Thesis Proposal

Language Interoperability and Logic Programming Languages

Jonathan Cook, LFCS, July 2001

Abstract

Logic programming languages, such as Prolog, are found to be particularly appropriate for solving problems involving logical deduction from a set of data. The .NET framework aims to support language interoperability, so it could be valuable to find ways of using languages such as Prolog within the .NET framework. One way in which this could be done is by translating Prolog to the new language C#, which is closely related to .NET. There already exist translators which translate Prolog to C and to Java, both languages closely related to C#. Thus, a translator to C# could be obtained by modifying one of these existing translators.

1 Motivation

1.1 Language Translation

There are now many programming languages available and so the question arises of how one should choose a language for solving a particular problem. Often it may be convenient to use a number of languages in one project, which raises the issue of how the languages are to work together. Language translation is a helpful technique for integrating code written in one language into code written in another. If we translate source code to source code the final program can be written in one language. This can be easier to compile than the use of tools which allow one language to call another language.

When a team of programmers is working on a project, some will be more familiar with some of the languages that are being used than others. Also some programmers will be required to maintain code written in a language other than one of those with which they are most familiar. In both these cases it could be helpful for the programmer to be able to use a tool which translates from a language with which they are less familiar to one with which they are more familiar. They can then either use this tool to better understand the code written in another language, or to abandon the less familiar language altogether and just use the language to which the code has been translated for the project.

For such a tool to be useful, then, ideally it should produce readable, well structured code so that it can be easily modified by a human. It should understand, and use where appropriate, common idioms and techniques, which are used by human programmers. Fully realising this ideal is a long way off, but progress can be made towards it. Language translation represents a considerable challenge to computer science, particularly when there is a significant semantic gap between the source and destination languages. Logic languages and imperative languages are separated by such a gap.

1.2 C# and the .NET Platform

1.2.1 C#

C#[3] is a relatively new object-oriented programming language which has drawn on the languages Java[23][16] and C++[28], in an attempt to combine the efficiency of C++ with the elegance and simplicity of Java. Many features not present in Java have been added. Some of these features are intended to allow the writing of efficient code, some are aimed at making it possible to write clearer and more concise code. The cost of this is that the language is more complex than Java. Below I will summarise the essential points of C#, see [3] for more details.

Like Java, C# is compiled into an intermediate language[15]. However, whereas the JVM was originally only intended to run code generated by Java, the intermediate language used by C# (IL) was designed to support very many languages. Work has been done on translating other languages to Java byte-code for example MLj which translates SML to Java byte-code. Whereas Java byte-code is sometimes interpreted, IL is designed to be always compiled to machine code before being run.

C# shares certain features of Java not found in C++, such as garbage collection, reflection, thread support (Java's synchronized keyword), inner classes, and the ability to add finally clauses to try blocks. Also like Java, arrays and strings are stored with information about their size and on every access it is checked whether or not an index is out of range.

C# also has features of C++ not found in Java, such as operator overloading, namespaces, jumps, enums and pointer arithmetic. Most of these are more restrictive in C# than they are in C++. Jumps cannot jump into loops. Pointer arithmetic can only be used in blocks of code marked unsafe. This is not a semantic restriction, but hopefully such "unsafe" blocks will be used rarely. One will then have the reassurance that most of the code certainly does not use pointer arithmetic. Also the garbage collector needs to know that pointers are being used.

With respect to efficiency, in addition to the possibility of using pointers, two types of array can be defined distinguished by different syntax. Firstly one can use Java style "jagged arrays", i.e. every array is an object. `new int[ 3 ][ 4 ][ 5 ]` creates 1+3+12 arrays. Secondly one can define C++ style rectangular arrays, `new int[ 3, 4, 5 ]` creates 1 array. In addition, C++ style structs are available which can be used instead of objects in order to obtain more efficient code. One can choose to allocate both structs and objects either on the stack or on the heap. The struct concept is an integral part of C#'s type system. The primitive types are all really structs, although the compiler is able to give them special treatment to avoid a performance penalty. Also function parameters can be passed by reference.

