Language Interoperability and Logic Programming Languages

Jonathan J. Cook

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2004

Abstract

We discuss P#, our implementation of a tool which allows interoperation between a concurrent superset of the Prolog programming language and C#. This enables Prolog to be used as a native implementation language for Microsoft's .NET platform. P# compiles a linear logic extension of Prolog to C# source code. We can thus create C# objects from Prolog and use C#'s graphical, networking and other libraries. P# was developed from a modified port of the Prolog to Java translator, Prolog Café.

We add language constructs on the Prolog side which allow concurrent Prolog code to be written. We add a primitive predicate which evaluates a Prolog structure on a newly forked thread. Communication between threads is based on the unification of variables contained in such a structure. It is also possible for threads to communicate through a globally accessible table. All of the new features are available to the programmer through new built-in Prolog predicates.

We present three case studies. The first is an application which allows several users to modify a database. The users are able to disconnect from the database and to modify their own copies of the data before reconnecting. On reconnecting, conflicts must be resolved. The second is an object-oriented assistant, which allows the user to query the contents of a C# namespace or Java package. The third is a tool which allows a user to interact with a graphical display of the inheritance tree.

Finally, we optimize P#'s runtime speed by translating some Prolog predicates into more idiomatic C# code than is produced by a naïve port of Prolog Café. This is achieved by observing that semi-deterministic predicates (being those which always either fail or succeed with exactly one solution) that only call other semi-deterministic predicates enjoy relatively simple control flow. We make use of the fact that Prolog programs often contain predicates which operate as functions, and that such predicates are usually semi-deterministic.

Acknowledgements

I would like to acknowledge the kind advice and assistance of my supervisor, Stephen Gilmore, who has read through and suggested improvements to many versions of this thesis, my publications and other documents. Stephen has met with me almost every week during my Ph.D. and offered a great deal of good advice and support throughout my time in Edinburgh.

The helpful comments of the anonymous referees on all the papers that I've submitted during my time at Edinburgh and in particular those which are contained in this thesis, are gratefully appreciated.

I would also like to acknowledge the helpful advice of my second supervisor Ian Stark and my other progress review panel members, David Aspinall and Don Sannella.

I would like to acknowledge Mutsunori Banbara and Naoyuki Tamura, the developers of Prolog Café, the tool on which P# is based, and without whom P# would not exist.

I thank the flat mates I have lived with during my time in Edinburgh: Laura Wisewell, Jeremy Brookman, Andrea Greve, Markus Edelbluth, and Jeroen van Bergeijk; and the office mates I have worked along side: Sam Lindley, Miki Tanaka, Shin-ya Katsumata and Hongqian Liang, for making my four years here so enjoyable.

I thank Douglas Blackwood, Sean Cross, Jonathan Crowson and Paul McAllister for their support during my time in Edinburgh.

During the first three years of my PhD, I had financial support from the EPSRC.

Finally, I thank my parents and sister for their love, and good advice.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Almost all of the text of chapters 3 to 4 and some of the text of chapter 5 of this thesis have appeared as the paper [Coo04a]. A reduced form of chapter 6 has appeared as the paper [Coo04b]. Chapters 1 and 7 both contain elements from both of these papers.

(*Jonathan J. Cook*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is widely believed that the declarative programming language paradigm is particularly helpful in developing succinct and correct programs that are easy to understand. It is also argued that it is easy to find and correct errors in a program written in such a language. Similarly, it is believed by many that modern object-oriented languages such as Java [GJSB00] and C# [Cor01] also have these advantages. Programming languages are sometimes chosen for a project because, when executed, programs written in those languages are more efficient in terms of time or space. Languages are also chosen because they have well developed libraries. In many cases, however, a great gain in productivity is achieved by using a language better suited to the problem to be solved. This is particularly the case when such a language supports interoperation with other languages that may provide features that it lacks, or that may be useful for solving other parts of the problem to be solved.

Declarative programming languages allow programs to be written that are modelled more on the problem that the program is to solve, than on the steps by which that problem is to be solved. The logic programming language Prolog [CM94], being a declarative language, has the advantages stated above, although it is an untyped language and the correctness of programs relies to some extent on the programmer not making mistakes such as using inconsistent naming, as such errors are not always detected by Prolog compilers. This is in contrast to a language such as SML [RMH90],

which is strongly typed and it is argued by some that once an SML program type checks it is likely to be correct. In this case, the programmer's effort can be expended more on getting the program to type-check than on writing functions which model a problem. If correctness is our primary concern, this effort is well rewarded. There are many problems for which a declarative language such as Prolog or SML is far better suited than an imperative language.

Imperative languages, which tend to be more widely used than declarative languages in commercial settings, have more of an emphasis on specifying the steps by which a problem should be solved. With an imperative language it is arguably easier for a programmer to make errors in their code which are less likely to be discovered during the development of the program and more likely to manifest themselves in a potentially more destructive manner when the program has been deployed to a user. This is because the question of how to solve a problem is one step removed from the problem itself, and errors can be introduced in making this step. Programs written in imperative languages tend to be more efficient than those written in declarative languages.

Modern object-oriented languages are usually imperative languages, however the object-oriented features of these languages are another approach towards representing problems rather than the steps used to solve them. Indeed, many programmers prefer and find it easier to program in such languages than in declarative languages. This could be because the imperative style of programming is more accessible because it is more concrete. Modern object-oriented imperative languages tend to have extensive support for Graphical User Interface (GUI) programming and networking.

For an implementation of a programming language to be successful it is important that it provides access to the best features of the platform or platforms on which it runs. The practicality of using a declarative language can often be significantly enhanced by providing well designed mechanisms that allow it to interoperate with other, possibly more widely used, languages. Virtual Machine (VM) based execution environments centred on a modern object-oriented imperative language, such as the Java platform [GJSB00, LY99] and Microsoft's .NET Platform [Pla02, NET, TL01] are becoming increasingly popular. Such environments are often used as multiple language platforms

because this allows many languages to interoperate with the platform's core language and other languages. Thus, it is desirable to implement declarative languages on these platforms. This allows the different programming language paradigms to interoperate and allows us to 'get the best of both worlds'.

In the remainder of the introduction we will first discuss how language interoperation can be achieved by compiling source code written in one language to source code expressed in another language. We will then discuss logic programming languages, modern Virtual Machine platforms and ways of allowing languages to interoperate other than by source-to-source translation. The final section of the introduction discusses our contribution to improving language interoperation for the logic programming language, Prolog.

1.1 Source-to-Source Language Translation

There are now many programming languages available and often it may be convenient to use a number of languages in one project. This raises the issue of how the languages are to interoperate. Language translation is one way of achieving language interoperation. If we translate source code to source code the final program can be expressed in a single language, although it has been written in many. This can make compilation easier than it is when developers use tools that allow one language to call another language. It can also promote a closer integration between the languages involved than is achieved with other forms of interoperation between languages.

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 with 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 can 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.

Generally, however, automatic translators produce code which is difficult for a human to understand and modify. Exceptions to this are Microsoft's Java to C# converter: the Java Language Conversion Assistant and the Octopus .NET Translator [Oct]. Octopus can perform some of the source-to-source translations between Java, C#, native C++, Visual C++.NET and Visual Basic.NET. This is made possible by the similarity between the languages.

We would like such a compiler to generate code that executes efficiently and uses the target language's constructs idiomatically. We would also like the tool to fully exploit the features of the target language. The ideal would be to develop a tool that produces code that is readable, well-structured and can easily be modified by a human. Fully realizing 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.

Many commercial language translation projects have involved the translation of legacy code in an older language such as COBOL or PL/I to a newer language. Older languages are still widely used, but there is a lack of skilled programmers in these languages. Thus, it is desirable to translate these languages into languages such as C++, Java and C#. Once the code is translated, there is a need to maintain and modify the new code. If the translator is reliable and does not need much human intervention, it is acceptable to modify the original code and re-translate it. However, it is preferable to be able to modify the translated code. Thus, we would like for it to be easy to read and to modify the translated code without inadvertently introducing errors into the code.

Many such projects have ended in failure and financial loss. The paper [TV00] argues that this is because conversions between programming languages are inherently difficult. Many software managers are fooled into thinking that a change of language can solve all of their problems. On hearing the name 'COBOL to Java' and without looking more deeply into the implications of language translation it is easy to assume

that such a change will lead to a significant gain in productivity. A loose chain of reasoning is followed where, given the facts that Java code is high-quality code and that the COBOL to Java translator generates Java, it is inferred that the COBOL to Java translator generates high-quality code. The phenomenon of being convinced by a name, rather than by the reality for which that name is a label is known as *name magic*.

