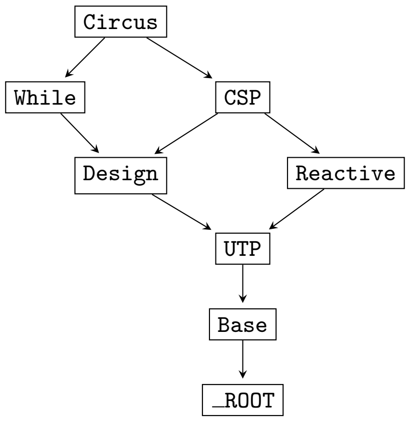

Fig. 1. A Hierarchy of `Theory`s

acyclic directed graph, where each slice inherits material from those below it. At the bottom we have the `_ROOT Theory` (slice), which is hardwired in[2], and simply contains just the axioms of the underlying logic. On top of this a full set of laws of predicate calculus are built (by positing conjectures and proving them), as well as useful theories about equality, and various datatypes, such as numbers, sets and sequences. In $U{\cdot}(TP)^2$ these are presented as a layer of `Theory` slices, but here we simply imagine them all encapsulated into the `Base Theory`, that sits on top of `_ROOT`. On top of this we construct a slice (`UTP`) that presents the language constructs that are common to most UTP theories, e.g. sequential composition, and non-deterministic choice. We then branch: a theory of Designs is implemented by building a `Design Theory` slice on top of `UTP`, and hence incorporating `Base` and `_ROOT`. Similarly, we can define, independently of `Design`, a `Reactive Theory` slice over `UTP`. It is this ability to re-use common material that motivates the splitting of UTP theories into $U{\cdot}(TP)^2$ Theory slices. Figure 1 also shows how further slices allow us to build a theory of a simple imperative language (`While`) on top of `Design`, as well as fusing `Design` and `Reactive` to get `CSP` [Ros97]. A similar fusing of `While` with `CSP` gives `Circus`[OCW09].

In the sequel we shall no longer distinguish between "proper" UTP theories and $U{\cdot}(TP)^2$'s `Theory` slices, simply referring to them all as "theories". We do not give details of the contents of a theory here, but instead elucidate these details as we go through the paper.

[2] `_ROOT` is the only thing hardwired—all other slices and their hierarchy can be custom-built to suit the user.

3 Logic

The logic of $U{\cdot}(\mathit{TP})^2$ is an adaptation of the first-order equational logic described by Tourlakis [Tou01], that fully formalises the logic of Dijkstra, Gries and Schneider [GS93].

3.1 $U{\cdot}(\mathit{TP})^2$ Logic Syntax

We define our logic syntax over a collection of given sets characterising different name-spaces:

$x, y, z \in \mathit{Var}$	(given)	Obs. Variables
$k \in \mathit{Const}$	(given)	Constants
$f, g, h \in \mathit{Name}$	(given)	(Function) Names
$E, F, G \in \mathit{EName}$	(given)	Expression Metavariable Names
$P, Q, R \in \mathit{PName}$	(given)	Predicate Metavariable Names

Variables, constants and function names are as one would expect in a logic with associated equational theories, but we also have explicit meta-variables for expressions and predicates, in the object logic, as many UTP laws are expressed using such.

Expressions and Predicates are defined by mutual induction, because both may contain instances of the other. Expressions denote values in the "world of discourse" (observations) and are typed. Expressions whose type is boolean ($c \in \mathit{Expr}$) form the class of *atomic predicates*:

$c, e \in \mathit{Expr} ::=$	$k \mid x$	Expressions
	$\mid\ f\ e$	Applications
	$\mid\ \lambda\, x \bullet e$	Obs. Abstraction
	$\mid\ \Lambda E \bullet e$	E-var. Abstraction
	$\mid\ \Lambda P \bullet e$	P-var. Abstraction
	$\mid\ \{x \mid p \bullet e\}$	Comprehension
	$\mid\ E$	Explicit Metavariable

Predicates are defined much as expected:

$p, q, r \in \mathit{Pred} ::=$	$\mathit{True} \mid \mathit{False}$	Constant Predicates
	$\mid\ e$	Atomic Predicate (Boolean-valued Expr.)
	$\mid\ \neg\, p$	Negation
	$\mid\ p \maltese q$	Composites, $\maltese \in \{\wedge, \vee, \Rightarrow, \equiv\}$
	$\mid\ P$	Explicit Metavariable
	$\mid\ ¥x \bullet p$	1st-order Quantifiers, $¥ \in \{\forall, \exists, \exists!\}$
	$\mid\ ¥P \bullet p$	higher-order Quantifiers, $¥ \in \{\forall, \exists\}$
	$\mid\ ¥E \bullet p$	higher-order Quantifiers, $¥ \in \{\forall, \exists\}$
	$\mid\ [p]$	Universal Closure (over observations)

The axioms of the logic are shown in Appendix A (A.1, A.3). The axioms are stored in the hardwired `_ROOT` theory, in the *laws* component of the theory, which maps law-names to laws, where a law is a predicate and a side-condition. Side-conditions are a conjunction of zero or more basic conditions, which typically capture relationships between given variables and the free variables (fv) of given predicates.

$$
\begin{aligned}
\mathit{Theory} &= \textbf{record} \\
&\qquad \mathit{laws} : \mathit{Name} \leadsto \mathit{Law} \\
&\qquad \ldots\textbf{end} \\
\mathit{Law} &= \mathit{Pred} \times \mathit{Side} \\
\mathit{Side} &= x \notin \mathsf{fv}.P \mid \{x, y, \ldots\} = \mathsf{fv}.P \mid \{x, y, \ldots\} \supseteq \mathsf{fv}.P \mid \ldots
\end{aligned}
$$

Here the notation $A \leadsto B$ denotes a partial finite function from A to B, and so is effectively a table using a key of type A to lookup a value of type B.