Event handling is implemented using delegates, a concept described in [3] as a "type-safe object-oriented function pointer, which is able to hold multiple methods". Certainly this is safer than function pointers in C++, but seems less elegant than Java's interfaces. In any case, translating from Prolog may not require any use of event handling.

Finally various syntactical concepts are added such as properties (getter and setters), indexers for treating objects as arrays, and syntactic sugar for simple for loops. In translating from Prolog it might be interesting to see how to detect appropriate occasions to use these.

1.2.2 .NET

.NET[25] is a framework developed by Microsoft intended to support the interaction of web services and clients via XML, with a view to enabling these services to be called across languages and platforms. C# and .NET are related, each being to some extent designed to work well with the other.

In order to facilitate the writing of web services a framework has been developed which allows a number of languages to work together by compiling them all down to a common intermediate language.

If Microsoft are correct in believing that XML Web services will revolutionise the way users interact with applications, with applications being invoked across the Internet, then C# may become an important language. Translating Prolog to C# also provides a means of using Prolog within the .NET framework, as Prolog could be translated first to C# and then to IL. This would enable us to take advantage of the close relationship between C# and .NET in a way that would not be possible if we, for example, used the wamcc (see below) to translate Prolog to IL via C.

1.3 Prolog

Sometimes languages are chosen because they are more efficient in terms of time or space; but often languages are chosen because it is easier to write correct programs or because the language is more suited to the problem which is to be solved. In many cases a great gain in productivity is achieved by using a language better suited to the purpose of the program. Prolog[10], for instance, is a logic language well suited to artifical intelligence programming. To write a program, say, to find a means of winning a simple game can be far more natural in Prolog than other languages. The same can be said for the problem of writing a program to draw logical, rather than statistical, conclusions from a set of data.

Prolog has features, its use of unification for example, which carry a significant performance penalty. However, ingenious techniques have been conceived for making it as efficient as possible. In many applications we would be prepared to sacrifice speed in order to have a clear, concise program. Prolog is not suited to, nor intended to be used for numerical computation, the most obvious example of a field of computing that requires a fast, efficient language. However sometimes, expert systems for example, may be required to draw logical inferences from a very large amount of data; or to solve a problem in which a large number of cases may be considered. In such cases it would be desirable for Prolog to be faster. Unfortunately attempts to translate Prolog to languages such as C (see below) have failed to yield programs which run faster than the corresponding Prolog program run on an efficient Prolog interpreter. Other benefits may be gained, though, such as portability and the ability to easily integrate Prolog with programs written in the faster languages.

1.4 Existing Translators

Many translation systems exist, below I provide brief details of a few. Translation from ML into Java and C is interesting because as in the case of Prolog these are examples of translation from higher-order languages to imperative languages.

1.4.1 Translating ML into Java byte-code

MLj[7][8] translates SML to Java byte-code, and thus allows the integration of ML within a Java program. ML and Java are similar in many ways, including the strength of typing, store management strategy and exception handling semantics. However each has many features that the other does not. The approach taken is to translate the SML into a typed intermediate language, Monadic Intermediate Language (MIL). Each structure is compiled into a MIL term and these terms are then combined to form a term for the whole program. This is then transformed into a low-level code, which then is translated to byte-code. The whole-program approach to compilation allows significant optimisation to be performed.

1.4.2 Translating ML into C

The paper [31] describes a translator which translates ML of New Jersey code to C without the use of assembly code, unlike preceding tools. The initial program is translated into a continuation-passing style, which the authors note, results in code very similar to a C program.

1.4.3 Translating Java into C

Toba[26] translates Java byte-code into C. It does this in a fairly direct manner. It does, however, avoid the creation of an explicit operand stack in the resultant C by taking advantage of Java byte-code's stack invariant. That is, at any point in the byte-code the number and type of items on the stack is the same regardless of the path used to get there. Thus it uses local variables for the operand stack slots, having computed the types of the slots at each point in the byte-code at compile time.

2 Technical Background

2.1 Compilation of Prolog: the WAM

Many of the fastest Prolog interpreters are based on a sophisticated compilation technique known as the WAM[32][33] (Warren Abstract Machine) named after its inventor David H. D. Warren. Work has been done on formally verifying that this compilation technique is correct[27].

