An Exploration of Prolog Compilation with LLVM

Mark McGarry
B.A.(Mod.) Computer Science
Final Year Project April 2014
Supervisor: Dr. Mike Brady

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

__________________________________________ April 23, 2014
Mark McGarry
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

__________________________ April 23, 2014
Mark McGarry
An Exploration of Prolog compilation with LLVM

Abstract

Prolog is a logic programming language used commonly in natural language processing and artificial intelligence. Many techniques have been studied to find the best approach when attempting to compile Prolog into the machine code of a target machine. This paper attempts to make use of components of LLVM while adhering to well studied techniques to compile Prolog. The output of this system can also be targeted very easily to a range of common architectures thanks to the code generator module of the LLVM toolchain. Overall, LLVM has been found to be a very useful and usable tool with which to approach this problem.
Contents

Abstract ................................................................. iii
Table of Contents .................................................. iv
Associated code from project is on attached DVD

List of Figures ......................................................... vi

1 Introduction .......................................................... 1
  1.1 Aims of this Project ............................................ 1
  1.2 Overview of Report ............................................ 2
  1.3 An Introduction to LLVM ........................................ 2
  1.4 A Quick Introduction to Prolog ............................... 4

2 Previous Research .................................................. 6
  2.1 Overview ........................................................ 6
  2.2 PLM - The Prolog Machine ..................................... 6
  2.3 Warren’s Abstract Machine .................................... 8
  2.4 Open Prolog .................................................... 10
  2.5 Wamcc - Compiling Prolog to C .............................. 13
  2.6 Summary ....................................................... 15

3 Design ................................................................. 16
  3.1 Summary ......................................................... 16
  3.2 Overall Design ................................................ 16
  3.3 Instruction Set ................................................ 18
    3.3.1 GET instructions ......................................... 19
    3.3.2 PUT instructions ......................................... 19
    3.3.3 SET instructions ......................................... 20
    3.3.4 Unify instructions ....................................... 20
    3.3.5 Control instructions .................................... 21
  3.4 Parser .......................................................... 21
  3.5 Summary ....................................................... 23
## Contents

4 Implementation ......................................................... 24
   4.1 Overview .................................................. 24
   4.2 Abstract Machine .......................................... 24
   4.3 Parser ..................................................... 26

5 Conclusion .............................................................. 30
   5.1 Evaluation .................................................. 30
   5.2 Further Work ............................................... 32

Bibliography .............................................................. 33
List of Figures

2.1 WAM Code example. ................................................. 9
2.2 Head, neck, cut and foot in a clause ............................... 10
3.1 Example Heap representation of clause and query ............... 19
5.1 Heap representation of terms .................................. 31
Chapter 1

Introduction

1.1 Aims of this Project

This report aims to explore the use of LLVM [9] in the process of compiling Prolog clauses and queries. The project will explore the approaches used in a number of projects and implement a system to compile a subset of valid Prolog clauses and queries.

A common approach when compiling any language is to first translate the source language into that of some abstract machine. Such a language is designed in a way so as to aid in the analysis of the code, providing an easier means for translation to the machine code of the target machine. Other projects in the past have made similar attempts at compiling Prolog using languages such as C as the target language. This project will target the LLVM Intermediate Representation language (LLVM IR). LLVM IR is a Single Static Assignment (SSA) language, designed specifically with code analysis in mind. This makes the process of compiling code from LLVM IR to that of some target machine relatively simple. In the case where Prolog is first compiled to a language such as C it will often be compiled again to produce an executable file. C will need to be compiled into another Intermediate language in the process of producing machine code of the target machine. This project attempts to
circumvent the need for this step by compiling straight to the language in use by our
compiler, LLVM. The use of LLVM will allow any code generated using this system
to be run on most common machine architectures and reap the benefits of the code
optimizations performed by LLVM.

The LLVM compiler infrastructure project has become increasingly widespread in
recent years and is in use in a range of industry leading projects. Both the speed of
compilation of source code as well as the speed of execution of the compiled machine
code compare very well against GCC, a very common alternative to LLVM. LLVM
in fact outperforms it in many test cases.

1.2 Overview of Report

Chapter 2 will explore previous research done in this field. A number of similar
projects have been done in this area and in this chapter an attempt will be made to
evaluate and learn from these.

Chapter 3 will discuss the design of this project and the key concerns that were
faced. An attempt will be made to draw from those projects discussed in Chapter 2
and build on the work done before.

Chapter 4 will describe the implementation of this design and some of the chal-
lenges faced in it’s implementation.

Chapter 5 will attempt to evaluate the project as a whole and draw some conclu-
sions from the work that has been done.