The inference rules (A.2) are implemented, in the main, by a pattern matching mechanism that takes a current proof goal and sees which laws can apply, and a process that allows the user to select and apply the desired one, storing the changed goal in a list that is assumed to be chained together by logical equivalence. The basic structural match has a judgement $\Gamma \vdash P \ddagger T \mid \beta$ that asserts that, given matching environment Γ, test predicate T matches pattern predicate P, with resulting bindings β. Bindings map variables to well-formed expressions or predicates, as appropriate. If we ignore Γ for now, then a representative collection of structural matching rules are:

$$\Gamma \vdash x \ddagger e \mid \{x \mapsto e\} \qquad \ll\text{MATCH-VAR}\gg$$

$$\frac{\Gamma \vdash P_i \ddagger T_i \mid \beta_i \qquad \beta_1 \cong \beta_2}{\Gamma \vdash P_1 \land P_2 \ddagger T_1 \land T_2 \mid \beta_1 \uplus \beta_2} \qquad \ll\text{MATCH-}\land\gg$$

$$\frac{P \ddagger Q \mid \beta_1 \qquad xs \ddagger ys \mid \beta_2 \qquad \beta_1 \cong \beta_2}{\forall\, xs \bullet P \ddagger \forall\, ys \bullet P \mid \beta_1 \uplus \beta_2} \qquad \ll\text{MATCH-}\forall\gg$$

The $\cong$ predicate asserts that two bindings do not map the same variable to different things. The $\uplus$ operator merges two bindings, provided they satisfy $\cong$. An attempted match of T against P fails if no rules apply, or an attempt is made to apply $\uplus$ to two bindings that do not satisfy $\cong$.

In order to facilitate proof, the theory has two components, one for conjectures, which can be viewed as aspirant laws (posited, hopefully true, but not yet proven), and theorems, which are conjectures with proofs:

$$
\begin{aligned}
\mathit{Theory} = {} & \textbf{record} \ldots \\
& \mathit{conjs} : \mathit{Name} \leadsto \mathit{Law} \\
& \mathit{thms} : \mathit{Name} \leadsto \mathit{Proof} \\
& \ldots \textbf{end} \\
\mathit{Proof} = {} & \textbf{record} \\
& \mathit{goal} : \mathit{Pred} \\
& \mathit{sc} : \mathit{Side} \\
& \mathit{done} : \mathbb{B} \\
& \ldots \textbf{end}
\end{aligned}
$$

The workflow is as follows: conjectures can be entered by the user and accumulated in *conjs*. A proof can then be started by selecting a conjecture, which creates a corresponding entry in *thms*, with *goal*, *sc* set to match the conjectured law, and the *done* flag set to false. More than one proof can be active at any one time. A proof is carried out using all the *laws* accessible from the theory. Once a proof is complete, the *done* flag is set true, the corresponding conjecture is deleted, and, usually, a corresponding entry is made into *laws*.

The mechanism as described so far is adequate for proving all and any conjectures based on propositional logic. However it needs extensions to cater for non-propositional logic, and the datatype theories. We will address the non-propositional extensions in the next section on generic UTP. Here we discuss briefly some practical issues with datatype theories. We can define a theory of natural number arithmetic using Peano axioms, for example—the tool supports the creation of a new named empty theory, and the addition of appropriate axiomsby the user into the *laws* table. Operations on natural numbers can be defined axiomatically by adding further laws as required. From this it is possible to prove a range of theorems about natural number operations, e.g. $m + 0 = m$. A similar exercise can be done for sets, and sequences, resulting in laws like $S \cup \emptyset = S$ and $s \frown \langle\rangle = s$. The problem is that we do not just match against whole laws, but can also match against just the lefthand or righthand sides of an equality or equivalence—so the righthand sides of all three laws above will match an arbitrary expression e, offering $e+0$, $e \cup \emptyset$ and $e \frown \langle\rangle$ as replacements. To prevent such spurious matches, we introduce a type system for expressions, and a type-inference engine, that uses context information to deduce the types of expressions like e, and serves to reduce spurious matches to a considerable degree. A theory contains tables to support this feature:

$$
\begin{aligned}
\mathit{Theory} \;=\; & \textbf{record} \ldots \\
& \mathit{type} : \mathit{Name} \leadsto \mathit{Type} \\
& \ldots \textbf{end}
\end{aligned}
$$

$$t \in \mathit{Type} ::= \mathbb{B} \mid \mathbb{Z} \mid \tau \mid \mathcal{P} t \mid \ldots$$

The *Name*s in *type* are typically names of variables or functions.

4 UTP

Some key concepts are common to most UTP theories, namely sequential composition ($;$), non-deterministic choice ($\sqcap$), refinement ($\sqsubseteq$) and conditional ($\lhd c \rhd$). Importantly, in most theories these all have the same definition:

$$
\begin{aligned}
P \mathbin{;} Q &\mathrel{\widehat=} \exists\, Obs_m \bullet P[Obs_m/Obs'] \land Q[Obs_m/Obs] \\
P \sqcap Q &\mathrel{\widehat=} P \lor Q \\
P \sqsubseteq Q &\mathrel{\widehat=} [Q \Rightarrow P] \\
P \lhd c \rhd Q &\mathrel{\widehat=} c \land P \lor \neg\, c \land Q
\end{aligned}
$$

The definitions for $\sqcap$, $\sqsubseteq$ and $\lhd c \rhd$ are unproblematical, and are easily handled by the existing machinery, with one key extension. The definition of $;$ not only makes use of explicit substitution notation, but also raises the question of how to interpret Obs_m, Obs' and Obs. Clearly they stand for the obervational variables of a UTP theory along with appropriate decorations, but how do we support this? In particular, how can we arrange matters so that we only define $;$ once, in such a way that it can be used by many different theories? We will first address the key extension alluded to above, and then return to the problem of sequential composition.