The difficulties of source-to-source translation are manifold. Even integral data types cause problems. Some of the programming languages in which legacy software was written have, by modern standards, bizarre means of representing a fixed range of integers and equally bizarre exact representations or approximations to real numbers. Thus, it can be nearly impossible to translate code using such types without resorting to, usually inefficient, code which simulates the operations of the older language's types. When translating between two more modern languages, differences in the size and precision of numeric types can cause problems. The size of numeric data types in the language C is platform dependent, whereas with Java it is platform independent, for example. Another problem is dealing with idiosyncratic library functions. Many such functions take or return special values to indicate exceptional circumstances, and the values used in such cases differ from language to language. For example, the `java.lang.String.indexOf( int c )` method of Java, which finds the location of a character in a string returns 0 to indicate the first location of the string and -1 to indicate that the character does not occur in the string. Turbo Pascal's analogous procedure `pos` returns 1 to indicate the first location and 0 to indicate that the character does not occur. Translations of strings between languages can also cause problems because languages have differing conventions on issues such as escape characters, termination and indexing.

The paper [TV00] gives an example of a process for language conversion. This involves restructuring the original program to make it suitable for a *syntax swap*. A syntax swap involves merely exchanging syntactical constructs in the original language with those of the new language. The resultant code is then restructured to produce code which is more readable to make future maintenance as easy as possible. This process is usually difficult and involves many compromises.

1.2 Logic Programming Languages

1.2.1 Prolog

Prolog [CM94] is the seminal logic programming language and is well suited to artificial intelligence programming. To write a program, say, to find a means of winning a game can be far more natural in Prolog than in 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 more efficient. In many applications we might be prepared to sacrifice speed in order to have a clear, concise program. Prolog is not ideally suited to numerical computation, the most obvious example of a field of computing that usually requires an efficient language. However sometimes, expert systems for example, may be required to draw logical inferences from a large amount of data; or to solve a problem in which a large number of cases may have to be considered. In such cases it would be desirable for Prolog to be faster. Unfortunately attempts to translate Prolog to languages such as C have failed to yield programs which run faster than the corresponding Prolog program run on an efficient Prolog implementation. Other benefits may be gained, though, such as portability and the ability to easily integrate Prolog with programs written in the faster languages.

A *Prolog term* is an atom, an integer term, a floating point number term, a list term, a structure term or a variable term. Structure terms and list terms are collectively known as *compound terms*. An *atom* is represented by a string of characters. If an atom begins with a capital letter or contains symbols with other meaning in Prolog, then it must be typed surrounded by single quotes. Examples of atoms are `yes`, `abcdef`, and `'Atom_1'`. *Integer terms* represent integers, and *float terms* represent floating point numbers. A list can either be the empty list, written `[]`, or a head together with a tail, written `[Head|Tail]` or as a list of items, such as `[1,2,3]`, which is equivalent to `[1|[2|[3|[]]]]`. A *variable term* can be regarded as a hole which

can at some point be instantiated to one of the other kinds of term. The names of variable terms start with a capital letter. A structure consists of a *functor*, which is an atom associated with an arity, and a number of arguments, for example the structure `foo(1,[2|T],T)` has the functor `foo/3` and three arguments of which the first is the integer 1, the second is a list whose first element is 2 and whose remaining elements are the same as the third argument. In fact, Prolog is an untyped language and thus a term such as `foo([],3,[1,[2,3],4])`, containing a heterogeneous list, is also a valid term. The dot functor, `./2` can be used to represent lists as structures, for example `[1,2,3]` is equivalent to `.(1,.(2,.(3,[])))`.

A *fact* is a structure without any variables. For example `married('John','Lucy').` is a fact which might represent the fact that John and Lucy are married. A *clause* consists of a head and a body separated by the symbol `:-`. A *goal* consists of a structure, which is intended to be executed. The *head* consists of a structure, and the *body* of a conjunction of goals separated by commas. An example of a clause is

```
sibling(X,Y):- parent(Z,X), parent(Z,Y).
```

This is read as: `X` is a sibling of `Y` if `Z` is a parent of `X` and `Z` is a parent of `Y` for some `Z`.

Two atoms unify if they are the same atom. Two integer terms unify if they are the same integer, and similarly for floating point numbers. Two list terms unify if they are both the empty list; or their heads unify and their tails unify. Two structures unify if they have the same functor and arity and the arguments respectively unify. If a variable term is unified with another variable term, then they effectively become the same variable. If a variable term is unified with another type of term, then that variable becomes instantiated to that term.

A *predicate* is defined by a number of clauses all having the same head functor. When the predicate is called, all of those head goals which unify with the calling goal have their bodies executed. The goals are executed in a depth-first manner. This is referred to as *backtracking*. A *cut* is a goal indicated by a `!`. If a cut is encountered there will be no further backtracking during the current predicate call. The cut may indicate that the single solution has been found or may be used as a way of controlling the flow of

control through the predicate.

A Prolog program consists of a set of predicate definitions. The program is executed by issuing a query. A *query* is a goal which is typed in at the top-level prompt.

A naïve Prolog program for finding the length of a list is as follows:

```
length( [], 0 ).
length( [_|T], N ) :- length( T, L ), N is L + 1.
```

This problem can be solved more efficiently by using the *tail-recursive* idiom, where the recursive call occurs as the final goal in the body as follows:

```
length( List, Length ) :-
    len( List, 0, Length ).

len( [], N, N ).
len( [_|T], N, Length ) :-
    N1 is N + 1,
    len( T, N1, Length ).
```

Here, the second argument of `len/3` is an *accumulator* to which the result is instantiated when the base case is reached.

Failure driven loops are examples of another Prolog idiom. They are best illustrated via the built-in `repeat/0` predicate which is defined by:

```
repeat.
repeat :- repeat.
```

This predicate is used in the following way:

```
repeat,
    ⟨code, C, to be repeated, this must always succeed⟩
    ⟨a goal, G, which fails when the loop is to be continued⟩
!.
```

The code block, *C*, is repeatedly executed. On each iteration if the loop is to be continued the goal, *G*, fails causing backtracking to the call to the `repeat/0` predicate.

When *G* succeeds, it is immediately followed by a cut to prevent further backtracking, and thus to prevent the next failure from causing the loop to be iterated further. One limitation of this idiom is that cuts cannot be used in the code block, *C*.

Prolog is equipped with a database feature, which allows terms to be asserted (added) to the database, retracted (removed) from the database and called (retrieved) from the database. With most Prolog implementations it is even possible to modify in this way the clauses of the running program, and thereby modify the behaviour of the program dynamically.

For more information on the Prolog language, see [CM94].

1.2.2 Concurrent Prologs

Concurrent Prolog derivatives can be classified into those with explicit concurrency constructs and those with implicit parallelism. With explicit concurrency constructs, the programmer must direct how the concurrency is to be exploited. With implicit parallelism the concurrency is exploited automatically by the evaluator. The survey paper [Cia92] gives examples of both types of language.

Explicitly concurrent languages can be divided into those that have explicit message passing primitives, for example DeltaProlog; those that have a shared blackboard for communication; and those that make use of guards or committed choice, for example Parlog [Gre87]. The concurrent logic languages Parlog, Guarded Horn Clauses and Concurrent Prolog are discussed in Part I of [Sha87].

DeltaProlog [PN84] is an extension of Prolog based on CSP [Hoa85]. And-parallelism is achieved via a ‘fork goal’. Communication is via special event goals, which come in several different flavours depending on whether the goal is backtrackable and whether it is synchronous.

SICStus MT [EC98] is an multithreaded extension of SICStus Prolog. SICStus MT has a predicate that spawns a thread, named `spawn/2`, Messages can be sent from one thread to another thread, by specifying in the call to the `send/2` predicate the desti-

nation thread's identifier. This message can then be received by calling `receive/1` on the thread that is the destination of the message. The other primitives provided include the `self/1` predicate that returns an identifier for the currently executing thread, a predicate that waits for a period of time, and a predicate that kills a thread.

Jinni [Jin] also has support for concurrency, based on Linda blackboards. Terms can be read from and written to the shared blackboard.

Languages that have implicit parallelism can be divided into or-parallel languages, such as Aurora, independent and-parallel languages such as &-Prolog [HG91] and dependent and-parallel languages such as Andorra, the latter being related to committed choice languages. [GPA+01], is a recent survey of languages with implicit parallelism.

Aurora [LBD+88] is an or-parallel Prolog, well suited to the parallel programming of parallel processors. This means that a predicate which is marked as parallel can have its clauses evaluated in parallel. Thus parallelism in Aurora is implicit, in that the user does not specify the details of parallel evaluation.