1.3 An Introduction to LLVM

The LLVM project is an open source collection of modular compiler and toolchain
technologies. It began as a research project in the University of Illinois. The name
‘LLVM’ is not an acronym (as it once was) but is now the full name of the project. The
modular nature of LLVM means each component is independent and the inputs and outputs of each component can be easily inspected. In fact, in a UNIX environment, LLVM components can be run as separate programs with the output of one being fed into another via a UNIX ‘pipe’. This allows one to implement solutions to a problem without a specific understanding of the underlying LLVM project as a whole. The design of LLVM is largely split into three components:

- Frontend
- Optimizer
- Code Generator

The frontend handles the parsing of the source code and the production of lexical tokens. Lexical tokens are short, meaningful strings found in the source code. From this, simple LLVM Intermediate Representation (LLVM IR) code is generated. This is then passed to the optimizer. The optimizer takes this LLVM IR and performs a number of optimizations. These are dependent on any options specified at compile time. The optimizer also outputs LLVM IR. Finally, this is now passed to the code generator. At this stage, a translation is performed from LLVM IR to the machine code of the target machine.

This structure of design is intended to allow for maximum reuse of components, which also allows for the LLVM system to be easily adopted by a wide range of projects. For example, to compile a new language using LLVM only a new frontend must be implemented. The source code must be parsed and split into tokens and LLVM IR code will be produced from this. From this stage onwards, everything can be reused. The same optimizer can be used as would be used by any other project, regardless of source language or target machine architecture. Code generators have been developed for most common architectures so these can also be reused.

Similarly, to design a compiler targeted to a new architecture, only a new code generator must be reimplemented. An existing front end can be reused, as it will
Chapter 1: Introduction

compile to LLVM IR regardless of the target architecture. The optimizer can be reused, as always. This feature of reusability, makes LLVM a very promising tool to take advantage of.

With this system in mind, this project will be looking to build a frontend for the LLVM system and attempt to make use of both the optimization and code generation stages. This will involve, parsing the Prolog source, identifying lexical tokens, and producing LLVM IR code as the result.

1.4 A Quick Introduction to Prolog

Prolog is a logic programming language developed in the early 1970s. It has proven very useful in the fields of artificial intelligence and natural language processing. Prolog is considered a very high level language, in which the programmer defines a set of facts and rules and a problem. Prolog then attempts to prove or disprove the query posed by the programmer. This differs from conventional procedural languages, where the programmer would define the steps involved in solving the problem as opposed to simply defining the problem. This approach is much more abstract, allowing one to focus on the problem at hand and leaving the solution to be found by the programmer’s tools.

As mentioned above a Prolog program is defined by a set of facts and rules. A fact is in the form of:

\[ \text{man(tom)} \]

This can be read as “tom is a man”. A rule is of the form:

\[ \text{man(X)} :\neg \text{male(X)} \]

As the name suggests, this provides a rule for determining some information. The neck symbol ‘\( :\neg \)’ can be read as ‘if’. This rule states that “Some person X is a man if they are also male”. This is obviously a very simple definition of a man and one
would perhaps like to add further constraints on this definition. Multiple conditions can be listed here, separated by a `;`, the symbol of conjunction.

\[
\text{man}(X) :- \text{adult}(X), \text{male}(X).
\]

This can be read as “some person X is a man if he is both an adult and he is male”. A clause can be defined using atoms, numbers and variables. Atoms are represented using identifiers beginning with lowercase characters. Numbers are represented simply as a series of digits and can have either integer or floating point values. Variables are represented by any identifier starting with a capital letter. Given that in Prolog, programs are defined as a collection of facts and rules, variables allow the programmer to define relationships between the subjects of Prolog programs. In this respect, Prolog is very powerful as quite abstract constraints can be defined for a problem and Prolog can be allowed to determine the result based on this.

When a query is posed, Prolog attempts to unify elements of the query with clauses found in the program. In a query, an atom will unify only with the same atom found in a clause. A variable will unify with any atom or structure. First, Prolog will attempt to unify with the first clause found with the same predicate name (functor) and number of parameters (arity) as the functor posed in the query. If the query fails to unify with this, it will continue on to the next clause in the program. This process is called backtracking. If the query fails to unify with any clause in the program, the query is determined to be false by merit that it cannot be determined to be true.
Chapter 2

Previous Research

2.1 Overview

Since Prolog’s conception in the early 1970s there have been a number of attempts to find an efficient way to compile Prolog. Prolog is normally an interpreted language, with the program being compiled and run clause by clause. The work of David H. D. Warren has been instrumental to more recent research projects in the area of compiling Prolog. This chapter will attempt to investigate the merits of some of these projects while exploring a means by this project may benefit from LLVM as a medium in Prolog compilation. A number of these projects have evolved over time from other, similar projects, attempting to improve on their designs.