4.1 Defining your own language in $U{\cdot}(TP)^2$

A key aspect of a UTP theory is the signature that captures the abstract syntax of the language being defined. This means that $U{\cdot}(TP)^2$ needs to support user-defined languages. This is achieved by having a table-driven parser for entering predicates, and providing a facility for the user to add new entries to the relevant tables:

$$
\begin{aligned}
Theory = {} & \mathbf{record} \ldots \\
& precs : Name \rightsquigarrow Precedence \\
& lang : Name \rightsquigarrow LangSpec \\
& \ldots \mathbf{end}
\end{aligned}
$$

The *precs* table maps the name of an infix operator to information about its parsing precedence and its associativity. The *lang* table maps a language construct name to a language specification (*LangSpec*) that describes the concrete syntactical structure of that construct. A language specification is a mix of keywords denoting syntactical components like variables (V) , expressions (E) , predicates (P), or various lists of such, interspersed with concrete syntax symbols. We won't give a full definition here but present some examples to give the idea:

- Refinement: we specify this as "`P |= P`", which states that `|=` is an infix operator between two predicates. When this is entered into the *lang* table, a corresponding entry is automatically created in the *precs* table with default values (mid-range precedence, non-associative) which can then be edited by the user to suit. Also entered is a dummy definition for the construct into the *laws* table, which itself then needs to be edited.

– Assignment: specified as "`V := E`", stating that `:=` is an infix operator in-between a variable and expression, resulting in a predicate.

In general defining a language construct (resulting in a predicate) involves adding entries to the *lang* and *laws* tables, and possibly also to the *types* and *precs* tables, depending on the precise nature of the construct. Infix expression operators do not have *lang* entries but require *laws*, *precs* and *types* entries.

When we talk about developing a theory of Designs (Section 5), we shall give a worked-out example of a language definition.

4.2 The problem with $\fatsemi$

The definition of sequential composition,

$$P \fatsemi Q \;\widehat{=}\; \exists\, Obs_m \bullet P[Obs_m/Obs'] \land Q[Obs_m/Obs]$$

says in effect that for each observation, x, say, in Obs, we replace any free occurrence of x' in p by x_m and any free occurrence of x in q by Obs_m, and use existential quantification to hide x_m. In effect the rule above is really a rule-schema, characterising an infinite number of rules, one for each possible alphabet represented by Obs. However, we don't want to repeatedly instantiate this rule and reason about its consequences for each specific alphabet we use. In fact, we want to use the definition in cases where only part of the alphabet is known (Designs again, Section 5). We would prefer to be able to do proofs with the definition as given above, only instantiating Obs where necessary, and then perhaps only partially. In fact, we want to support the following proof (of the associativity of $\fatsemi$) which does not require any instantation of Obs:

$$
\begin{aligned}
& P;\ (Q;\ R) \\
\equiv\ & \exists\, Obs_m \bullet P[Obs_m/Obs'] \land (Q;\ R)[Obs_m/Obs] \\
\equiv\ & \exists\, Obs_m \bullet P[Obs_m/Obs'] \land (\exists\, Obs_n \bullet Q[Obs_n/Obs'] \land R[Obs_n/Obs])[Obs_m/Obs] \\
\equiv\ & \exists\, Obs_m, Obs_n \bullet P[Obs_m/Obs'] \land Q[Obs_n/Obs'][Obs_m/Obs] \land R[Obs_n/Obs][Obs_m/Obs] \\
\equiv\ & \exists\, Obs_m, Obs_n \bullet P[Obs_m/Obs'][Obs_n/Obs'] \land Q[Obs_n, Obs_m/Obs', Obs] \land R[Obs_n/Obs] \\
\equiv\ & \exists\, Obs_n \bullet (\exists\, Obs_m \bullet P[Obs_m/Obs'][Obs_n/Obs'] \land Q[Obs_m/Obs][Obs_n/Obs']) \\
& \qquad \land R[Obs_n/Obs] \\
\equiv\ & \exists\, Obs_n \bullet (\exists\, Obs_m \bullet P[Obs_m/Obs'] \land Q[Obs_m/Obs])[Obs_n/Obs'] \land R[Obs_n/Obs] \\
\equiv\ & \exists\, Obs_n \bullet (P;\ Q)[Obs_n/Obs'] \land R[Obs_n/Obs] \\
\equiv\ & (P;\ Q);\ R
\end{aligned}
$$

In effect we want to reason within our logic about "schematic" variables like Obs and treat the substitution notation as part of the object logic, rather than meta-notation describing the behaviour of an inference rule.

To achieve this we have to add another linguistic innovation to the logic. A common shorthand in most presentations of logic is to view $\forall\, x, y, z \bullet p$ (say) as a shorthand for $\forall\, x \bullet \forall\, y \bullet \forall\, z \bullet p$. Our innovation is not only to add the former as a full part of the logic syntax, but also a further extension. We want to be able to have quantifier variables (e.g. *Obs*) that represent lists of "ordinary" quantifier variables. We do this by splitting the list into two parts, separated by a semi-colon, with those in the first part being ordinary, whilst those in the second part denote lists of variables. The revised syntax of $\forall$ is now:

$$\forall\, x_1, \ldots, x_m \,; xs_1, \ldots, xs_m \bullet P \qquad m \geq 0, n \geq 0, m + n \geq 1$$

Other observation (1st-order) quantifiers are modified similarly. The x_i and xs_j above are "quantifier variables", and will be disambiguated were necessary by referring to the x_i (before the ; sysmbol) as "single variables" and the xs_j (after ; as "list variables"). A list where $m = 0$ is referred to as an "ordinary list". The meaning of a quantifier variable list of the form $x_1, \ldots, x_m \,; xs_1, \ldots, xs_m$ is that it matches an ordinary list of the form $y_1, \ldots, y_{m+k}, k \geq 0$ where each x_i binds to one y_j, each xs_i binds to zero or more y_j, and every y_j is bound exactly once. In principle the bindings associated with a variable like xs_i are non-deterministic, albeit they must be consistent with bindings derived from the match as a whole, i.e. the wider context in which that variable occurs. In practice, heuristics are used in the implementation to select a binding that is hopefully as "good" as possible.