Since the invention of the WAM, variations on it have been suggested[17][22]. More major variants include the VAM[21][20] (Vienna Abstract Machine) and the LLPAM [29] which compiles an extention of Prolog which is a linear logic language.

The book [9] from 1984 brings together many articles concerning techniques for implementing Prolog at the time the WAM was proposed, including a Prolog interpreter implemented as a very short LISP program.

The book [1] and slides [2] build up the WAM in stages, starting with an abstract machine M_0 which is only capable of determining whether a goal unifies with a given term. This is then extended to a machine M_1, where the program may consist of more than one fact, with at most one fact per predicate name. The next stage, M_2 is capable of compiling Prolog without backtracking, that is we introduce the ability to express conjunction by having rules of the form

$$a_0 \;:-\; a_1,\; ...,\; a_n$$

M_3 extends this to Pure Prolog, by adding disjunctive definitions (allowing more than one rule for each predicate). Hence support for backtracking is added at this stage. However, this machine still does not support the cut.

Finally support for cut is added, various design optimisations employed, and support for constants, lists and anonymous variables is added.

Below the construction process is briefly summarised.

2.1.1 M_0: Unification

Terms, for example $p(Z, h(Z, W), f(W))$, are represented on the heap using pointers to avoid duplication. Each term consists of a cell stating its predicate and arity, for example $p/3$ followed by, in this case, 3 cells each pointing to structures representing the terms Z, $h(Z, W)$ and $f(W)$. A query term is translated into instructions which build a representation of it, of the form described, on the heap.

The program term is translated into instructions in a similar way to the query term except that the first instruction is for the outermost term, whereas the query term is built bottom up. When executing the program we can assume that the query has already been built. The instructions for the program, however, operate in two different modes: a READ mode and a WRITE mode. We start off in READ mode with the program term being matched functor for functor against the query term. When we encounter an unbound variable in the query, we enter the WRITE mode and the corresponding term in the program is built on the heap. Then the unbound variable is bound to this newly created term.

The read mode uses a standard unification algorithm (UNION/FIND), which uses a stack to recursively match the query term against the program term. During this match at every stage if a binding is not possible then the unification fails, and if it is, the two heap cells are bound by making an unbound one point to the other.

2.1.2 M_1: Allowing Programs with more than One Fact

We now have to solve several unification equations simultaneously. We simply then have to store the code for each fact in a CODE area, and introduce instructions to jump to the relevant piece of CODE. We can only have one fact per predicate name, so we know immediately from the query which piece of code to execute.

2.1.3 M_2: Adding Conjunction

Our program is now a set of clauses of the form

$$a_0 \ :\!- \ a_1, \ ..., \ a_n$$

where a_0 is referred to as the head, with for each predicate name, at most one clause whose head has that predicate as its outermost predicate. A query is of the form:

$$?\!- \ g_1, \ ..., \ g_k$$

The semantics of executing such a query are that leftmost resolution is applied repeatedly. That is we attempt to unify the leftmost goal (g_1) with the head of the clause in the program which has the same outermost predicate as g_1. If this fails, the entire query fails. If it succeeds we replace g_1 in the query with the $a_1, \ ..., \ a_n$ on the right of the program clause we have selected.

We continue to do this until we fail, and the query fails, or we end up with the empty query which trivially succeeds. In the process of getting to this point all the relevant bindings will have been made and can be reported to the user.

Consider, for example, the following program:

```
a(X) :- b(X), c(X).
```

```
b(1).
c(1).
```

and the query `?- a(1), a(Y).`

The reduction proceeds in the following steps:

```
?- a(1), a(Y).
b(1), c(1), a(Y).   expanding a(1).
c(1), a(Y).         expanding b(1).
a(Y).               expanding c(1).
b(Y), c(Y).         expanding a(Y).
c(Y).               expanding b(Y). Y=1
. [success]         expanding c(Y). Y=1

Y = 1.
```

When we execute a query, then, we need code for each clause of the program which checks whether unification is possible, and if so replaces the head with the body.