2.2 PLM - The Prolog Machine

DEC-10 Prolog is a compiler based Prolog implementation. It uses of an abstract machine, known as the PLM. This abstract machine is used as a stepping stone towards the machine code of the DEC-10. The PLM uses structure sharing and is stack based. This means that internally represented structures can be shared. This is intended to allow for low memory usage and reduce the need to copy structures
between instances of the structure. The PLM was integral to the work later done by
David H. D. Warren in his thesis.

In the PLM, clauses are compiled individually. The PLM defines a number of
instructions for linking these clauses together. When a query is posed, the first clause
with the same functor name and arity is located. A ‘try’ instruction links to the code
for this clause. If unification with this clause fails execution moves to the next ‘try’
instruction. Each clause in the procedure is linked by a series of ‘try’ instructions.
The final clause is linked to by a ‘trylast’ instruction.

Individual clauses are comprised of distinct code areas: ‘head code’, ‘neck code’,
‘body code’, ‘foot code’ and literal code. The head code unifies the arguments of the
head with the corresponding arguments in the goal. With the successful completion
of the head code, the neck code sets up the clause instance’s environment. The body
code is a sequence of function calls to the goals of the clause. The foot code indicates
the termination of the clause, removes the environment and transfers execution back
to the caller.

In certain cases, special instructions are used in place of a combination of these
instructions. For instance, in the case of a fact where the clause has no body, the
neck code would set up an environment never to be used. In this case a specialized
‘neckFoot’ instruction is used, removing this overhead.

In an attempt to make efficient use of memory, the PLM categorizes variables into
global, local temporary and void. Variables occurring in any structure are global.
Variables which occur in the head or body of the clause, but not in any structures are
local. Of these, those which occur only in the head are considered to be temporary.
Finally, those which occur only once in the clause and are not referred to again are
voids. No space needs to be allocated for voids as nothing makes reference to them.
Local and global variables can be stored in a local and global stack respectively. Local
variables will have a shorter lifetime than global variables and this categorization
allows the PLM to reallocate memory faster than it would otherwise be able to do.
As was mentioned before, the PLM uses structure sharing to represent compound terms. This makes use of pairs of pointers; one pointing to skeleton code of the structure and the other to the location of the values contained in the structure. By representing structures in this manner the skeleton code representing the structure can be reused, saving memory.

Aspects of the PLM have been very influential to some systems later designed and thus is a very influential project in this area.

2.3 Warren’s Abstract Machine

In 1983, David H. D. Warren wrote “An Abstract Prolog Instruction Set” [2] describing an abstract machine designed for compiling Prolog clauses and queries. This abstract machine is commonly referred to as the WAM (Warren’s Abstract Machine). It is a register based design (as opposed to a stack based design used by some other approaches), which employs structure copying to represent terms internally. Most Prolog implementations since have followed these choices. It has largely been accepted as the de facto standard for compiling Prolog in this respect. However, Warren’s original report is considered to be difficult to understand and only intended for the expert reader. Thankfully, in 1991 Hassan Aït-Kaci wrote “A tutorial reconstruction of the WAM” [1]. This was intended to make Warren’s work more accessible to the common reader - or at least those with a little knowledge of the area. Hassan Aït-Kaci’s book provides a step by step introduction to the WAM and progressively moves from a very simple implementation, referred to as M0 to the complete implementation, referred to as M3. In this book, all WAM instructions are explicitly given in a pseudo-code format. This provides a very straight forward means with which to begin building a working system.

In the WAM, when parsing Prolog clauses and queries, terms are represented in an area referred to as the heap. This data structure is effectively a stack and can
be indexed into. Warren defines a small instruction set, from which instructions are emitted upon encountering such tokens in the Prolog source. Each clause in the program is encoded separately. Recall that in Prolog a variable is denoted by an identifier beginning with a capitalized letter and atoms are denoted by an identifier beginning with a lower case letter. Clauses of the same procedure, ie. with the same functor name and arity, are linked together and will be analysed individually. Some instructions, such as 'get_value' and 'unify_value', among others, indicate that a symbol has been seen previously in the source and refer back to the previous declaration of the symbol in the heap. The WAM’s use of registers allow for previously seen symbols to be checked to determine which instructions to emit. This is crucial as it allows recurring symbols to be matched when parsing the clause. When a heap representation of both the clause and the query have been constructed, they can be compared to see if the query does in fact unify with a clause.

The WAM has influenced many projects and versions of it are still in use today in projects such as SWI-Prolog and GNU Prolog. These two projects are considered to be quite efficient and are probably the two most commonly used Prolog implementations available today.
### 2.4 Open Prolog

Open Prolog [6], developed by Michael Brady, is a more recent implementation of a Prolog compiler. Developed in 2005 with an eye to exploring a different approach to what the author terms “the standard scheme” of representing terms. As with the approaches in the previous sections, Open Prolog also uses an abstract machine, called the OPAM (Open Prolog Abstract Machine). However, Open Prolog attempts to reinvestigate the approach of the PLM, using structure sharing and a stack based internal representation of terms. Open Prolog also aims to combine the benefits of both a compiler and an interpreter based approach. It tries to maintain the properties of an interpreter - simple inspection, debugging and modification of programs - while maintaining the speed and efficiency of a compiler based approach. In this respect, Open Prolog is attempting to achieve a number of goals outside the scope of this project, but it does study a number of areas that are relevant to this project.