As our proof above largely depended on properties of (explicit) substitution, we have to add it into our logic as well. So we revise our syntax for predicates:

$p, q, r \in Pred$	$::= \ldots$	
	$\mid$ ¥$qvs \bullet p$	1st-order Quantifiers, ¥ $\in \{\forall, \exists, \exists!\}$
	$\mid\ p[e/x]$	Explicit Obs. Substitution
	$\mid\ p[e/E]$	Explicit E-var. Substitution
	$\mid\ p[p/P]$	Explicit P-var. Substitution
$qvs \in QVars$		Quantifier Variable lists
	$::=\ x_1, \ldots, x_m \,; xs_1, \ldots, xs_m$	$m \geq 0, n \geq 0, m + n \geq 1$

Explicit substitutions are also added to expressions as well. Laws regarding explicit substitutions also need to be developed, e.g.

$$p[e/x][f/y] = p[e, f/x, y], \quad x \neq y, y \notin \mathsf{fv}.e$$

but we do not list these here.

This extension allows us to introduce axioms like:

$$(\forall\, x \,; xs \bullet p) \Rightarrow (\forall \,; xs \bullet p[e/x]) \qquad \ll\text{AX-}\forall\, x\text{-INST}\gg$$

rather than relying on a simple single quantifier axiom and the usual conventions regarding the $\forall\, x, y, z$ shorthand. In essence what we have done is to formalise and automate this convention.

To support the definition of $\fatsemi$ we need one further step. The list variable *Obs* does not stand for an arbitrary list of single variables, but is instead intended to stand for precisely those un-dashed variables that are present in the alphabet of the current theory, even if that alphabet has not been fully described. Similarly, Obs' stands for all the dashed variables, and Obs_m denotes the decoration of all the Obs variables. In effect we designate certain list variables (like Obs) as having a special meaning.

The basic matcher described in Section 3, has to be enhanced to perform appropriate matching where non-ordinary quantifier lists are present. To make this work, we need to extend theories to have a table that records the theory alphabet:

$$\begin{aligned} \mathit{Theory} = {} & \mathbf{record} \ldots \\ & \mathit{obs} : \mathit{Name} \rightsquigarrow \mathit{Type} \\ & \ldots \mathbf{end} \end{aligned}$$

The obs table needs to become part of the matching context $\mathit{\Gamma}$, and we introduce rules for matching quantifier lists:

$$\frac{}{\mathit{\Gamma} \vdash ;\mathit{Obs} \ddagger ;\mathit{Obs} \mid \varepsilon}$$

$$\frac{\mathit{Obs}(\mathit{\Gamma}) = \{o_1, \ldots, o_n\}}{\mathit{\Gamma} \vdash ;\mathit{Obs} \ddagger \{o_1, \ldots, o_n\} \mid \{\mathit{Obs} \mapsto \{o_1, \ldots, o_n\}\}}$$

The first rule allows Obs to match itself, and so we can do proofs that do not require it to be expanded to an ordinary list. Note also that in this case an empty binding (ε) is returned. Other matching rules not shown here, take care of decorations, ensuring that Obs matches x, y, z, if appropriate, but not x', y', z'.

We can now define sequential composition in our revised logic as:

$$P \mathbin{;} Q \mathrel{\widehat{=}} \exists ;\mathit{Obs}_m \bullet P[\mathit{Obs}_m/\mathit{Obs}'] \land Q[\mathit{Obs}_m/\mathit{Obs}]$$

and produce a proof as shown earlier. There is an additional extension required to the logic to do this, but we shall motivate and introduce it in the section on Designs (Section 5).

5 Designs

The UTP theory of Designs [HH98, Chp 3] introduces two boolean observation variables (ok, ok') to model program start and termination, and new notation $P \vdash Q$ to represent a predicate with pre and post-conditions:

$$\begin{aligned} ok, ok' &: \mathbb{B} \\ P \vdash Q &\mathrel{\widehat{=}} ok \land P \Rightarrow ok' \land Q, \qquad ok, ok' \notin \mathsf{fv}.P \cup \mathsf{fv}.Q \end{aligned}$$

A key feature to note is that in this theory we do not specify the entire alphabet, but only stipulate that whatever it is, it must contain ok and ok'. In this light we see an even stronger need for special list-variables like Obs as already introduced.

We can already capture this with our theories as described so far:

$$\begin{aligned}
obs(ok) &= \mathbb{B} \\
obs(ok') &= \mathbb{B} \\
lang(\vdash) &= P \vdash P \\
prec(\vdash) &= (n, NonAssoc), \quad n \text{ is desired precedence} \\
laws(\vdash\!-\!DEF) &= (P \vdash Q \equiv ok \land P \Rightarrow ok' \land Q, ok, ok' \notin \mathsf{fv}.P \cup \mathsf{fv}.Q)
\end{aligned}$$

Here we see some side-conditions that assert that neither P nor Q should mention either ok or ok'. These are important side-conditions, without which we do not obtain the desired behaviours (algebraic laws) for designs. However, in proving properties of designs in UTP, we find that the side-conditions play a more active role than encountered in more traditional presentations of logic. In many logics, side-conditions about free variables are syntactic in nature and can always be checked/discharged when applying a rule to a predicate in the logic. In particular, when applying a rule like the one above, both P and Q will have been instantiated to concrete predicates, and so it will be easy to establish the truthfulness of these side-conditions. However in a UTP proof about the properties of designs, we work with explicit meta-variables P and Q for which it is not possible to compute side-condition rules at rule-application time.

Instead, we have to add a post processing stage to law matching. Assuming that a target predicate match involving a law has succeeded returning a binding, We use that binding to translate any side-condition with the law to a corresponding one in the target world. We then need to show that the translated law-side condition is a consequence of any side-conditions associated with the conjecture goal.

In effect, in addition to a syntax for side-conditions, we have to implement a side-condition inference engine that can deduce when one side-condition implies another. Let psc denote the translated pattern side-condition, and tsc denote the side-condition associated with the conjecture being proven. We have to demonstrate that $tsc \Rightarrow psc$. As side-conditions are a conjunction of a few basic primitive side-conditions, we simply take both tsc and $tsc \land psc$, reduce both to a canonical normal form, and check for equality.

To illustrate all of this, here is a proof that $R \vdash S \equiv R \vdash R \land S$, given that $ok, ok' \notin \mathsf{fv}.R \cup \mathsf{fv}.S$. Here we deliberately state our conjecture using different meta-variables to those used to define designs, to show the translation aspect at work. Our proof strategy will be to take the lefthand side and transform it into the righthand side[3].