The following is an example of parallel programming in Aurora taken from [Cia92].

```
:- parallel select/3

select( X, [X|Xs], Xs ).
select( X, [Y|Ys], [Y|Zs] ) :-
    select( X, Ys, Zs ).
```

The solutions, `Z`, of the query `select( X, L, Z )` are all those lists which are the list `L` with one occurrence of the element `X` removed. This definition is the same as a standard definition of this predicate with the exception of the line:

```
:- parallel select/3
```

which specifies that the clauses can be evaluated in parallel.

Aurora provides two types of database modification primitive. With one, the alteration to the database can occur concurrently, asynchronously, anywhere in the proof tree. The other blocks until it is in the leftmost branch of the search tree, adding some degree

of determinism. In Aurora, and also the language MUSE, this behaviour is extended to other extra-logical predicates allowing these languages to mimic the semantics of sequential Prolog.

The parallel Prolog language &-Prolog automatically parallelizes standard Prolog code. However, it also allows explicit parallelism. A conjunction of goals can be executed in parallel by separating them with the `&` operator. In addition &-Prolog has predicates that wait for variables and acquire and release locks on terms. The &-Prolog language is an independent and-parallel language. This means that when conjoined goals are evaluated in parallel there can be no variable conflicts. Both strict and non-strict independent and-parallelism are supported. With strict and-parallelism goals performed in parallel do not share variables. Built-in predicates are provided that can be used to control the use of memory and multiple processors. A construct called the Conditional Graph Expression (CGE) is added to the language. This allows a set of goals to be either executed in parallel or sequentially depending on whether a given condition succeeds or fails. The source code is translated by the compiler into PWAM (Parallel Warren Abstract Machine) code. The run-time system consists of one or more PWAMs executing in parallel.

Flat Concurrent Prolog (FCP) [Mie84] is a stream-based parallel logic language. Communication is via shared variables. Each clause is guarded by goals. The clause that is executed is one of those whose guard succeeds. If more than one succeeds, any of those may be executed. This is referred to as ‘don’t care non-determinism’. The guards may only contain predefined predicates, hence ‘flat’, for efficiency reasons.

FCP is and-parallel. This means that conjoined goals in a clause can be evaluated in parallel. Communication via variables is facilitated by the ability to mark variables as read only with a question mark: such a variable will await an instantiation from another goal. Thus a producer can be coupled with a consumer as follows, see [Cia92]:

```
?- prod( X ), cons( X? ).
```

FCP is very unlike Prolog. Indeed it is non-trivial even to simulate Prolog programs in FCP.

FCP, Parlog and FGHC (Flat Guarded Horn Clauses) are examples of committed choice languages [Tic95]. In general, programs written in these languages consist of guarded Horn clauses. A guarded Horn clause consists of a set of ask guard goals, tell guard goals and body goals. The difference between ask guards and tell guards is that ask guards can only match arguments, whereas tell guards can make variable bindings. The term 'committed choice' refers to these language's use of 'don't care' non-determinism.

Andorra is a dependent and-parallel Prolog derivative. Such languages tackle the problem of avoiding redundant computation when executing two goals in parallel where there is a dependency between the variables used in the two goals. This problem is solved by ensuring that one of the goals, the consumer, cannot bind the dependent variable. The other, which can, is referred to as the producer.

The Prolog system, CIAO Prolog [CIA], also has support for concurrency consisting of an extensive choice of predicates which spawn new threads and occasion cuts or backtracking on the created threads. There are also predicates for locking and unlocking on an atom and a predicate that can be used to assert that a given predicate is concurrent allowing it to be used for communication and synchronization between threads. CIAO Prolog uses a Prolog database for communication in the same way as a blackboard, an idea proposed in [CH99].

1.2.3 Linear Logic Variants

Linear logic programming is based on intuitionistic linear logic. For more information on linear logic, see [Gir87, Gir95]. The resource-conscious nature of the languages makes them idea for the solving problems involving resources, which can only be used a fixed number of times. Linear logic is a very natural logic for describing many problems, for example in the case of a the Eight Queens problem, each row and each column might be a resource.

Examples of linear logic programming languages include Lolli and LLP, which are both supersets of Prolog. LLP is a subset of Lolli.

1.3 The Java and .NET Platforms

1.3.1 The Java Platform

Implementations of the Java programming language [GJSB00] compile Java to an intermediate language called the Java Virtual Machine Language (JVML) [LY99], also known as Java byte-code, which is stored in Java class files. The JVML is either interpreted, or more usually compiled Just-In-Time (JIT) to native code by a Java Virtual Machine (JVM). This JIT compilation allows efficient code, which is tailored to the relevant processor, to be generated on almost any modern desktop computer. Using an intermediate language offers many advantages [MM]. They include the fact that the JVML is far more compact than Java source code and therefore more easily transmitted across the Internet. Now that there is an implementation of the JVM on most desktop machines, the same JVML code can be widely distributed, without the need to distribute different binaries for different platforms. Also a client computer onto which a Java class file has been downloaded is better able to enforce security requirements on the JVML code than it would on native code, as native code is usually untyped and runs without a security manager. These are also advantages in the cases of other intermediate languages. The JVML has only enough features to be an excellent target language for Java.

The JVML is a stack based language. Byte-code instructions pop items off an operand stack, operate on them and then push the result back onto the operand stack.

When a method is called a stack frame is created containing arguments to the method and local variables, amongst other data. The data types of the JVML are Boolean, byte, short, char, int, long, float, double and address. Where relevant a separate instruction is provided for each type. The type of each argument, local variable and operand stack slot at each point in the byte-code can be determined statically from the byte-code. However an individual local variable or argument may contain different types at different points during the execution of a method. Every argument, local variable and operand stack slot is 4 bytes wide: long and double values occupy two slots. This can cause difficulties for compiler developers.

Instructions are provided which perform arithmetic operations, push constants onto the stack, load from arguments, local variables or arrays onto the stack, save the top item of the stack into an argument, local variable or array, get and put static data, and rearrange or duplicate items near the top of the stack.

There are instructions which transfer control within a method based on a condition or a `switch` statement and instructions for jumping to and returning from a subroutine in the same method. Four instructions are provided for calling other methods and for most of the internal types of the JVM a return instruction is provided. In some cases one of the return instructions covers several similar types.

Other instructions create new objects or arrays, push the length of an array onto the operand stack, throw an exception, deal with casting between types, enter or exit a monitor in concurrent code and convert between types.

Garbage collection, threads and monitor locks are supported at the virtual machine level. Each Java object has a lock associated with it, allowing code to acquire or release a lock on that object.

The Java Platform is supported by mature and comprehensive libraries which have evolved over a number of years.

1.3.2 The .NET Platform

The Microsoft .NET Platform [Pla02, NET, TL01] consists of a number of technologies, including development environments, enterprise servers and operating systems. One of the principal goals of the platform is 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. In order to facilitate the writing of web services and Windows applications, the .NET Framework has been developed, which allows a number of languages to work together by compiling them all down to a common intermediate language. Arguably, XML Web services are likely to revolutionize the way users interact with applications, with applications being invoked across the Internet.

The .NET Framework is central to the .NET Platform and consists of the Common Language Runtime (CLR) and libraries which are shared between all of the languages available for the .NET Platform. These include support for Web Services, Web Forms, Windows Forms, and XML. The design goals of the .NET Framework include support for a component infrastructure which does not have the constraints nor require the excessive plumbing that COM [Box98] does. Other goals were to promote reliability, security and interoperation with the Internet. The .NET Framework tries to simplify the processes of development and deployment, and to eliminate what has become known as 'DLL Hell'. It has often occurred in the past that updating the DLL for one application on a Windows system causes another application to stop working. A Global Assembly Cache (GAC) is provided, which ensures that each application uses the DLLs with which it was built.

The CLR provides run-time support for .NET languages, and is roughly the .NET analogue of the JVM. It handles garbage collection, memory layout of data, security, exception handling and threading. The CLR's Virtual Execution Engine accepts .NET executable files, which consist of code, data and metadata. The Virtual Execution Engine consists of a class loader and JIT compiler which verifies code and then JIT compiles it, and support for executing and managing the native code that is produced by the JIT compiler.

The CLR is able to execute both managed and unmanaged code. Unmanaged code is compiled for specific hardware and is not subject to the control of runtime with regard to issues such as memory management and security. Managed code is expressed in the .NET Common Intermediate Language (CIL).

1.3.3 The .NET Common Intermediate Language (CIL)

Microsoft's .NET Framework allows multiple languages to interoperate by compiling them to the .NET Common Intermediate Language (CIL) also known as MSIL. This is similar to the JVML, with the Common Language Infrastructure (CLI) being anal-

ogous to the JVM. [1] However, the CLI has a number of additional features [MM], in order to extend the class of languages which can be effectively compiled to the intermediate language to a wide range encompassing almost all language paradigms. Whereas Java byte-code is sometimes interpreted, CIL is designed to be always compiled to native code before being run.