Open Prolog describes a large instruction set. It defines instructions for all constructs in the Prolog language, like the PLM. Integers, atoms, and variables have their own instructions, as well as neck, foot, and cut among other control instructions. However, it also defines extra instructions such as privateCall and privateLastCall.

This larger instruction set adds some complexity to the instruction set of the OPAM but allows for a more efficient implementation in some cases. This is particularly evident in the instructions 'neckfoot' and 'neckfootcut'. As with the PLM, these
instructions make reference to the different portions of a clause, shown in Figure 2.2. This instruction 'neckfoot' replaces and removes the need for two separate instructions, namely 'neck' and 'foot'. An instruction 'neckfoot' signifies that at this stage in the Prolog clause, the parser has reached both the neck and the foot. This means that it must be parsing a fact as there is no body of the clause. This allows for the combination of the operations involved in both the 'neck' and 'foot' instructions and achieving some speedup by taking advantage of this.

An example of Open Prolog Machine listing code is shown below in Listing 2.1, for the head/2 predicate. This predicate will determine if an atom occurs at the head of a given list.
<table>
<thead>
<tr>
<th>Resource</th>
<th>'PRLC'</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>; head/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0030 0020</td>
<td>$22-2</td>
<td></td>
</tr>
<tr>
<td>0032 0000</td>
<td>2-2</td>
<td></td>
</tr>
<tr>
<td>0034 0004</td>
<td>global</td>
<td></td>
</tr>
<tr>
<td>0036 0010</td>
<td>2*8</td>
<td></td>
</tr>
<tr>
<td>0038 0028</td>
<td>structure</td>
<td></td>
</tr>
<tr>
<td>003A 000A</td>
<td>$44- (*)</td>
<td></td>
</tr>
<tr>
<td>003C 002C</td>
<td>neckFoot</td>
<td></td>
</tr>
<tr>
<td>003E 0018</td>
<td>3*8</td>
<td></td>
</tr>
<tr>
<td>0040 0040</td>
<td>clause_size</td>
<td></td>
</tr>
<tr>
<td>0042 0012</td>
<td>$44-$32</td>
<td></td>
</tr>
<tr>
<td>0044 00000000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0048 000C</td>
<td>var</td>
<td></td>
</tr>
<tr>
<td>004A 0010</td>
<td>2*8</td>
<td></td>
</tr>
<tr>
<td>004C 0010</td>
<td>varLand</td>
<td></td>
</tr>
<tr>
<td>004E 0008</td>
<td>1*8</td>
<td></td>
</tr>
<tr>
<td>0050 000C</td>
<td>(<em>)(</em>)-$44</td>
<td></td>
</tr>
</tbody>
</table>

; names
Name .
Arity: 02
Name length: 01
2E
No. of uses: 0001
0044

Name head
Arity: 02
Name length: 04
68656164
No. of uses: 0001
0032

Listing 2.1: Open Prolog code listing

To explain the listing above, the leftmost column of hexadecimal digits signifies
the program address of a given instruction. The next column signifies the opcode or value of the expression to the right. The final column is the actual instructions and operands of the code. For example at address $34 exists an instruction ‘global’. This is followed by an operand of two bytes (2*8) signifying two globals are represented in this clause. At the end of the listing, a description of the predicates used in the program is given. In this example, two predicates were used ‘.’ and ‘head’. The ‘.’ predicate is how lists are represented without the syntactic sugar of “[...]”. With each predicate description, Open Prolog also gives the arity of the predicate as well as a list of program locations at which this predicate was called. This output could be considered to be very verbose but this extent of information allows the compiler to make informed decisions when compiling this code to machine code.

### 2.5 Wamcc - Compiling Prolog to C

The wamcc project [7] is exceedingly relevant to the aims of this project and what it is attempting to achieve. This project translates Prolog to C, via the WAM. The idea behind this conversion is to then have this C code compiled for practically any architecture, thanks to the widespread support of the C programming language. There are C compilers developed for practically every architecture. This makes the output of wamcc extremely portable and quite simple to understand.

First, the Prolog source is translated into the machine code of the WAM. This can be done exactly as for the original WAM as described by Warren [2]. From here the WAM code is translated into a C program structured in such a way as to construct a logical, and readable program.