The first step proceeds when a match of $R \vdash S$ succeeds against pattern $P \vdash Q$ returning the binding $[P \mapsto R, Q \mapsto S]$. However, we need to discharge the side-condition $ok, ok' \notin \mathsf{fv}.P \cup \mathsf{fv}.Q$. We use the bindings to translate this to $ok, ok' \notin \mathsf{fv}.R \cup \mathsf{fv}.S$. This then has to be implied by our conjecture side-condition, which in this case is identical to the law condition, so we can deduce

[3] The strategy in play is noted in the *Proof* record.

that it holds. The proof then proceeds as follows:

$$\begin{array}{ll}
 & R \vdash S \\
\equiv & \quad \text{“ as just discussed above ”} \\
 & ok \land R \Rightarrow ok' \land S \\
\equiv & ok \Rightarrow (R \Rightarrow ok' \land S) \\
\equiv & ok \Rightarrow (R \Rightarrow R \land ok' \land S) \\
\equiv & ok \land R \Rightarrow ok' \land S \land R \\
\equiv & \quad \text{“ see below ”} \\
 & R \vdash R \land S
\end{array}$$

The last step up is similar to the first, as the matching of righthand sides succeeds, and the bindings and translation are the same. This raises a new and important issue to do with observational variables. The variables ok and ok' mentioned above are not arbitrary, but denote specific observations, and so it is important for UTP that they only match themselves in laws, unlike general variables that can match arbitrary expressions (including other variables). This leads to the need to indicate that certain variables in patterns stand for themselves. Such variables are described as being “known”. All *obs* variables are known, and there is also a facility for a user to give names to constants and expressions, and so those names would also be considered “known”. We will not give further details here.

The structural matching rule for variable patterns needs to be modified, using the context Γ to check if a variable is known, here written as $x \in \Gamma$:

$$\frac{x \in \Gamma}{\Gamma \vdash x \ddagger x}$$
$$\frac{x \notin \Gamma}{\Gamma \vdash x \ddagger v \mid \{x \mapsto v\}}$$
$$\frac{x \notin \Gamma}{\Gamma \vdash x \ddagger e \mid \{x \mapsto e\}}$$

Note that when a known variable matches against itself, no binding entry is produced.

At this point, given the hierarchy of Figure 1, we have a theory called `Design`, which has access to the laws of logic, equality, arithmetic and sets, as well as the definitions and associated laws of $\fatsemi$, $\sqcap$, $\sqsubseteq$, $\lhd c \rhd$ and $\vdash$, as well as the known observation variables ok and ok'. In particular, we stress that by being linked in the hierarchy shown, the `Design` theory inherits all the material defined in `UTP`, and all its ancestors. This is quite abstract at this point, so now we move to ground it all a little more.

5.1 Healthiness Conditions

A key feature of UTP is the use of healthiness conditions, expressed typically as monotonic idempotent predicate transformers. To support this in $U{\cdot}(TP)^2$ we need to extend the predicate syntax to include notation for functions over predicates, and the application of those to predicates, and appropriate axiomatisation:

$$\begin{array}{rcll} p, q, r \in \mathit{Pred} & ::= & \ldots & \\ & | & \Lambda P \bullet p, & \text{Predicate Abstraction} \\ & | & p(q), & \text{Predicate Application} \\ (\Lambda P \bullet p)(r) & \equiv & p[r/P] & \end{array}$$

It is at this point that we definitely leave 1st-order logic behind and move up towards 2nd- and higher-orders of logic. At this point it is useful to have a facility to give names to frequently used constructs like healthiness conditions or common predicate fragments, such as the predicates called II, B and J used in the definition of the Reactive theory [HH98, Chp. 8]. In effect we want to give definitions like the following (not necessarily from the theory of Designs):

$$\begin{aligned} \mathbf{H}1 &\mathrel{\widehat=} \Lambda P \bullet ok \Rightarrow P \\ J &\mathrel{\widehat=} (ok \Rightarrow ok') \land wait = wait \land tr' = tr \land ref' = ref \end{aligned}$$

We achieve this by adding in tables into a theory that allow us to write such definitions, and modifying the matching algorithm to treat all names in those tables as "known":

$$\begin{aligned} \mathit{Theory} = \mathbf{record} & \ldots \\ & \mathit{preds} : \mathit{Name} \leadsto \mathit{Pred} \\ & \mathit{exprs} : \mathit{Name} \leadsto \mathit{Expr} \\ & \ldots \mathbf{end} \end{aligned}$$

So, for example, in this theory of Designs we have $\mathit{preds}(\mathbf{H}1) = \Lambda P \bullet ok \Rightarrow P$. The rest of the $U{\cdot}(TP)^2$ machinery can then be used to reason about and use these healthiness conditions in the normal way, so for example, $\mathbf{H}1(q)$ can be converted into $ok \Rightarrow q$, and vice-versa.

6 Programs

To get concrete, we are now going to define the semantics for a simple imperative programming language (a.k.a. *While*), as a UTP Design. To keep things simple for now, we assume the language has exactly three program variables: `x`, `y`, and `z` (we look at the issue of many variables below in Section 6.1).

$$\begin{array}{rcll} u, w \in \mathit{While} & ::= & \mathit{Skip} & \text{do nothing} \\ & | & v := e & \text{Assignment, } v \in \{x,y,z\}, \mathsf{fv}.e \subseteq \{x,y,z\} \\ & | & u \mathbin{;} w & \text{Sequential Composition} \\ & | & u \lhd c \rhd w & \text{Conditional, } \mathsf{fv}.c \subseteq \{x,y,z\} \\ & | & c \circledast w & \text{While-loop, } \mathsf{fv}.c \subseteq \{x,y,z\} \end{array}$$