To a first approximation we translate

$$p_0(...)\ :-\ p_1(...),\ ...,\ p_n(...)$$

into

get arguments of p_0
put arguments of p_1
call p_1
⋮
put arguments of p_n
call p_n

The problem that is encountered is that variables are reused by each successive p_i and so we need to save permanent variables (those which occur in more than one body goal) in an environment stored in a stack of environments. We add instructions `allocate N` and `deallocate` to make space for the permanent variables at the beginning of a call and pop it off at the end. Thus the code given above is modified by adding an `allocate` at the start and a `deallocate` at the end.

2.1.4 M_3: Adding Disjunction

Now we wish to add backtracking, that is, when a goal fails it may be the case that there exists another clause in the program with the same predicate that would succeed. Hence, failure at this point should not cause the entire query to fail. We should instead backtrack to the last *choice-point* and continue from there.

The choice point contains the argument registers, a pointer to the current environment, a pointer to the choice point to backtrack to if everything from this choice point fails, the next clause to try and so forth. In effect, it contains all the data needed to reconstruct what was going on before the failed attempt at unification began.

Initially one might think of allocating the choice points on a separate stack to the environments. We have the problem, however, that environment frames on the environment stack might end up being popped and then needed again because of backtracking. This is solved by putting both the environment frames and choice points onto a single stack. Then the choice points can protect the environments that preceeded them. Only when all courses of action from a given choice point have failed, is that choice point popped, and then the environment frames which are no longer needed can be popped as well. Thus this scheme does not prolong the life of environment frames for longer than is necessary.

We add three instructions: `try-me-else`, `retry-me-else` and `trust-me`, described below, and the code for a predicate name becomes:

try-me-else L_1
[code for first clause] (as above)
L_1: retry-me-else L_2
[code for second clause]
$\vdots$
L_{k-1}: retry-me-else L_k
[code for penultimate clause]
L_k: trust-me
[code for final clause]

`try-me-else L` pushes a new choice point frame with its next clause field set to `L`. `retry-me-else L'` loads the data stored in the choice point back into the relevant machine variables and changes the next clause field to `L'`. Finally, `trust-me` loads the data in the choice point and then pops it from the stack.

2.1.5 Optimisations

Optimisation is based on three WAM principles. Firstly, heap space should be used sparingly. Secondly registers should be allocated to minimise unnecessary data movement and code size. Thirdly special instructions for special situations should be used where that is more efficient.

Constants and lists enjoy special representations on the heap and special instructions for putting them there and reading them therefrom. Anonymous variables need no registers; and multiple anonymous variables in a row can be processed in one go by a single instruction.

Registers are allocated in a clever way so that some of the instruction instances in the program become vacuous and can be eliminated.

2.1.6 The Cut

We add a backtrack cut register which records the choice point to return to when backtracking over a cut. Cuts can be classified as shallow cuts where the cut comes before the first body goal, for example,

$$h \ :\!-\ !, b_1, b_2.$$

and deep cuts, for example,

$$h \ :\!-\ b_1, !, b_2.$$

We have specialised instructions for these two cases.

2.2 Compilation of Prolog to C: the wamcc

The wamcc[11] translates Prolog to C via the WAM. The paper[11] lists as requirements for a Prolog compiler: extensibility, portability, efficiency and modularity. The authors go on to note that emulating the WAM instructions in C is either inefficient, or if optimised, excessively complex. Hence, they decided to exploit features of the C language to go beyond emulation. The main issue addressed is how branches performed by the WAM are implemented. In a emulator, the program counter is stored as a variable and modified as appropriate after each instruction. An emulator is slowed by its reliance on a fetch, decode, execute cycle.

Much of the paper[11] details how four systems, namely Janus, KL1, Erlang and, wamcc itself, deal with control flow. The reason for this restriction is that the code for each instruction, setting aside control flow, closely follows that of the original WAM described above.

Each system has a different way of dealing with the two types of branch. There are direct branchings, where the location to be jumped to is known when the program is compiled. In addition, we need indirect branching, where the location to be jumped to is only known when the program is run on account of it, in the WAM, being stored in a register. The principal example of an indirect branch is the instruction which terminates the code written for a given predicate.