Labels are inserted into the resulting C code along with matching function prototypes, as can be seen in Listing 2.2 below. This has the effect of simulating function calls as simple jumps in the resulting assembly code. When the compiler encounters a function prototype declaration an instruction is emitted with a hole in it to be filled
by the linker at a later stage. This hole can be filled with the address of the label when it is encountered. This trick allows for the C code outputted by wamcc to be split into small modules, with separate clauses being represented by separate functions in the code. By manipulating the compiler’s output like this the aim is to reduce (or possibly remove) the overhead and stack growth involved in making function calls, while benefiting from the ability to reallocate space previously used on the stack.

```c
void labelp(); /* prototypes */
void labelp1();
void labelq();
void labelr();

#define Direct_Goto(lab) lab()
#define Indirect_Goto(p lab) (*p lab())

void fct p() /* p: q,r,* /
{
    asm("label_p:");
    push(CP); /* allocate */
    CP=label_p1; /* call(q) */
    Direct_Goto(label_q); /* : */
}

void fct p1()
{
    asm("label_p1:");
    pop(CP); /* deallocate */
    Direct_Goto(label_r); /* execute(r) */

    void fct q() /* q. */
    {
        asm("label_q:");
        Indirect_Goto(CP); /* proceed */
    }
}
```

Listing 2.2: Wamcc code example
2.6 Summary

A number of different projects have been discussed here. Some projects such as the PLM have an emphasis on memory efficiency, making excellent reuse of objects in memory through structure sharing and variable usage analysis. The WAM on the other hand has more of an emphasis on speed of execution. These values will be discussed in the chapters to follow as a design is constructed and a simple implementation is developed. The wamcc project outlines a number of concerns and difficulties faced by a project of this nature including the logistics of producing a valid, executable file. This again, will be considered in the design to follow.
Chapter 3

Design

3.1 Summary

This chapter will explore the design decisions and the design process taken in implementing this project. As discussed before, this project will implement a front end system for an LLVM based compiler. The compilation process can be handled entirely by LLVM after obtaining an LLVM IR representation of the Prolog program. As this is the case, the system is only concerned with the translation from Prolog source code to LLVM IR. To be precise, the system is concerned with the compilation of Prolog clauses and queries to LLVM IR. Essentially, a parser and tokeniser for Prolog will be designed. It must be able to identify tokens such as atoms, variables, structures and control sequences. With an internal representation of these tokens the aim will be to produce an executable program to attempt to prove or disprove the given query.

3.2 Overall Design

After studying the projects described in the previous chapter it was decided to follow an approach similar to that of the WAM. Warren’s machine, as described by
Hassan Aït-Kaci provides a very simple and straightforward approach to beginning this project. Aït-Kaci describes a series of stack operations to build heap representations of all terms, clauses and queries, which can be implemented relatively easily in any procedural language. Such a process has been shown to be very useful by projects such as wamcc. A WAM based approach is also used in projects such as SWI-Prolog and GNU Prolog. Such extensive use of the WAM as a starting point for these industrial strength projects is the primary reason it has also been chosen for this project.

The heap will be implemented as a global stack. This will allow terms to be easily constructed and removed from memory through push and pop operations. As these operations will be implemented in LLVM IR, the frontend of this system must output valid LLVM IR to achieve this. So the first task in implementing this project will be to implement these operations.

Next, the parser will be implemented. This will require more thought. The Prolog source must be parsed to identify all significant constructs. Atoms, variables, structures and control sequences must be correctly identified. Upon finding any such term the appropriate instructions must be outputted. All instructions must be outputted in the correct order to produce a syntactically correct LLVM IR listing. As this is the case, some instructions may need to be buffered before adding to the output of the program. With a listing of this code, only the corresponding sequences of instructions for each command need to be run. This will build heap representations of the clause and query in question. This can be achieved by compiling the LLVM IR code through the 'llc' module of LLVM. This module will produce an executable, capable of being targeted to any common architecture specified by command line arguments. Below, is an example of the listing file `wamCode.s` being targeted to an arm architecture.

`llc -march=arm wamCode.s`
3.3 Instruction Set

The instruction set of the abstract machine described above is similar to that of the original WAM. For this machine, there are five categories of instructions: get, put, set, unify, and control instructions. These will all be explained briefly here and later implemented in LLVM. In the descriptions to follow a reference to Xn refers to a variable register n, while a reference to Ai refers to an argument register i. A full listing of these operations is located in Appendix B.

Executing these instructions will build a representation of each clause and query on the heap. These constructs can then be compared for unification. By this process, Prolog can be transformed to an executable file via LLVM.

In Figure 3.1 below it can be seen how clauses and queries are to be represented on the heap. In the representation of the clause on the left, the two subterms to the query (Z and f) can be seen. The variable Z is represented by a reference cell REF at location 0, which points to location 0. This circular reference implies that this variable is unbound.