The alphabet of this theory now contains x, y, z, x', y', z' in addition to ok, ok' inherited from the Design theory. Also inherited are the definitions of $\fatsemi$ and $\lhd\ c\ \rhd$, where now Obs can bind to $ok, x, y, z, ok', x', y', z'$ in pattern matching. We can use the language specification facility to introduce the syntax to $U(TP)^2$, so in $While.lang$ we have:

$$
\begin{aligned}
Skip &\mapsto \texttt{Skip} \\
:= &\mapsto \texttt{V := E} \\
whl &\mapsto \texttt{E ** P}
\end{aligned}
$$

6.1 The $U{\cdot}(TP)^2$ semantics of *Skip* and $x := e$

We start to define the semantics of *Skip*, and we could immediately write:

$$Skip \;\widehat{=}\; True \vdash x' = x \land y' = y \land z' = z$$

While correct, we may worry about what happens if the number of variables increases, or if we want to have some dynamism regarding the number and names of program variables. While we discuss another possible approach to program variables later, for now let's see what we can do to improve things. We could try to use special list variable Obs, to get

$$Skip \;\widehat{=}?\; True \vdash Obs' = Obs$$

but this is not satisfactory, as Obs (Obs') includes ok (ok') and these cannot occur in the design predicates, as per the side-condition used in the Design theory.

The solution here is realise that in many UTP theories we actually have two classes of observations: those associated with the values of variables in the program text under consideration (here x, y and z), and those that capture overall program properties, independent of any program variable (here ok and ok', denoting termination). We shall refer to the former as script variables and the latter as model variables, and add in two new special list-variables called Scr and Mdl to match against the two classes. So in this theory, Scr can match x, y, z, while Mdl matches ok. Also Obs can now match Scr, Mdl, or combinations such as Scr, ok. This requires us to modify the obs table in a theory slightly as we must now record observation class, as well as its type:

$$
\begin{aligned}
Theory \;&=\; \textbf{record} \ldots \\
&\qquad obs : Name \rightsquigarrow Type \times OClass \\
&\qquad \ldots\textbf{end} \\
OClass &::= Model \mid Script
\end{aligned}
$$

So, for example, in theory `Design` we have $obs(ok) = (\mathbb{B}, Model)$, while in theory `While` we have $obs(x) = (t, Script)$, where t is some type. We can now define the semantics of *Skip* as:

$$Skip \;\widehat{=}\; True \vdash Scr' = Scr$$

This definition will now work in a range of theories, provided the observations are classified appropriately. However it does also require a further extension of the law matching algorithm. This has to be modified to allow a pattern like $Scr' = Scr$, given bindings $Scr \mapsto x, y, z$ and $Scr' \mapsto x', y', z'$, to match against a predicate fragment like $x' = x \wedge y' = y \wedge z' = z$. This feature is quite easily implemented as part of the structural matcher.

We now turn our attention to the definition of assignment. The following is *not* satisfactory:

$$x := e \mathrel{\widehat{=}} ?\ True \vdash x' = e \wedge Scr' = Scr$$

First, as x is known, this rule will only match assignments whose variable is x, so we would need a different definition for each program variable—not a good idea! Secondly, $Scr' = Scr$ will match $x' = x \wedge y' = y \wedge z' = z$ as already described, and so we can match $x' = e \wedge x' = x$ which reduces to $x = e$, and then probably *False*. We could try to make the matching of $Scr' = Scr$ against $x' = x \wedge y' = y \wedge z' = z$ "context sensitive", only matching an equality if both sides do not appear "elsewhere", but it is currently very unclear if this is at all feasible. Instead, we extend the list-variable notation to allow modifiers, so we can write the following satisfactory definition for assignment:

$$v := e \mathrel{\widehat{=}} True \vdash v' = e \wedge (Scr' \setminus v') = (Scr \setminus v)$$

The law/pattern variable v is not known, so it will match any of x, y or z, and even ok. However as ok cannot appear in the predicates in a design, any matching of v to ok will lead to a proof that eventually freezes up because the side-condition defining $\vdash$ won't be satisfiable. Imagine we are matching the righthand side of the above definition with $y' = f \wedge x' = x \wedge z' = z$. The matching algorithm will attempt match y' against v', returning a binding $v' \mapsto y'$. This binding gives us enough information to be able to match $(Scr' \setminus v') = (Scr \setminus v)$ against $x' = x \wedge z' = z$.

A further complication arises when we try to prove laws such as:

$$\begin{array}{l}(v := e \fatsemi v := f) \equiv (v := f[e/v]) \\ (u := e \fatsemi v := f) \equiv (v := f[e/u] \fatsemi u := e), \quad v \notin \mathsf{fv}.e\end{array}$$

We will not elaborate on details here, but we find the need to use special list variables like Scr and Scr' in substitutions, so the matching algorithm needs to handle those cases as well.

6.2 Merging program variables

Another way to handle program variables is to group them together into an environment, a mapping from variable names to values:

$$\rho \in Env = Var \rightsquigarrow Val$$

We can then introduce *Model* variables called *state* and *state*$'$. This simplifies the alphabet handling, as it is now fixed, and we can model variable declarations with map extensions. In effect we have no script variables, just model ones, with the consequence that the theory of the alphabet is now independent of the program script. The added complexity now emerges in the type system, because *Val* needs to include all types in *Type*, and the definition of assignment now requires an *eval* function of type $Env \to Expr \leadsto Val$ (here $\oplus$ denotes map override):

$$v := e \;\widehat{=}\; True \vdash state' = state \oplus \{v \mapsto eval(state)(e)\}$$

$U{\cdot}(TP)^2$ can support either style of program variable handling, although the environment-based approach requires a theory of finite maps, and laws defining *eval* for every expression construct, with an added complication of having to handle explicit expression *syntax* in laws. However, the provision of such an *eval* function is not quite as onerous as it sounds as laws providing the meaning of all expression constructs are required in any case.

We are not going to elaborate too much on how to give a semantics to the while-loop construct here, apart from noting that it requires a fixpoint construct in the logic syntax, and an appropriate axiomatisation of fixpoint theory. Then the loop can be defined as the least fixed point of the appropriate functional.

$$\begin{array}{rcll} p, q, r \in Pred & ::= & \ldots & \\ & | & \mu\, P \bullet F(P) & \text{Fixpoint Operator} \\ c \circledast w & \widehat{=} & \mu\, W \bullet (w \mathbin{;} W) \lhd c \rhd Skip & \end{array}$$

7 Soundness

Is $U{\cdot}(TP)^2$ sound? For now, the simple answer is no, due mainly to two reasons.