2.2.1 Janus, KL1 and Erlang

In Janus, normal C branching with the `goto` statement is used. However ANSI C does not support an indirect version of the `goto` statement, and `goto` cannot jump outside of the current function call. Thus Janus compiles a Prolog program into a single C function using a huge switch statement. The resultant enormous C function takes a long time to compile. The nature of Prolog, where queries are entered and compiled on-the-fly, means that a long compilation time is not acceptable.

In KL1 the program is sliced into several functions, with each predicate compiled into a separate function and branchings implemented as function calls. We might consider a scheme which resembles the use of continuations, in that the functions never return, each calls another function before returning. However this can lead to stack overflow. The solution to this is to use a supervisor function of the form

```
fct_supervisor( ) {
  while( PC )
    (*PC)();
}
```

This calls each function, which changes the value of PC to the appropriate continuation, and then returns. The supervisor then calls the next function. The authors of the paper feel that this would be the best solution if one wished to avoid anything beyond ANSI C.

The third system, Erlang, exploits a feature of gcc, that allows us to store a label in a pointer, and to jump to the location contained in that pointer. We again compile each predicate into a separate C function, and we maintain a global table of addresses to jump to. This approach has a number of disadvantages: all variables used must be global for instance, as there is no space reserved for local variables on the C stack.

2.2.2 The wamcc

The wamcc translates a WAM branch into a native code jump, by using the `asm` directive in C. The compiler is fooled into thinking that the label is an external function by declaring a prototype for it. The example given in the paper is for the program:

```
p:- q, r.
q.
```

The WAM code for this program is

```
p: allocate
   call( q )
   deallocate
   execute( r )

q: proceed
```

and this becomes:

```
void label_p( );
...other prototypes for labels...

#define Direct_Goto(lab)      lab()
```

```
#define Indirect_Goto(p_lab)  (*p_lab)()

void fct_p( ) {
  asm( "label_p" );
  push( CP );
  CP = label_p1;
  Direct_Goto( label_q );
}

void fct_p1( ) {
  asm( "label_p1" );
  pop( CP );
  Direct_Goto( label_r );
}

void fct_q( ) {
  asm( "label_q" );
  Indirect_Goto( CP );
}
```

Hence first `fct_p( )` is called as the function for the clause `p :- q, r`. This tries the goal `q`, making a note to jump to code to try the goal `r` afterwards by putting the label `label_p1` into the continuation pointer.

2.2.3 Summary

Thus, much of the work aimed at translating into C has been directed towards the production of code which is a fast as possible by exploiting the ability in C to get very close to the machine level.

To some extent, the translation of each predicate into a separate function makes the code more readable and natural.

C# on the other hand, should certainly not allow these horrific tricks. It is hoped that there will be other ways to exploit C#'s features to produce fast indirect jumping without resorting to deceiving the compiler. There may be some scope for ingenuity in doing this. My aim, is however, more towards readable code than speed.

2.3 Compilation of Prolog to Java

The papers [4][5][6] describe a system, called Prolog Café which translates LLP to Java via the LLPAM. LLP is a linear logic[12][13] programming language which is a superset of Prolog. In LLP it is possible to specify that assumptions can only be used once, in fact that is the default. Thus the language is resource-conscious and ideally suited to many problems which are based on the consumption of resources.

The example given in the paper is of tiling a board with dominoes. Each square of the board can only be covered by half of one of the dominoes.

Prolog Café is an extension of jProlog[19] which uses a continuation passing style compilation referred to as binarization and detailed in [30]. A commercial system, called MINERVA[24], which compiles Prolog to Java is also available.

2.3.1 Representing Terms

Each term is an object, which is an instance of one of the classes: VariableTerm, IntegerTerm, SymbolTerm, ListTerm, StructureTerm. These classes are all subclasses of an abstract class called Term which declares methods for unification and testing for equality of two terms amongst other things.

Hence the inheritance mechanism of Java is exploited to allow us to have functions which takes terms as an argument, without knowing what type of terms they are.