There are now additional advantages of using an intermediate language to those mentioned above. Implementing ℓ languages on p platforms requires $\ell + p$ compilers when an intermediate language is used. Otherwise $\ell \times p$ compilers are required. Also the languages that are compiled to CIL can more easily interoperate with one another than can binaries.

Weaknesses of the JVML as an intermediate language for languages other than Java include the lack of support for type-unsafe features, such as pointers. Unboxed structures and unions are required for languages such as Pascal. Tail calls are required for functional languages where recursion is the only form of repetition. Variable length argument lists and function pointers are required by C. Some languages also require overflow sensitive integer arithmetic. Whereas the JVML supports none of these, the CIL supports them all.

However, it is difficult to get an intermediate language intended to be a target for many languages right first time, and the CIL is certainly not perfect. For example, when it became desirable to extend C# to generic C#, this necessitated changes to the .NET Platform virtual machine itself. Indeed, it is evident that Virtual Machines are complex pieces of software and improving on them or modifying them [Coo02] is not a trivial task.

The CIL has an evaluation stack, the analogue of the JVML's operand stack. Code is organized into namespaces (similar to Java packages) which are composed of classes. These are themselves composed of methods. Unlike Java, whose methods cannot be longer than 65,535 bytes, CLI methods have unlimited length. With the CLI, unlike the JVM, an individual local variable has the same type throughout a method call. Also

[1] The CLI is a subset of the CLR, which has been submitted to ECMA, to assist in the implementation of the CLR for platforms other than Windows.

with the CLI, arguments and local variables are separated into two zero-based arrays; and arguments, local variables and evaluation stack slots may have any length.

Central to language integration in the .NET Framework is the Common Type System (CTS). This is a type system which every .NET language much adhere to. The types available are divided into value types, which include primitive types and user-defined value types; and reference types, such as objects. The primitive types of the CLI include objects, strings, Booleans, integers of various sizes, unsigned integers of various sizes and machine dependent native integers, single and double precision floating point values and typed references. A typed reference is composed of a pointer and a type.

Type constructors are available which convert a type reference into a value type, class or array. Also function pointers can be generated.

Many of the CLI's instructions are similar to JVML instructions. A major difference is that CIL instructions only need their return type hard-coded into them, instead of having their argument types hard coded as with the JVM. This shifts some burden from compilation that targets the intermediate language, to JIT compilation. Other instructions which have no JVML counterpart include those which allow boxing, unboxing, tail-calls, call by reference, and the invoking of a function pointer.

All CIL code falls into one of four categories. Invalid CIL is CIL that cannot be compiled to native code by a JIT compiler. Valid CIL can be compiled to native code and may use facilities such as type-unsafe pointer arithmetic. Type-safe CIL is CIL which respects the accessibility constraints publicly provided by code with which is interacts. Verifiable CIL is both type-safe and can be proved to be type-safe using a specified algorithm.

1.3.4 C#

The principal .NET language is C# [Cor01, Lib01, Alb]. Thus, C# is an important language. C# and .NET are each to some extent designed to work well with the other.

C# is a relatively new object-oriented programming language which has drawn on the

languages C++ [Str00] and Java [GJSB00, LY99]. Many features not present in Java have been added. Some of these features are intended to allow the writing of efficient code, and 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 the essential points of C# are summarized.

C# shares certain features of Java not found in C++, such as garbage collection, reflection, thread support, static 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. Unlike Java, if a method is to be overridden it must be explicitly declared as **`virtual`** and the overriding method must explicitly be declared as overriding. It is even possible to hide inherited methods instead of overriding them, in much the same way that in Java class instance fields are hidden by declaring a field with the same name as an inherited field. C# supports Java-style interfaces, and abstract methods, and like Java does not allow multiple inheritance except for interfaces. The support for multi-threading is similar to that of Java, being based on locked regions and monitors. The support for reflection goes so far as to make it possible to emit intermediate language instructions on the fly and therefore dynamically construct methods.

C# also has features of C++ not found in Java, such as operator overloading, namespaces, jumps, **`enum`**s, preprocessing directives 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`**. It is necessary to 'pin' objects which are accessed via a pointer, using the **`fixed`** keyword, to ensure that the garbage collector does not try to move them while they are being accessed. C#'s operator set is very similar to that of C++, however fewer of the operators can be overloaded.

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', that is every array is an object. **`new int`**`[2][3][5]` creates 1+2+6 arrays. Secondly, one can define C++ style rectangular arrays, in this case

the code, **`new int`**`[2, 3, 5]`, creates 1 array. In addition, C++ style **`struct`**s are available which can be used instead of objects in order to obtain more efficient code. The **`struct`**s can encapsulate methods, but have no concept of inheritance. Value types, which include **`struct`**s, are allocated either on the stack or inline, for example as a field of an object. Objects are allocated space on the heap. It is possible to allocate an array of value types on the stack using the **`stackalloc`** keyword. The **`struct`** concept is an integral part of C#'s type system. The primitive types are all really **`struct`**s, although the compiler is able to give them special treatment to avoid a performance penalty.

Function parameters can be passed by reference, and the formal arguments of a method may end with an array of any type (with the keyword **`params`**) so that any number of parameters of that type can be used at the end of the actual arguments. This allows C# to have a `WriteLine()` method similar to C's `printf()`, as even integers, being **`struct`**s, can be regarded as **`object`**s.

Event handling is implemented using delegates, a concept described in [Alb] 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. If a delegate is declared to store methods which do not return a value, it is referred to as a multicast delegate, and can hold more than one method (of the specified type). When such a delegate is invoked all the methods it contains are invoked in the order that they were added. The use of delegates is not, however, restricted to event handling.

C# has extensive library support for XML and regular expressions. It also shares with Java support for networking, something that can be a difficulty when using C++.

C# has also learnt some lessons from Visual Basic, for example the **`foreach`** statement, which iterates through an array or collection.

Finally various syntactical concepts are added such as properties (getters and setters) and indexers for treating objects as arrays.

1.4 Other Forms of Language Interoperation

Language designers and developers should consider language interoperation carefully. There are various ways in which languages can talk to one another other than by source-to-source translation. One is to use foreign language interfaces. These often result in code which is ugly and difficult to maintain. Many languages have foreign language interfaces which enable them to interoperate with C. For example, Java has the Java Native Interface (JNI) [Gor98] for this purpose.

Another approach that can be used in some instances is to compile from one language to the Virtual Machine intermediate language of another language. An example of this is MLj [MLj, BK99, BKR99], which is a tool that compiles SML [RMH90] to the JVML. The Jython language [Jyt] is implemented as a compiler from Python to the JVML. The Kawa Scheme system [Kaw] is a implementation of the Scheme LISP dialect which is written in Java and compiles Scheme [Dyb96] to the JVML.

The SML.NET [SML, BKR04] compiler compiles SML to the .NET intermediate language, CIL, allowing interoperation between SML and the other languages available on the .NET Platform. The Scala [OCRZ03] language, which is described as a fusion between the object-oriented and functional programming paradigms is implemented for both the Java and the .NET Platforms. The Mondrian language [Mon] is a lazy functional language designed for interoperation with object-oriented languages. It is implemented as a compiler to the CIL.

Many other languages are supported under .NET, most of which are implemented by direct compilation to the CIL. Microsoft's Visual Studio Integrated Development Environment supports by default the languages C#, managed C++ and Visual Basic.NET. Managed C++ is a variant of C++ with added keywords which allow the C++ programmer to exploit .NET facilities such as garbage collection. Mercury is a functional logic language available on the .NET Platform. Other, third party, .NET languages [.NE] include APL, Cobol, Eiffel, Forth, Fortran, Haskell, Pascal, Perl and Scheme among others. This thesis describes the addition of Prolog to this list.

Microsoft's Common Object Model (COM) [Box98] and the Common Object Request

Broker Architecture (CORBA) [Pop97] provide another method of language interoperation. A program written in, say, Haskell, can be expressed as a COM or CORBA object, and then this program is able to interoperate with code in other languages expressed as such an object.

The XML is an example of a protocol that allows interoperation between languages by specifying a common format for the representation of data when it is passed between the languages.

1.4.1 Foreign Language Interfaces for Prolog

There are many instances where a C library call is required in a Prolog application. An implementation of the Prolog language which lacks such a facility is likely to be little used. In almost all cases the internal representation of Prolog terms, even integers, is incompatible with the types of the language with which it is to interoperate by a foreign language interface. Code is required to perform necessary conversions when control passes from Prolog to the other language and back again. Also naming conventions are required so that identifiers in one language can be referred to from the other language. Difficulties arise from the fact that the language with which Prolog is to interoperate almost always lacks control structures that are integral to Prolog, for example backtracking. In the case of exceptions, both C++ and many Prolog implementations have such a facility, but the details are different, so that, for example, it is not possible for Prolog to catch an exception thrown by a call to a C++ function.