Firstly, users can add their own laws (axioms), and this always leads to the risk of defining a theory that is inconsistent. As we consider the typical user to be a UTP practitioner with experience in logic and axiomatics, developing foundational theories, we feel it is reasonable to expect such (power) users to be able to use their judgement to avoid such pitfalls. Having said that, it will probably make sense in future versions of the tool to support users at different levels of experience, with the more advanced and dangerous features disabled for novices.

Secondly, the underlying proof engine is very complex, reflecting the complexity of the logic required. At present we are not in a position to guarantee soundness of every action that can be invoked. However, in mitigation, we do point out that the outcome of each basic proof step is highly visible in the tool's GUI. It is clear that eventually we will have to pay serious attention to ensuring the prover is sound (modulo any inconsistencies introduced via user-defined axioms). We envisage two possible approaches:

1. Identifying a very small core from which the whole logic can be developed conservatively, and producing a small piece of prover kernel code that can then be verified. This is the LCF approach adopted for prover systems like HOL[NPW02] and Coq[The08].
2. Developing an encoding of the $U\cdot(TP)^2$ logic into the logic of a system with a verified kernel, such as HOL or CoQ, and using those systems to do automated proof checks, possibly even for each proof step as it is done.

8 Exploitation

Assuming that we have addressed the soundness of the implementation of $U\cdot(TP)^2$, and have used it to develop a nice theory of an interesting language, how useful will the results be if we try to apply them to a real problem? In principle, we could use $U\cdot(TP)^2$ to prove properties of a program written in the language described by our theory. In fact some work has already been done exploring a feature that allows us to take a predicate-transformer theory (e.g. weakest precondition, as per [HH98, Chp. 2, p66]), and a program, and automatically generate proof obligations. However, $U\cdot(TP)^2$ is an interactive proof assistant, designed to support UTP theory development, rather than theory use. In practise, there is no way that $U\cdot(TP)^2$ can realistically compete with existing industrial-strength tools that can both generate and discharge such proof obligations with a high degree of efficiency.

However what does seem to be feasible, is to develop a facility whereby a UTP theory, once complete, can be translated and exported as a theory useable by just such industrial-strength provers. We are currently exploring building such a theorem-prover link to HOL, as recent work has looked at encoding UTP in ProofPower/HOL[OCW06, ZC08], or Isabelle/HOL [FGW10, FGW12]. We hope to be able to make use of these results to build such a $U\cdot(TP)^2$-to-HOL bridge.

9 Conclusions

We can, in effect, summarise the paper by giving a requirements list summarising all the special logic features we desire for $U\cdot(TP)^2$: predicate and expression meta-variables; user language definitions; quantifier list variables, with specials to identify alphabets; explicit substitutions; "semantic" side-conditions; and predicate transformers.

All the above could be implemented using Isabelle, or CoQ, or PVS, or pretty much any higher-order theorem prover. However any algorithm can, in principle, be written in the pure lambda calculus, or expressed as a Turing machine, but this does not make it feasible, desirable or practical to use those notations. Similarly we feel that encoding our requirements into one of the above higher-order systems, at least to the extent that it would be visible to the user, is not the way to meet our requirement for machine-assisted support for UTP foundational reasoning.

The resulting logic is quite large, and space limitations have prevented us from giving a complete description here. More details can be found in a draft of the $U\cdot(TP)^2$ Reference Manual [But12].

References

[But10] Andrew Butterfield. Saoithin: A theorem prover for utp. In Shenchao Qin, editor, *Unifying Theories of Programming, Third International Symposium, UTP 2010, Shanghai, China, November, 2010.*, volume 6445 of *LNCS*, pages 137–156, Shanghai, China, November 2010. Springer.

[But12] Andrew Butterfield. $U \cdot (TP)^2$ reference manual (draft, ongoing). Technical report, School of Computer Science and Statistics, Trinity College Dublin, July 2012. available from `https://www.scss.tcd.ie/Andrew.Butterfield/Saoithin/`.

[FGW10] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Unifying theories in isabelle/HOL. In Shengchao Qin, editor, *Unifying Theories of Programming - Third International Symposium, UTP 2010, Shanghai, China, November 15-16, 2010. Proceedings*, volume 6445 of *Lecture Notes in Computer Science*, pages 188–206. Springer, 2010.

[FGW12] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Isabelle/circus: A process specification and verification environment. In Rajeev Joshi, Peter Müller 0002, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, volume 7152 of *Lecture Notes in Computer Science*, pages 243–260. Springer, 2012.

[GS93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Berlin: Springer Verlag, 1993.

[HH98] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming.* Prentice-Hall, 1998.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.

[OCW06] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying theories in proofpower-Z. In Steve Dunne and Bill Stoddart, editors, *Unifying Theories of Programming, First International Symposium, UTP 2006, Walworth Castle, County Durham, UK, February 5-7, 2006, Revised Selected Papers*, volume 4010 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2006.

[OCW09] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP semantics for circus. *Formal Asp. Comput*, 21(1-2):3–32, 2009.

[Ros97] A.W. Roscoe. *The Theory and Practise of Concurrency.* Prentice-Hall (Pearson), 1997. revised to 2000 and lightly revised to 2005.

[The08] The Coq Development Team. The coq proof assistant, reference manual, version 8.2. Technical report, INRIA, Roquencourt, France, 2008.

[Tou01] George Tourlakis. On the soundness and completeness of equational predicate logics. *J. Log. Comput.*, 11(4):623–653, 2001.

[ZC08] Frank Zeyda and Ana Cavalcanti. Encoding *circus* programs in ProofPower-Z. In Andrew Butterfield, editor, *Unifying Theories of Programming, Second International Symposium, UTP 2008, Trinity College, Dublin, Ireland, September 8-10, 2008, Revised Selected Papers*, volume 5713 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2008.