Following this cell is an STR or structure cell which points to location 2. At that location lies a cell containing the functor name (f) and the arity (0). If the functor f had arity N, the following N cells would be populated with REF cells for each subterm.

A similar sequence of terms can be seen in the representation of the clause on the right. The clause is comprised of a structure at location 0. This structure cell points to location 4, where the functor name (f) and arity (1) can be found. As this functor has an arity of 1, this cell is followed by 1 REF cell for that subterm.

The arrows show in the diagram show the unification of the variable reference at location 0 in the query with the structure cell at location 0 in the clause. Similarly, the structure cell at location 1 unifies with the structure cell at location 1 as both structures refer to a functor f with arity 0.
3.3.1 GET instructions

Get instructions are used to encode the arguments in the head of a clause. Each argument in the head of the clause will correspond to a get instruction.

- get\_structure f/a Xi
  
  Used when encountering a structure or an atom. This causes a register to be allocated with a structure functor.

3.3.2 PUT instructions

Put instructions are concerned with encoding the arguments of a query. Similar to the get instructions listed previously, each argument of a query will result in the emission of a put instruction. A put instruction has the effect of an argument being 'put' into a register.

- put\_structure f/a Xi
– Used when encountering a structure in a clause. This instruction will cause a STR cell to be pushed to the heap. Immediately following the cell will be a functor name and arity.

### 3.3.3 SET instructions

Set instructions are used when parsing subterms in a query. There are only two set instructions defined here. These are emitted upon encountering a variable or atom in the query.

- set_variable Xn
  – Used when encountering a given variable for the first time in a query. This instruction will push and unbound REF cell to the heap. A later reference to this variable will point to this cell.

- set_value Xn Ai
  – Used when a later occurrence of a variable is found. This copies the value of the argument register into the variable register.

### 3.3.4 Unify instructions

Unify instructions concern the arguments of structures and lists of a clause. For example, if argument number one of a clause is a structure with it’s own arguments this will cause a unify instruction to be outputted.

- unify_variable Xn
  – Used when encountering a first occurrence of the variable in a structure or list. This instruction operates differently depending whether the machine is in read mode or write mode. In read mode, the contents of the address of the heap is copied to the variable register given. In write mode, an unbound REF cell is written to the heap and copied to the register given.
• **unify_value Xn**
  - Used when a repeated occurrence of a variable in a structure or a list is found. Again, this instruction operates differently depending on the current mode. In read mode, the given register is unified with the S (stack) register. In write mode, the register is written into the next location on the heap.

### 3.3.5 Control instructions

Control instructions are the final category of instructions to be described for this machine. These alter the flow of control of the program. They can be used to direct the program to the correct clause to try to unify against the clause.

• **call f a**
  - A call instruction follows the completion of the instructions representing the query in the output LLVM IR listing. This instruction causes a branch in the execution to the clause matching the same functor(f) and arity(a) as that of the query.

• **proceed**
  - A proceed instruction signifies the termination of a given clause. It operates by moving the current program counter P to the value stored in our continuation pointer, CP.

### 3.4 Parser

The next stage of the design process is to determine the design of the parser. This is the core of the project and entails the majority of the work. After describing the instructions to be targeted it is clear which elements of the source code it is necessary to identify.
This observation, thankfully defines a short set of requirements for the parser. The parser must be capable of identifying:

- Structures
- Variables
- Recurrences of variables
- Different portions of a clause (Head, neck, body)

In order to identify a structure, there simply needs to be a check that the first character of a token is a lower case, alphabetic character. This constraint will identify all structures and atoms. It is important to realize at this point that an atom can be considered to be a structure with arity 0, ie. one with no subterms.

Variables need to be identified next. Similar to the identification of structures above, only the first character of a given identifier needs to be considered. A variable can be recognised by the first character of the identifier being an upper case alphabetic character.

It must also be possible to recognise a recurring token. Due to the fact that the instruction set defines instructions specifically for the first and following occurrences of variables, it must be possible to keep track of this. The parser must store a list of symbols seen previously in the given clause or query so as to match these symbols. As the scope of any variable is local to a given clause or query however, this symbol table must be cleared before each clause and query is parsed.

Lastly, it is necessary to identify which portion of a clause is currently being parsed. The need for this stems from instructions such as “unify_variable” used when parsing a clause. This instruction is used to denote a list or structure argument in a clause. Similarly, upon reaching the end of a query, there must be a call instruction to cause a branch in the execution to the location of the clause instructions.
3.5 Summary

Both the design of an abstract machine instruction set and the design of a parser of Prolog source have been covered here. The following chapter will discuss the implementation of these two components of this system and verify their design.
Chapter 4

Implementation

4.1 Overview

As was discussed in the previous chapter, this project will involve both an implementation of an abstract machine based on the WAM and a parser for Prolog source. The Prolog parser will emit LLVM IR instructions for the abstract machine. This output can then be compiled and run using LLVM.