There are several ways in which code, in C say, can be linked to a Prolog program. If, as is usually the case, the Prolog engine is written in C, it can be combined with the required C code and recompiled. Alternatively, the engine and the additional C code can reside in separate object files which are linked together. A final possibility is that the Prolog engine dynamically loads C objects.

The paper [BC02] discusses these issues and details the foreign language capabilities of several Prolog systems.

1.5 P#: A Concurrent Prolog for the .NET Platform

Microsoft's .NET Platform [NET] offers an unparalleled opportunity to build systems based on a number of interoperating languages. Logic programming is currently underrepresented on the .NET Platform, with no direct support for the seminal logic programming language, Prolog. We add support for Prolog to .NET by translating it to the core .NET language, C#. By translating Prolog to C# we gain the ability to interoperate with C# and hence with the other languages available on the .NET Platform.

There already exist translators that translate from Prolog to C, see [DM94], for example GNU Prolog [GNU, CD95], Janus and Erlang. GNU Prolog was formerly known as wamcc. There are also many Prolog tools that are based on Java, such as Prolog Café [BT97, BT98, BT99, HWTK98, PC], BinProlog [Bin], Jinni [Jin], B-Prolog [BPr] and the commercial product MINERVA [MIN]. The Prolog to C translators have an emphasis on efficiency that leads them to produce unnatural code, in the case of GNU Prolog involving jumps into functions. However, Java is more restrictive in the way in which flow of control can be programmed, for example it does not have a `goto` construct, and so the output of Prolog Café is more readable than that of the Prolog to C translators. As a consequence of the code being better structured, some runtime efficiency is lost, but the ability to easily use Java's libraries from Prolog is gained. With Prolog Café, Prolog code and Java code interoperate in much the same way as any other foreign language interface [BC02]. However, the integration is closer and more easily programmed than with, for example, a C to Java foreign language interface since in Prolog Café all of the Prolog types are internally represented as Java objects.

C# attempts to combine the efficiency of C++ with the elegance and simplicity of Java. So it seems natural to modify existing translators to translate from Prolog to C#. We hoped, in this way, to find a compromise between speed and readability which would produce reasonably efficient, well-structured code.

Translating Prolog to C# provides a means of using Prolog within the .NET Framework, as Prolog can then be translated first to C# and then to the .NET Intermediate Language, CIL. This enables us to take advantage of the close relationship between C#

and .NET in a way that would not be possible if, for example, we used GNU Prolog to translate Prolog to CIL via C.

Another way in which Prolog could be used within the .NET Framework is by translating Prolog directly to CIL. However, if we translate through C#, the C# compiler will do much of the optimization for us, and few languages, if any, are in a better position to produce well optimized CIL code than C#. This provides a strong incentive for generating code which is as close as possible to code written by programmers—the C# compiler should be better at optimizing this code than at optimizing machine-generated code that is less idiomatic.

We developed P# [P#] (pronounced 'P sharp') by porting and extending the Prolog to Java translator, Prolog Café. We found that there was scope for interesting work on implementing for P# some of the many existing extensions of Prolog. In particular it is useful to add support for concurrency by taking advantage of the multi-threading constructs that C# shares with Java. Most Prolog implementations are built on languages that do not have as good support for concurrency as C#. In adding a form of concurrency to P# we wished to choose a design that would focus on interoperation with C#. In designing our language features, we drew inspiration from existing concurrent versions of Prolog, such as DeltaProlog [PN84] and FCP [Mie84]. We wanted the concurrency to be explicit, with the programmer explicitly stating in the Prolog source code where it is to be used. In general we did not want to add features in such a way as to make it difficult for programmers with experience of just the core of Prolog to use P#.

We did not want any non-concurrent P# Prolog program to be broken, provided that it did not happen to use predicate names that were to be given new meaning. In addition we wanted programming multi-threaded operations to 'feel like' programming in Prolog. Finally, we sought a model that would naturally and efficiently integrate with the C# threading model. That is, we wanted clean integration on the C# side as well as the Prolog side.

We achieved this by adding to P# several new built-in predicates, without changing the Prolog syntax. These predicates approximately match the facilities for concurrency

found in C#: that is creating a thread, locking and more sophisticated functions of monitors such as waiting and pulsing. Pulsing is the C# term for notification.

We retained a Prolog feel to these features by using shared variables as message channels and unification as a means of sending messages, as with other concurrent forms of Prolog.

We also allow interaction between threads by providing a global database that all threads are able to read and modify, while still associating with each thread a local database allowing it to manipulate data imperatively without interference from other threads.

Having done this, it is possible for multiple P# threads and multiple C# threads to interact with each other. In particular a C# thread may interact with a P# thread while it is running, rather than having to wait for it to succeed or to fail.

In this thesis we will discuss three case studies which show the P# language in typical use. The first case study is an application which allows several users to modify a database. The users are able to disconnect from the database and to modify their own copies of the data before reconnecting. On reconnecting, conflicts must be resolved. The other two case studies are both P# implementations of software engineering tools. The first is a tool for querying the contents of an object-oriented library. The second is a tool for viewing an object-oriented class hierarchy. These tools provide a motivation for an extension to P# which allows larger databases to be stored and queried than those which can be handled by Prolog Café.

The C# produced by version 1.1.3 of P#, which has concurrency support but retains the compilation scheme of Prolog Café, is very unlike the code which a human programmer might produce. We will show how, aided by mode and type annotations, more idiomatic C# can be generated by translating tail-recursion into **`while`** loops and by applying liveness analysis to remove unnecessary variables. We apply these optimizations to semi-deterministic and deterministic predicates which do not have non-deterministic predicates beneath them in the call tree, since such predicates enjoy relatively simple control flow. We demonstrate the benefit of generating C# code closer

to that which a human programmer might produce and which can therefore be compiled efficiently and well by a C# compiler. This improvement in the high-level code generated by P# significantly speeds up the execution of a range of benchmarks that we have compiled. A secondary benefit of this approach is that the generated code is easier for a human programmer to read or to modify without inadvertently introducing flaws in the program logic which manifest themselves as subtle run-time errors.

Chapter 2

Existing Technology

2.1 Logic Languages and Functional Logic Languages

2.1.1 The Warren Abstract Machine

Many of the fastest Prolog interpreters are based on a sophisticated compilation technique known as the WAM [AK91, War83, War88] (Warren Abstract Machine), named after its inventor David H. D. Warren. Work has been done on formally verifying that this compilation technique is correct [Pus96].

Since the invention of the WAM, variations on it and optimizations of it have been suggested [KN90, Han92, Li96]. More major variants include the LLPAM [TK96], which compiles the linear logic extension of Prolog, LLP.

The book [Cam84] 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 [AK91] builds 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 machine at the next stage, M_2, is capable

of compiling Prolog without backtracking, that is the ability to express conjunction by having rules of the form

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

is introduced.

With the machine M_3, this is extended 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 the cut is added, various design optimizations employed, and support for constants, lists and anonymous variables is added.

Below the construction process is briefly summarized.

2.1.1.1 M_0: Unification

Terms, for example $a(X, b(Y))$, are represented on the heap using pointers to avoid duplication. Each term consists of a cell storing its predicate and arity, for example $a/2$ followed by, in this case, two cells each pointing to structures representing the terms X, $b(Y)$. A query term is translated into instructions that build a representation of the query term in 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 it can be assumed 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. The Prolog evaluator starts off in the READ mode with the program term being matched functor for functor against the query term. When an unbound variable is encountered in the query, the WRITE mode is entered 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.

2.1.1.2 M_1: Allowing Programs with more than One Fact

Next, several unification equations have to be solved simultaneously. This is done by storing the code for each fact in a CODE area, and introducing instructions to jump to the relevant piece of CODE. There can only be one fact per predicate name, so it is known immediately from the query which piece of code to execute.

2.1.1.3 M_2: Adding Conjunction

The 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. For each predicate name there is 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 involve the repeated application of leftmost resolution. That is the evaluator attempts to unify the leftmost goal (g_1) with the head of the clause in the program that has the same outermost predicate as g_1. If this fails, the entire query fails. If it succeeds g_1 is replaced in the query with the body, $a_1, \ ..., \ a_n$, on the right of the program clause that has been selected.