2.3.2 Representing Predicates

Like terms, each predicate is an instance of the Predicate class. This has fields for the stack (of VariableTerm's); for a Predicate representing the goal to try next (we use a continuation style); and, the code which "executes" the predicate.

A predicate f/n is compiled into a class called PRED_f_n, which is a subclass of Predicate. This contains a function for each clause, compiled as follows: first the head is compiled, then the body is compiled in continuation form, i.e. with each goal of the body calling the next.

Thus the Prolog code:

```
p :- q, r.
p.
```

yields roughly the following Java code:

```
public class PRED_p_0 extends Predicate {
  public PRED_p_0( ) { }
  public PRED_p_0( Predicate cont ) { this.cont = cont; }
  public void init( Term[] args ) { }

  /* p:- q, r. */
  private boolean clause1( ) {
    Predicate v1 = new PRED_r_0( cont );
    Predicate v2 = new PRED_q_0( v1 );
    v2.exec( );
    return false;
  }
```

```
  /* p:- true. */
  private boolean clause2( ) {
    cont.exec( );
    return false;
  }

  public void exec( ) {
    if( clause1( ) ) return;
    if( clause2( ) ) return;
  }
}
```

A more recent version uses the continuation style of KL1, with supervisor functions, in order to avoid stack overflows. In this style, instead of returning nothing, exec( ) returns the continuation.

2.3.3 Dealing with Resources

Thus far, I have described how Prolog Café deals with Prolog. However, as mentioned above, it actually translates a linear logic programming language into Java. It has, therefore, to deal with the creation and consumption of resources.

Resource formulae are compiled into closures, each containing a reference to the bindings of free variables (resource formulae without free variables can be treated as normal), and a pointer to the code. In Java, these closures are represented as objects.

A resource table is created, which contains an array of objects, each representing a primitive resource. Each primitive resource consists, amongst other things, of a consumption level and closure. Resources are added to the table by the use of the operators $\Rightarrow$ and $\multimap$, and removed when we backtrack.

A register, L, is added to store the *current consumption level*. At some point in the proof tree, only resources with consumption level equal to the current value of L may be consumed. When creating a new resource, if it is an exponential resource (may be consumed as many times as you wish) then the value 0 is stored in the consumption level field. Otherwise it is allocated an initial consumption level equal to the current value of L. It is also necessary to store *deadline* information, indicating the point by which a given resource must have been used. These issues are dealt with in detail in [18].

2.3.4 Summary

This technique of compilation, with its use of classes, makes intelligent use of Java's object-oriented features, and appears to be a good starting point for a translator to C#. Also, it adopts the continuation style strategy used by some of the translators to C mentioned above. To implement the strategy used by the wamcc for branchings in Java is, though, inconceivable.

3 Proposed Contribution

3.1 General Discussion

The intended principal contribution of the proposed work is to implement a compiler which will compile Prolog source code into C# source code. This compiler will probably be obtained by modifying an existing compiler. Hopefully such an approach will minimise the amount of mundane and technically uninteresting work which is normally a major part of the implementation of a compiler.

The intended content of the thesis is mainly twofold. Firstly there will be a development of the theory of Prolog translation to the extent required to translate into C#. Secondly there will be a discussion of the implementation with details of the design decisions made and how they relate to the theory developed.

Above I have discussed two very different ways of translating Prolog, one non-object-oriented and one object-oriented. One of the major design decisions is which of these to opt for. Taking into account a desire for readability above speed it seems likely that the object-oriented nature of C# will be exploited. The work will, however, attempt to combine the exploitation of C's efficiency found in the wamcc with the exploitation of Java's object-oriented model found in translators which translate to Java. The starting point will probably be an existing translator to Java. This will be modified to use the extra features that C# provides where appropriate.

It is hoped that the use of object-orientation will yield more readable code than that produced by wamcc; and that the use of C#'s features taken from C++ will yield more efficiently executable code than translators to Java can produce.