4.2 Abstract Machine

As it is the focus of this project, the abstract machine being used in the compilation process will be defined through LLVM. Conveniently, the original WAM was well documented by Aït-Kaci in his book [1]. Together with Warren’s own work [2], the attempt made in this chapter has been to rebuild the WAM through LLVM. This process hopes to verify that translation to LLVM IR is a worthwhile step in the compilation of Prolog. Aït-Kaci describes in detail the operations required for each instruction in the instruction set of the WAM.

This project has depended heavily on the work of Aït-Kaci and has largely adhered to the process described by him. Each of the operations required for this instruction
set have been outlined in the previous chapter. For ease of writing and understanding, these instructions have been first written in C, then translated to LLVM IR through clang. Clang is the C/C++ front end for LLVM and will solely be used here in the development of the abstract machine.

In order to use clang to emit LLVM IR, use the command below. This takes a file “code.c” and produces equivalent LLVM IR code in the file llvm.ll. An example of this is listed below. It is easy to see from this listing that code written in C is simpler to read and understand, so this is the approach that has been taken.

```plaintext
clang -emit-llvm -S code.c -o llvm.ll
```

```plaintext
#include <stdio.h>

int main(){
    printf("call to printf()\n");
    return 0;
}
```

Listing 4.1: C code

```plaintext
#define ModuleID = 'test.c'
target triple = "x86_64-pc-linux-gnu"
@.str = private unnamed_addr constant
    [17 x i8] c"call to printf()\00",
    align 1

define i32 @main() nounwind uwtable {
    %1 = alloca i32, align 4
    store i32 0, i32* %1
    %2 = call i32 (i8*, ...)* @printf(i8
        * getelementptr inbounds ([17 x
        i8]* @.str, i32 0, i32 0))
    ret i32 0
}

@printf(i8*, ...)
```

Listing 4.2: LLVM IR

Each of the instructions described briefly in the previous chapter has been translated from C to LLVM IR by this process. As the problem of developing the WAM in C is a problem approached by a number of other projects in the past, this project has drawn on the work they have done at this point [8]. Source code for the WAM,
written in C has been taken and tailored for the needs of this project.

Each function defined in the implementation of the WAM will maintain its function name and parameter types when translated to LLVM IR. This means the resulting code listing can easily be inspected to examine the workings of the abstract machine if necessary.

4.3 Parser

Implementation of the parser requires significantly more work. With the instruction set defined as above, it is known which elements of a clause or query will need to be identified. Once identified, the appropriate corresponding instructions must be emitted.

Parsing of clauses will be discussed first. The clause will be parsed from beginning to end. In order for the clause to be selected for execution the functor name and arity must first be determined. This can be accomplished easily once the structure of a
clause in Prolog is understood. Recall that a clause can either be a rule or a fact. Therefore, a clause must either be of the form:

\[ \text{functorName(arg1, arg2, argN)}. \]

Or of the form:

\[ \text{functorName(arg1, arg2, argN)} :- \text{body1, body2, bodyN}. \]

For the abstract machine, a clause is defined to be comprised of a functor name with some number of arguments. These arguments can be atoms, variables or structures. This is then either followed by the symbol ".", or the symbol " :- ". The former indicates a fact while the latter signifies a rule. A fact is terminated at this point. A rule however, then contains some number of terms. These terms must be a comma separated sequence of structures. Finally, this list of terms must be terminated by the character ".". This definition defines a number of enforceable constraints for the parser. As the clause is a structure in Prolog, it is enforced that the starting character must be a lower case alphabetic character. The characters which follow this will comprise the functor name, until an opening parentheses is found. This identifies the end of the functor name.

The arity of the functor is then determined. With the location of the first parentheses found, determining the arity of the functor is achieved by traversing what remains of the clause head. The arity can be determined by passing over and counting the number of terms within the parentheses. It is important not to count any terms contained within any of these terms so upon encountering any subsequent opening parentheses, simply continue until a matching closing parentheses is found before continuing counting.

The functor name and arity must be stored for processing later.

Each term within the parentheses in the head of the clause will then be considered. Each term here will require some output to be generated. Firstly, each of the terms found here will be added to a table of symbols found in the clause.
If a variable is found, first it must be determined if the symbol has been seen previously. If the symbol has not been seen previously the system will output instruction (1) below. This is an LLVM function call to ‘get_variable’ which was defined in the previous section. In this case the first parameter to this function will be next available register. For this reason, the maximum register used so far must be tracked throughout this process. If the variable has been seen before, it is necessary to output instruction (2) below. Here, the parameter taken by ‘get_value’ is the same register which was used for this variable previously. In both cases the second parameter is the argument position in the clause, ie. a variable in argument position one takes a 1, position two takes a 2 and so on.