This continues until the query fails, or the empty query remains, 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
true.              expanding c(Y). Y=1

Y = 1.
```

When a query is executed, then, code is needed 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 the clause

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

is translated into the WAM instructions:

get arguments of p_0
put arguments of p_1
call p_1
$\vdots$
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 it is necessary to save permanent variables, being those which occur in more than one body goal, in an environment stored in a stack of environments. The instructions

`allocate N` and `deallocate` are added. These instructions 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.

2.1.1.4 M_3: Adding Disjunction

Now, backtracking is added. 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. Instead the evaluator should backtrack to the last *choice-point* and continue from there.

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

Initially one might think of allocating the choice points on a separate stack to the environments. There is 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 preceded them. Only when all courses of action from a given choice point have failed, is that choice point popped, and then the environment frames that 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.

Three instructions are added: `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]
⋮
L_{k-1}: retry-me-else L_k
[code for penultimate clause]
L_k: trust-me
[code for final clause]

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

2.1.1.5 Optimizations

Optimization is based on three WAM principles. Firstly, heap space should be used sparingly. Secondly, registers should be allocated to minimize 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.

2.1.1.6 The Cut

A backtrack cut register is added, 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,

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

There are specialized instructions for these two cases.

2.1.2 Translating Prolog to C: GNU Prolog

GNU Prolog (formerly known as wamcc) [CD95] translates Prolog to C via the WAM. The paper [CD95] 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 optimized, 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 [CD95], which we summarize below, details how four systems, namely Janus, KL1, Erlang and, GNU Prolog 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 branches, where the location to be jumped to is known when the program is compiled. In addition, indirect branching is needed, 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 `proceed` instruction, which ends the code written for a given predicate.

2.1.2.1 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 the current function call. Thus, Janus compiles a Prolog program into a single C function using a switch statement. The resultant large C function takes a long time to compile.

In KL1, each predicate is compiled into a separate function and branching is implemented as a function call. One might consider a scheme that resembles the use of continuations, in that the functions never return. Instead, each function calls another function before returning. However this can lead to stack overflow. The solution adopted is to use a supervisor function of the following 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 function 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, which makes it possible to store a label in a pointer, and then to jump to the location contained in that pointer. Each predicate is again compiled into a separate C function, and a global table of addresses to which we may wish to jump is maintained. This approach has a number of disadvantages. For instance, all variables used must be global, as functions are called by jumping into them, bypassing any setup for local variables.

2.1.2.2 GNU Prolog

GNU Prolog 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.
```

which is translated into:

```
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 the function first called when executing the clause `p :- q, r`. This tries the goal `q`, making a note to jump to the code which tries the goal `r` afterwards by putting the label `label_p1` into the continuation pointer.

2.1.2.3 Summary

Thus, much of the work aimed at translating into C has been directed towards the production of code that is as fast as possible by exploiting the ability in C to get 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, does not allow these 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, as GNU Prolog does in the way that it jumps into the middle of functions. There may be some scope for ingenuity in doing this.

2.1.3 Translating Prolog to Java: Prolog Café

Prolog Café is a program developed by Mutsunori Banbara and Naoyuki Tamura. P# is based on version 0.4.4, which was released in 1999. Since the work described in this thesis was carried out, version 0.6.1 has been released, which includes support for concurrency of a different nature to ours. Prolog Café translates LLP via a linear logic extension of the WAM, namely the LLPAM [TK96], to Java.

The linear logic features that P# has inherited from Prolog Café are detailed with examples in the paper [TK97].

Prolog Café is an extension of jProlog [jPr], which uses a continuation passing style of compilation referred to as binarization and described in [TB90].

Prolog Café consists of a run-time system written in Java, which simulates the WAM derivative, and several Prolog/LLP files that implement:

- translation to Java,
- a Prolog interpreter,
- input/output, and
- the ability to call certain Java methods from Prolog.

These Prolog files are translated by Prolog Café into Java. Thus the compiler is bootstrapped, in that it is able to compile that part of itself that is written in Prolog. Then, both these sets of Java files are added to a Jar file.

The user is able to run this program as a normal, but limited, Prolog interpreter. In addition, the program can be used to compile some other Prolog source files to Java. The resultant Java files can be compiled with a standard Java compiler to produce class files that can be coupled with the Prolog Café class files.

The Prolog source file will usually have an entry predicate, called `main/0`, say. Prolog Café can then be told to run this predicate when started. It should be noted that the Java class files of the run-time system of Prolog Café are essential. The class files generated from the Prolog source can do nothing on their own.

In Prolog Café, each term is represented as an object, which is an instance of one of the classes: `VariableTerm`, `IntegerTerm`, `DoubleTerm`, `SymbolTerm`, `ListTerm` and `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 take terms as arguments, without knowing what type of terms they are.

Each predicate is implemented as a subclass of the `Predicate` class. This has fields for the arguments passed to the predicate and for a `Predicate` representing the goal to try next (using a continuation style); and the code which 'executes' the predicate.

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

2.1.3.1 Dealing with Resources

As mentioned above, Prolog Café 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 nor-

mal), 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 of a consumption level and closure, amongst other things. Resources are added to the table by the use of the operators $\Rightarrow$ and $\multimap$, and removed on backtracking.

A register, L, is added, which stores the *current consumption level*. At some point in the proof tree, only resources with a consumption level equal to the current value of L may be consumed. When creating a new resource, if it is an exponential resource (one which may be consumed as many times as wished) 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 [HWTK98].

2.1.3.2 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 in Java the strategy used by GNU Prolog for branchings is, though, inconceivable.

2.1.4 Jinni

Jinni [Jin] is a Prolog implementation which compiles Prolog to Java byte-code or to the CIL. A Prolog file is compiled to a Jar file containing the compiled Prolog predicates and the Jinni run-time system, or to an .NET executable file. Jinni, which stands for, Java INference engine and Networked Interactor is designed for combining knowledge processing capabilities with Java objects in distributed applications.

Java can be called from Jinni through a reflection mechanism, which converts Prolog

data-types to their closest matches in Java. If there is no Java method which is an exact match for the types of the arguments in a call from Prolog, then Jinni searches through the available methods and attempts data conversions in an attempt to find the behaviour intended by the programmer. A table of persistent objects is maintained, to which Java object can be added and from which they can be removed. This table contains object handles which can be used from the Prolog side to invoke Java methods and access Java fields.

Jinni supports an object-oriented extension of Prolog, which integrates with Java's object-oriented features. It is possible to wrap a Java class up in a Jinni class, using the reflection mechanisms described above.

Jinni code can be called from Java code, by expressing a query as the same string that might be typed in to a Prolog interpreter, with a modified syntax for returning results to the Java side, for example

```
prologMachine.run( "X :- X is 1 + 1" )
```

evaluates to `2`.

Jinni supports *fluents*, which represent stateful resources. The state can be modified by putting and getting operations. When all of the resources of a fluent are used, it reaches the end of its life. Fluents can be used for interoperation with Java, for example reading from and writing to files, strings or sockets.

2.1.5 Mercury

A functional logic language, called Mercury [Bec, CSH, HCS$^+$, Mer], is available for use with the .NET platform. Mercury, despite being reminiscent of Prolog, is a fully declarative language. Thus, the developers of Mercury did not have to deal with some of the issues arising from the use of Prolog cuts. The basic syntax of Mercury is similar to that of Prolog, with added notation for mode declarations and function declarations. Because Mercury is declarative, I/O has to be programmed by passing a variable around which represents the current 'state'.

In many cases, it can be difficult or tedious to translate existing Prolog applications to Mercury. This is because Mercury does not support failure driven loops, user defined operators or difference lists; and the support for cuts and I/O is different from that of Prolog. These issues are dealt with in [CSH].

In Mercury, unlike in Prolog, it is possible to declare function facts and function rules. A function fact is of the form: `Head = Result` and a function rule is of the form

```
Head = Result :- Body.
```

The following is an example taken from [Bec].