Translating a linear logic programming language to C# seems a good idea for two reasons. The languages tend to be supersets of Prolog, so expressiveness is gained, and nothing lost apart from perhaps a loss of performance. Also the problem is more technically interesting than that of compiling just Prolog to C#. The Prolog language is rife with interesting extensions and there should be some scope for interesting work in finding ways of implementing some of these in C#.

The possible original contributions comprise novel extentions to the WAM to exploit features in the C# language, and novel techniques employed to obtain more readable code. It is expected that more progress will be made in the former direction than the latter which seems far more challenging.

3.2 Specific Tasks

The following are some specific tasks which could form part of the work.

• Modification of a linear logic language to Java translator to produce C# classes rather than Java classes. This will involve modifications to account for the different object-orientation features provided by C#.

• Finding a scheme for direct and indirect branching in C# which is as fast as possible. Considering the importance placed on this issue in the existing literature, it is conceivable that a paper could result from this, if a suitably novel scheme is found.

• Investigating the relative merits of using structs instead of objects; and of allocating those structs or objects on the stack or on the heap.

• Investigating the use of operator overloading to produce more concise code, but only if it makes sense and is still readable. For example, it could be used to express the fact that two terms unify concisely.

• Implementing some extensions of Prolog in C#. For example, adding support for concurrency by taking advantage of the good support in Java and C# for multi-threading. Current work on translation to Java has focused on linear logic, so there is scope for new work on other extensions of Prolog. GNU Prolog[14], already implements many extensions to the language, which could be considered.

• Investigating the relative merits of retaining support for linear logic.

3.3 Timetable

I plan to start the second year by investigating the internals of Prolog Café. Prolog Café is written in Java (the run-time system) and Prolog (the actual translator). The translator itself is 2,500 lines of SICStus Prolog.

Having done so, I would hope to obtain at least a naïve translator to C# and hopefully one which is to some extent optimised to match C# by the end of that year.

Time permitting after that, I would like to consider further optimisations, and/or extensions to Prolog. This can only be done after a working translator has been obtained.

Ideally work would be done on producing more readable code, which may be quite theoretical in nature. However, very little progress might be made on this aspect of the project due to its ambitious nature. I nevertheless hope that some advance might be made in this direction. This part of the project should guide the design decisions made during implementation, perhaps from an early stage.

3.4 Thesis Chapters

Below, I summarise an estimate of the chapter structure of the thesis and what each chapter may contain.

- Introduction : introduction to language translation issues
- Language Interoperability : survey of existing language translation and .NET
- Prolog and Linear Logic
- Prolog Café
- Producing Naïve C# : minimal modifications of Prolog Café to produce C# instead of Java
- Optimising the Design : indirect jumps
- Producing More Readable Code
- Implementing Concurrent Prolog
- Implementing other Prolog Extensions
- Conclusion

References

[1] Aït-Kaci, H., (1999) Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (out of print) Available from `http://www.isg.sfu.ca/~hak`

[2] Aït-Kaci, H., (1991) Warren's Abstract Machine: A Tutorial Reconstruction. Slides for ICLP'91 Pre-Conference Tutorial. Available from `http://www.isg.sfu.ca/~hak`

[3] Albahrari, B. A Comparative Overview of C#. Available from `http://genamics.com/developer/csharp_comparative.htm`

[4] Banbara, M., Tamura, N., (1999) Translating a Linear Logic Programming Language into Java. In Proceedings of ICLP'99 Workshop, 1999.

[5] Banbara, M., Tamura, N., (1997) Java Implementation of a Linear Logic Programming Language. In Proceedings of the 10th Exhibition and Symposium on Industrial Applications of Prolog, pp. 56-63.

[6] Banbara, M., Tamura, N., (1998) Compiling Resources in a Linear Logic Programming Language. In Proceedings of Post-JICSLP'98 Workshop on Parallelism and Implementation Technology for Logic Programming Languages.

[7] Benton, N., Kennedy, A., (1999) Interlanguage Working Without Tears: Blending SML with Java In ICFP'99, pp. 126–137.

[8] Benton, N., Kennedy, A., Russel, G., (1998) Compiling Standard ML to Java Bytecodes. In Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming, September 1998, Baltimore.