A Rules

A.1 Propositional Axioms

$((P \equiv Q) \equiv R) \equiv (P \equiv (Q \equiv R))$	≪Ax-≡-assoc≫
$P \equiv Q \equiv Q \equiv P$	≪Ax-≡-symm≫
$\mathit{true} \equiv Q \equiv Q$	≪Ax-≡-id≫
$\mathit{false} \equiv \neg \mathit{true}$	≪Ax-*false*-def≫
$\neg(P \equiv Q) \equiv \neg P \equiv Q$	≪Ax-¬-≡-distr≫
$P \vee Q \equiv Q \vee P$	≪Ax-∨-symm≫
$(P \vee Q) \vee R \equiv P \vee (Q \vee R)$	≪Ax-∨-assoc≫
$P \vee P \equiv P$	≪Ax-∨-idem≫
$P \vee (Q \equiv R) \equiv P \vee Q \equiv P \vee R$	≪Ax-∨-≡-distr≫
$P \vee \neg P$	≪Ax-Excl-Mdl≫
$P \wedge Q \equiv P \equiv Q \equiv P \vee Q$	≪Ax-Golden-Rule≫
$P \Rightarrow Q \equiv P \vee Q \equiv Q$	≪Ax-⇒-def≫

A.2 Inference Rules

$$\frac{P}{P[Q := R]} \qquad \text{(Substitution)}$$

$$\frac{P \equiv Q}{R[S := P] \equiv R[S := Q]} \qquad \text{(Leibniz)}$$

$$\frac{P, P \equiv Q}{Q} \qquad \text{(Equanimity)}$$

A.3 Non-propositional Axioms

$p \vee (\forall ;xs, ys \bullet q)$	≪Ax-$\vee$-$\forall\, x$-scope≫
$\equiv (\forall ;xs \bullet p \vee (\forall ;ys \bullet q)), \quad xs \notin p$	
$p \vee (\forall ;Es, Fs \bullet q)$	≪Ax-$\vee$-$\forall\, E$-scope≫
$\equiv (\forall ;Es \bullet p \vee (\forall ;Fs \bullet q)), \quad Es \notin p$	
$p \vee (\forall ;Ps, Qs \bullet q)$	≪Ax-$\vee$-$\forall\, P$-scope≫
$\equiv (\forall ;Ps \bullet p \vee (\forall ;Qs \bullet q)), \quad Ps \notin p$	
$(\forall ;xs \bullet p \wedge q) \equiv (\forall ;xs \bullet p) \wedge (\forall ;xs \bullet q)$	≪Ax-$\forall\, x$-distr≫
$(\forall ;Es \bullet p \wedge q) \equiv (\forall ;Es \bullet p) \wedge (\forall ;Es \bullet q)$	≪Ax-$\forall\, E$-distr≫
$(\forall ;Ps \bullet p \wedge q) \equiv (\forall ;Ps \bullet p) \wedge (\forall ;Ps \bullet q)$	≪Ax-$\forall\, P$-distr≫
$(\forall\, x ; xs \bullet p) \Rightarrow (\forall ;xs \bullet p[e/x])$	≪Ax-$\forall\, x$-inst≫
$(\forall\, E ; Es \bullet p) \Rightarrow (\forall ;Es \bullet p[e/E])$	≪Ax-$\forall\, E$-inst≫
$(\forall\, P ; Ps \bullet p) \Rightarrow (\forall ;Ps \bullet p[q/P])$	≪Ax-$\forall\, P$-inst≫
$(\exists ;xs \bullet p) \equiv \neg\, (\forall ;xs \bullet \neg\, p)$	≪Ax-$\exists\, x$-def≫
$(\exists ;Es \bullet p) \equiv \neg\, (\forall ;Es \bullet \neg\, p)$	≪Ax-$\exists\, E$-def≫
$(\exists ;Ps \bullet p) \equiv \neg\, (\forall ;Ps \bullet \neg\, p)$	≪Ax-$\exists\, P$-def≫
$\exists! \,; xs \bullet p$	≪Ax-$\exists! x$-def≫
$\equiv (\exists ;xs \bullet p) \wedge \exists ;ys \bullet p[ys/\,;xs] \Rightarrow ys = xs$	
$e = e$	≪Ax-=-refl≫
$(e = \theta x \bullet p)$	≪Ax-θ-Def≫
$\equiv p[e/x] \wedge (\forall\, y \bullet p[y/x] \Rightarrow y = e),\ x \notin e$	
$(\lambda\, x \,; xs \bullet e)f = (\lambda \,;xs \bullet e)[f/x]$	≪Ax-β-OReduce≫
$(\Lambda E \,; Es \bullet q)e \equiv (\Lambda \,; Es \bullet q)[e/E]$	≪Ax-β-EReduce≫
$(\Lambda P \,; Ps \bullet q)r \equiv (\Lambda \,; Ps \bullet q)[r/P]$	≪Ax-β-PReduce≫
$(\bigwedge_{i=1}^{n} x_i = e_i) \Rightarrow (p \equiv p[e/x]), \quad x_i$ distinct,	≪Ax-Leibniz≫
x_i distinct	
$p[x := e] \equiv p[e/x]$	≪Ax-OSubst≫
$p[e/Es] \equiv p[Es := e]$	≪Ax-ESubst≫
$p[q/Ps] \equiv p[Ps := q]$	≪Ax-PSubst≫
$\mathit{true}[e/x] \equiv \mathit{true}$	≪Ax-*true*-OSubst≫
$\mathit{true}[e/Es] \equiv \mathit{true}$	≪Ax-*true*-ESubst≫
$\mathit{true}[q/Ps] \equiv \mathit{true}$	≪Ax-*true*-PSubst≫
$\mathit{false}[e/x] \equiv \mathit{false}$	≪Ax-*false*-OSubst≫
$\mathit{false}[e/Es] \equiv \mathit{false}$	≪Ax-*false*-ESubst≫
$\mathit{false}[q/Ps] \equiv \mathit{false}$	≪Ax-*false*-PSubst≫