1. call void @get_variable(i32 %d, i32 %d)
2. call void @get_value(i32 %d, i32 %d)

If a structure or atom is found it is necessary to output instruction (3) below. This is an LLVM function call to get_structure which was defined in the instruction set in the previous section. It takes three parameters: a reference to the functor name as a string, the arity of the functor, and a register into which to store the this structure. The functor name and arity, has already determined. Which register is to be used is again determined by the argument position of this term. If the term lies in position one it takes a 1 as a parameter, 2 if in position 2, etc.

3. call void @get_structure(i8* getelementptr inbounds ([%d x i8]* @.str%d , i32 0, i32 0), i32 %d, i32 %d)

If the structure has any subterms they must also be processed, but this will be postponed until later. All terms at this level must be allocated registers first.

Once all terms at this level have been identified and parsed as above with instructions emitted for each as described, each term must then be revisited. If any structures were found in the first pass of the clause, the parameters of each must then
be parsed. This is the reason each of the terms found in the head were stored in the symbol table. This allows the parser to take each argument in turn. Again, it must be determined if a given symbol has been seen before. If the symbol has been seen before, a `unify_value` instruction must be outputted, as below. In this case the register it will be allocated will be the same as it was allocated previously. If the symbol has not been seen before, a `unify_variable` instruction must be outputted. In this case the instruction will take the next available register as a parameter.

4. call void @unify_value(i32 %d)
5. call void @unify_variable(i32 %d)

In the case of a fact, the head of a clause will be followed immediately by a `.`. The parsing of the clause can be terminated here in this case. Alternatively, the clause must be a rule, which should be followed by a `:-`. If the clause conforms to neither of these requirements, the clause is invalid and the user can be told there is a syntax error at the current location in the clause.
Chapter 5

Conclusion

5.1 Evaluation

This chapter will discuss the resulting system that has been designed and implemented throughout the project.

As stated in the introduction to this report, the aim of this project was to implement a frontend of a compiler for Prolog, using LLVM. This compiler was required to be capable of compiling a subset of valid Prolog clauses and queries. For this project to be considered a success, there would also be a requirement that it determined if the use of LLVM IR as an intermediate language is a worthwhile step in the compilation process.

In terms of building a frontend of a compiler for Prolog clauses and queries this project has been successful. This compiler is capable of compiling Prolog facts and queries and the system will correctly build heap representations of both. These representations can then be compared and unified. Where the query should unify with the clause, it does as expected. Where the query should fail, the query fails.

Figure 5.1 shows an example of the heap representation of both a query and a clause generated by this system. The unification of the different terms can be seen. The structure at address 0 in the query unifies with the structure at address 0 in the
Figure 5.1: Heap representation of terms

clause. Similarly, the structure at address 3 in the query unifies with the structure at address 3 in the clause. At address 5 and 6 in the query lie two self referential cells signifying unbound variables. These will unify with the two structures found at address 5 and 6 in the representation of the clause.

In all cases which have been tested, the system responds as expected. Heap representations of both the clause and query are built on the heap and compared for unification. In this respect, this project has met this requirement and is considered a success.

As for the implicit requirement that the system should produce good quality, fast executable files. In the compilation process, simple LLVM IR code is generated. This code is produced without specific optimizations in mind and makes no specific attempt to be efficient. However, after producing this code, as part of the compilation
process, the LLVM IR listing that is produced is then passed through the LLVM optimizer module. This performs many optimizations on the code, including function inlining, loop unrolling and dead code elimination. The use of function inlining is very advantageous to this project. The LLVM IR code being emitted uses many function calls but through examining the resulting optimized code it can be seen that the overhead of this is largely removed through function inlining.

Without a fully functional implementation of Prolog it is difficult to benchmark the results of this project against existing projects. However, the use of the LLVM optimizer and observing the resulting code that is produced seemingly quite efficient and fully functional. With more care taken at the stage of producing LLVM IR, it is likely further optimizations can be made.

With respect to the goal of verifying the usefulness of LLVM IR as a medium to Prolog compilation this also seems to have been successfully verified. A working system has been developed to compile a small subset of the language of Prolog and it is believed that this could, relatively easily if given more time, be extended to a full implementation of the language.

5.2 Further Work

The current implementation of this project covers only a subset of the language of Prolog. It is clear that more work could be done to expand this to an implementation of the entire language of Prolog. It is believed that the work done here is a proof that a more complete implementation is achievable.

This project would also benefit from further work done to improve the efficiency of the implementation. If an effort were made to examine the LLVM IR listing being initially produced, it may be possible to achieve a more efficient implementation. Little effort has been made in this regard for this project so far, so it is quite possible a significant speedup could be achieved here.
Bibliography