[9] Campell, J.A., (1984) Implementations of PROLOG. Ellis Horwood.

[10] Clocksin, W. F., (1997) Clause and Effect. Springer-Verlag Berlin and Heidelberg GmbH & Co. KG

[11] Codognet, P., Diaz, D., wamcc: Compiling Prolog to C. In PLILP'95.

[12] Girard, J.-Y. (1987) Linear Logic. Theoretical Computer Science, 50: 1–102.

[13] Girard, J.-Y. (1995) Linear Logic: Its Syntax and Semantics. In Girard, J.-Y., Lafont, Y. and Regnier, L., editors, Advances in Linear Logic. pp. 1–42. Cambridge University Press.

[14] GNU Prolog home page. `http://pauillac.inria.fr/~diaz/gnu-prolog/`

[15] Gordon, A.D., Syme, D. (2001) Typing a Multi-Language Intermediate Code. The 28th ACM Symposium on Principles of Programming Languages.

[16] Gosling, J., Joy, B., Steele, G., Brancha, G., (2000) The Java Language Specification, Second Edition. Addison Wesley.

[17] Hans, W. (1992) A Complete Indexing Scheme for WAM-Based Abstract Machines. In Proceedings of the 4th International Symposium, PLILP '92, August 1992, Leuven, Belgium. LNCS 631 pp.232–244.

[18] Hodas, J. H., Watkins, K., Tamura, N., and Kang, K.-S. (1998) Efficient Implementation of a Linear Logic Programming Language. In Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming.

[19] jprolog home page. `http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/`

[20] Krall, A., Berger, T. (1992) Fast Prolog with a VAM_1p based Prolog Compiler In Proceedings of the 4th International Symposium, PLILP '92, August 1992, Leuven, Belgium. LNCS 631 pp.245–259.

[21] Krall, A., Neumerkel, U. (1990) The Vienna Abstract Machine. In PLILP'90.

[22] Li, X. (1996) Program Sharing: A New Implementation Approach for Prolog. In Proceedings of the 8th International Symposium, PLILP '96, September 1996, Aachen, Germany. LNCS 1140 pp. 259–273.

[23] Lindholm, T. and Yellin, F. (1999) The Java Virtual Machine Specification, Second Edition. Addison Wesley Longman Inc.

[24] MINERVA home page: `http://www.ifcomputer.com/MINERVA/home_en.html`

[25] The Microsoft Developer .NET home page. `http://msdn.microsoft.com/net`

[26] Proebsting, T.A., Townsend, G., Bridges, P., Hartman, J.H., Newsham, T., Watterson, S. A. (1997) Toba: Java For Applications—A Way Ahead of Time (WAT) Compiler. Technical Report, Dept. of Computer Science, University of Arizona, Tucson.

[27] Pusch, C., (1996) Verification of Compiler Correctness for the WAM. In Proceedings of the 29th International Conference, TPHOLs'96, August 1996, Turku, Finland. LNCS 1125 pp.347–361.

[28] Stroustrup, B., (2000) The C++ Programming Language, Special Edition. Addison Wesley.

[29] Tamura, N., and Kaneda, Y. (1996) Extension of WAM for a linear logic programming language. In Ida, T., Ohori, A. and Takeichi, M. editors, Second Fuji International Workshop on Functional and Logic Programming. pp. 33-50. World Scientific.

[30] Tarau, P. and Boyer. M. (1990) Elementary Logic Programs. In Deransart, P. and Maluszyński, J., editors, Proceedings of Programming Language Implementation and Logic Programming, LNCS 456, 159–173, Springer.

[31] Tarditi, D., Lee, P., Acharya, A. (1992) No Assembly Required: Compiling Standard ML to C. ACM Letters on Programming Languages and Systems 2(1) 161–177.

[32] Warren, D. H. D, (1983) An Abstract Prolog Instruction Set. Technical note 309, SRI International, Menlo Park, CA, October 1983.

[33] Warren, D. H. D, (1988) Implementation of Prolog. Lecture notes, Tutorial No. 3, 5th International Conference and Symposium on Logic Programming, Seattle, WA, August 1988.