```
:- func fibonacci(int) = int.
:- mode fibonacci(in) = out is det.

fibonacci( N ) = F :-
    ( if N =< 2 then
        F = 1
     else
        F = fibonacci( N - 1 ) + fibonacci( N - 2 )
    ).
```

The function `fibonacci` is defined as one from integers to integers taking the argument `N` and returning `F`. The mode declaration indicates that the `N` is an `in` parameter, that is, it is instantiated on the call to `fibonacci` and the `F` is an `out` parameter, that is, it is instantiated on exit from the function. The `det` mode declaration indicates that the function is deterministic which means that the function produces exactly one solution. Other possible modes are `semidet` which means that the function or predicate produces zero or one solution, `multi` which means that it produces at least one solution and `failure` which means that it produces no solutions.

The equivalent Prolog for this example would be:

```
fibonacci( N, F ) :-
    ( N =< 2  ->
        F = 1
    ; (
        fibonacci( N - 1, F1 ),
        fibonacci( N - 2, F2 ),
        F is F1 + F2
      )
    ).
```

This contains no hints to the Prolog compiler that `fibonacci/2` is going to be used in a functional manner, and without the compiler seeing the client code there is no way for it to infer this.

Failure driven loops cannot be coded in Mercury because of its declarative nature.

2.1.6 HAL

The HAL language [HAL, dlBDMS02] is, like Mercury, a functional logic language. It is strongly typed and is focused on supporting the construction and extension of constraint solvers. The design objectives of HAL also included efficient constraint solving, low-overhead interoperability with languages such as C and providing more compile-time checking than preceding Constraint Logic Programming (CLP) languages.

2.2 Functional Languages

2.2.1 Translating ML into C

The paper [TLA92] describes a translator that 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.

2.2.2 MLj

MLj [MLj, BK99, BKR99] translates SML to Java byte-code, and thus allows the integration of SML code with a Java program. SML 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 source code 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 low-level code, which is then translated to byte-code. The whole-program approach to compilation allows significant optimization to be performed.

2.2.3 SML.NET

The interoperation of SML [SML, BKR04] with other .NET languages is integral to the design of the SML.NET language. Where SML has features equivalent to .NET features the SML feature is mapped to the appropriate .NET feature. In other cases, the SML language itself is extended with a feature similar to the .NET feature. For example, the module system is mapped to namespaces and classes, and SML constructors are mapped to .NET class constructors of the same name. The primitive types of SML are a good match for the .NET types. Types of .NET objects map to a similar SML type wrapped in an `option` constructor so that **`null`** objects can be handled. Methods which are declared **`void`** have `unit` result type. Methods which take no arguments have `unit` argument type, and those which do take arguments have a tuple argument type.

The close analogies between SML and C# mirror those between SML and Java remarked on above and allow SML.NET to achieve close integration of SML with the .NET platform. Indeed, SML.NET is based on MLj.

2.2.4 A Haskell COM Server

The paper [FLMJ99] discusses a Haskell COM server. This allows Haskell programs to be expressed as COM objects using any Haskell implementation that has a foreign language interface that allows Haskell functions and pointers to be imported into Haskell and exported from Haskell. That paper details how developing such a tool required consideration of subtle issues. The implementation had to minimize what was required from the foreign language interface of the Haskell implementation in order that the tools which generate the COM code do as much as possible of the work, and the range of supported Haskell compilers is as broad as possible.

2.3 Other Implementations

2.3.1 Translating Java to C

Toba [PTB+97] 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.

2.3.2 Translating Java to C#

Translation from Java to C# can be easily automated, producing in most cases readable and idiomatic C# code. This is a result of the similarity of the language's syntax and libraries. Many Java keywords have direct equivalents in C# with identical or almost identical semantics.

Chapter 3

Translating Prolog to C#

3.1 Porting Prolog Café

3.1.1 Bootstrapping the Translator

In developing P# from Prolog Café, the most fundamental modification was to the translator, which is written in Prolog. Modifications needed to achieve the generation of naïve C# were straightforward, as the Java produced is simple and does not rely on libraries. The only modifications needed were changes of syntax, for example, 'extends' becomes a colon; and changes due to the fact that, unlike in Java, in C# one needs to be explicit about method overriding.

The Prolog Café translator can, with minor modifications, be compiled by SICStus [SIC] Prolog version 3.7.1. However, some of the other parts of Prolog Café that are written in Prolog use linear logic resources in places, and so cannot be compiled by SICStus Prolog. Resources are used in the interpreter and the input/output code. We were able to obtain a bootstrapped compiler to C# written in C#, from Prolog Café, by running only Java programs. How this was done will be explained in prose and with the aid of a diagram.

We will denote a program that compiles LLP to language *D*, which is itself written in

language E, by P_E^D. Thus we start with P_{Java}^{Java}. We denote the LLP to Java translation engine by T, and the modification that produces C# by T'. Essentially $T = P_{LLP}^{Java}$ and $T' = P_{LLP}^{C\#}$.

By running P_{Java}^{Java}, on T' we obtain a program that compiles LLP to C#, but which is written in Java, that is $P_{Java}^{C\#}$. By then running $P_{Java}^{C\#}$ on T' we obtain a program that compiles LLP to C#, but which is written in C#, that is $P_{C\#}^{C\#}$.

Finally, we apply this program to T' to verify that it is correctly bootstrapped, that is, $P_{C\#}^{C\#}$ run on T' yields $P_{C\#}^{C\#}$.

By writing $A(B)$ to mean the source code output of the program with source code, A, run on the source code of program B we can summarize this entire process by the equation:

$$\left(P_{Java}^{Java}\left(P_{LLP}^{C\#}\right)\right)\left(P_{LLP}^{C\#}\right) = P_{C\#}^{C\#}$$

Figure 3.1 shows this process in the form of a T-diagram. Each T represents a program that occurs at some point in the bootstrapping. At the bottom of each T the language in which that program is written is printed. The top part of the T shows the language that the program translates from and the language that it translates to. On the top of the arrow the name of the program is written. The `plcs` program is the translator from LLP to C# that was obtained by modifying the Prolog file that translates from LLP to Java in Prolog Café.

3.1.2 The Runtime-system

The above process produces the C# files corresponding to the translator. It does not, however, produce a run-time system. This part had to be hand-translated from Java to C#. On the whole this was a straightforward process. The libraries of the two languages are similar, as are the semantics. Some of C#'s keywords are semantically practically equivalent to those of Java, and these could be changed by a search and replace procedure. For example: **`public final class`** `A` **`extends`** `B` becomes **`public sealed class`** `A : B`.

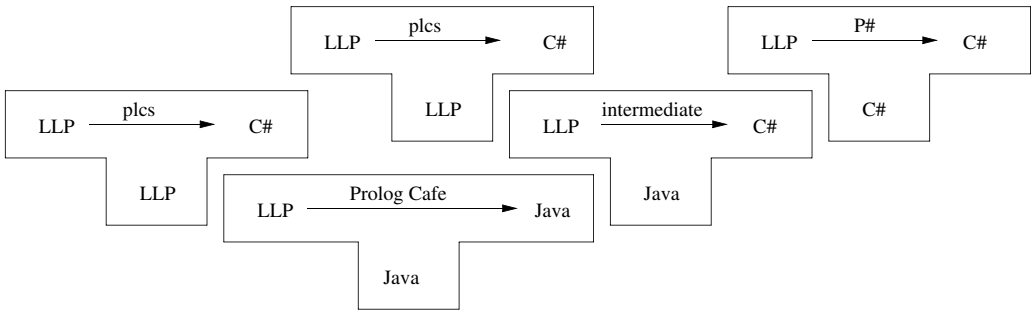

Figure 3.1: Obtaining a bootstrapped translator to C#

3.1.3 Architecture

The basic unit of deployment for the .NET Framework is an assembly. An assembly consists of a number of modules, metadata and possibly resources. An assembly may be a DLL (dynamic link library) or an EXE (executable) file.

It was necessary to decide whether the core of P# should be placed in a DLL and the user generated files in an executable file or *vice versa*. The two possibilities have different advantages, and both seem to be sensible. It is easier for a user to generate an EXE file, which can have a standard class to call the DLL incorporated into it. On the other hand, it is the P# code that is called first, is in control, and calls the user's code. Furthermore, the user may wish to split their code across several .NET assemblies.

The P# interpreter uses reflection to locate the C# translations of Prolog predicates. Reflection is also used to locate the main predicate when running a translated Prolog program. Whether P#'s main code resides in a DLL or not, we need to locate classes in assemblies other than the assembly containing the main P# code. This is because the main P# code needs to be deployed as a unit to the user, who will then generate their own code in separate assemblies. Thus, we added to P# a class storing a list of assemblies, and a predicate that loads a given assembly. This predicate can be called both during an interpreter session and from Prolog that has been compiled to C#. To dynamically find a class P# first looks in its own assembly and then tries each of the

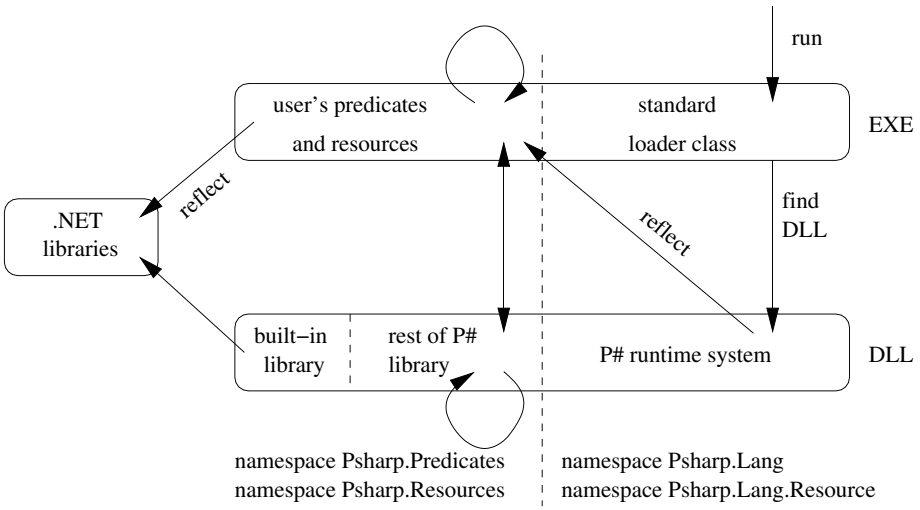

Figure 3.2: Separation into a DLL and an EXE

assemblies in the assembly list.

We decided that it was important to protect the user from the issues involved in generating a DLL and then having to make it visible to the executable program that uses it. Thus, the P# runtime-system and libraries were placed together in a DLL. The user, having used P# to generate C# files for their predicates, then compiles their C# files together with a special `Loader` class.

The `Loader` class simply calls the main method of P# in the DLL. This method discovers which assembly called it, that is, the user's assembly, and then adds that assembly to the list of assemblies mentioned above. P# can now find the user's main predicate by reflection and call it. This process is summarized in Figure 3.2. Usually after the first two reflections have occurred the predicates that are to be called can be determined statically, thus there is little overhead associated with the use of reflection. When a C# field is read or altered or a C# method is invoked, however, we need to use reflection again. As with Prolog Café, some of the more frequently required calls into the libraries are hard coded into built-in predicates. This is the section of the DLL labelled 'built-in library' in the figure.

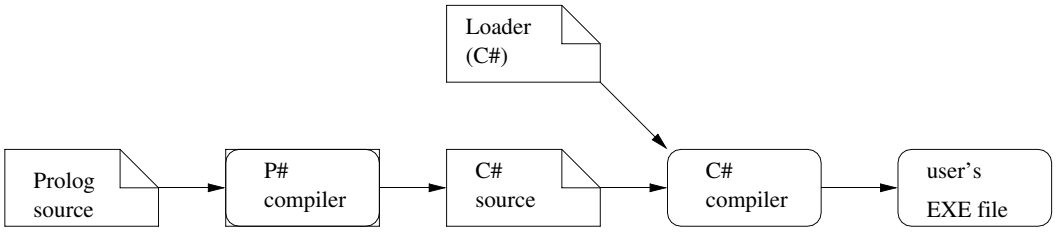

Figure 3.3: How the user generates their EXE file

It is necessary also to give the C# compiler a path to some copy of the DLL since the user's C# files will contain references to classes in the DLL.

In some cases the user may wish to create two or more assemblies of their own to exploit P#. In this case one of the assemblies can call the assembly load predicate to load the other one. In addition we provide a trivial executable file that contains only a class similar to the `Loader` class mentioned above. This loader runs the DLL directly in the interpreter mode. This program therefore allows the interpreter to be run as a command-line application. Thus, we have essentially allowed P# to be used as either a DLL or as an executable application.

Figure 3.3 shows the process by which a user is able to generate a stand-alone C# application from a Prolog/LLP source file.

3.2 Use of C# Features

We investigated whether value classes or delegates could be used to improve the efficiency of the translated code.

P# inherits from Prolog Café a supervisor function scheme for continuation style code. The supervisor function is of the following form:

```
Predicate code = <initial code>;
while( code != null )
    code = code.exec( engine );
```

Thus, each predicate call returns the `Predicate` object which is to be executed next, that is the continuation. The `Predicate` object is being used as a function pointer, so the same effect can be achieved using delegates. Each `Predicate` class can be given a static field that stores a pointer to a static `exec()` method, and these can be passed around instead of objects.

The delegate is defined as follows:

```
public delegate void PredicateCall( CallArray a );
```

where `CallArray` is a class containing a stack of `Call` objects, and a `Call` object is a class or `struct` containing an instance of the `PredicateCall` delegate and some arguments to be passed to the predicate (an array of `Term`s). Let the delegate field be named `d`.

The supervisor function now becomes:

```
CallArray ca = <initial stack of calls>;
while( true ) {
    next = ca.Pop( );
    if( next.d == null )
        break;
    next.d( ca );
}
```

Each predicate is compiled into code that pushes new predicate calls onto the stack `ca` that it is passed.

A test program was written to make many simple Prolog calls using the original scheme and using the delegate scheme described above. In each case, the Prolog query `a.` was run with the following program a large number of times.

```
a :- b, c, b, c.
b :- c.
c.
```

With the delegate scheme the `CallArray` variable was initialized to contain just a call to `a`. The timings were taken by using the C# `System.DateTime.Now.Ticks` property.

Table 3.1: Experiments with structs and delegates

Constructs used	time in seconds with Debug build	time in seconds with Release build
Objects only (original)	13.2	5.2
Delegates	7.9	5.9
Delegates and structs	11.1	8.8
Interfaces	10.4	5.5

It was found that the delegate scheme was marginally slower in the Release build and somewhat faster in the Debug build. This difference is probably due to the fact that the C# compiler is good at optimizing code that makes heavy use of objects, but is unable to optimize so effectively the less natural use of delegates. Although the methods pointed to by the delegates are static, the call to the delegate is translated into a virtual call in the CIL.

We also measured the effect of using a **`struct`** to store the data for a predicate call. We found that the use of **`struct`**s was far less efficient than the use of objects.

We also investigated the efficiency of a scheme where a `Predicate` interface was used instead of predicates being subclasses of a `Predicate` superclass.

Table 3.1 shows the results of the experiment.

We concluded that delegates, structs and interfaces should not be used in this way in P#.

We decided that it was desirable for a P# project to be free of unmanaged code. Thus we took a design decision not to make any use of unsafe code blocks. In so doing we had to sacrifice a potential improvement in runtime performance as for example the use of pointers may have improved the efficiency of the Prolog stacks of P#.

C# has adopted the Visual Basic ability to declare property setters and getters. This practice allows fields to be accessed and assigned to as though they were variables, when in fact they are properly encapsulated in their own class and accesses go through methods that the programmer specifies using a special syntax. This is a pattern that

occurs frequently in Prolog Café, and indeed most object-oriented code, so where possible setters and getters were used in P#.

It is possible to call other .NET languages from C#, and for other languages to call C#. Thus, by going via C#, it is possible for P# Prolog code to interoperate with other .NET languages.

C# is a typed language and Prolog is an untyped language. However, in Prolog, terms may be one of integers, floats, atoms, structures, lists, variables and in P#, also C# objects. As in Prolog Café, Prolog integers map to the **`int`** type, Prolog symbols map to strings and Prolog lists map to arrays of objects. In the implementation of P# integers are represented by objects of the class `IntegerTerm`, which encapsulates a C# integer. Implementing types in this way, with each P# Prolog type implemented as a C# type, promotes interoperation between Prolog and C#.

We define a new predicate `:/2`, which is used as an infix operator for calling C# methods. For example the `Max` method in `System.Math`, which takes two integer arguments and returns their maximum, can be called in the following way:

```
'System.Math':'Max'( 3, 4, M ).
```

This call will instantiate the variable `M` to 4. A method can also be called on an object, with for example, the following code:

```
Object:'Method'( InArg1, InArg2, InArg3, Out ).
```

The colon is usually used in Prolog to separate a module name from a predicate in the module. This usage is analogous to our use of the colon, which separates a C# class or object from a method defined therein.

The C# libraries contain methods that compile C# source code to either an executable file or to an assembly that resides in memory. This enabled us to define a predicate called `plcomp/1` that compiles a given Prolog file into memory.

```
a( X ) :- b( X ), c( 2 ).
a( X ) :- d( X ), e( X, Y ).
```

Figure 3.4: A simple Prolog predicate

3.3 Example Code Generated by P#

Figure 3.4 shows a simple Prolog predicate, chosen to illustrate the way in which both conjunction and disjunction are compiled, and Figure 3.5 shows part of the C# class into which it is compiled. Figure 3.6 shows the WAM code produced by GNU Prolog for this Prolog predicate. The `try` and `trust` WAM instructions can be seen in both the C# code and the WAM code. The `execute` WAM instruction is implemented in C# by adding a predicate call object to the continuation. The `allocate` and `deallocate` WAM instructions are implemented by the P# runtime-system.

The C# code is directly comparable to the Java code produced by Prolog Café, with the main difference being the use of namespaces. The predicates are run by a supervisor method. This method calls each predicate, which returns the predicate that is to be called next (the continuation), and then calls the next predicate. The `exec()` method of `A_1` generates a choice point frame by constructing a `Predicate` that models the operation of the `try` WAM instruction. This calls `A_1_1` first, which executes the first clause of the Prolog predicate `a/1`. If this fails or more solutions are requested `A_1_sub_1` is then called, which in turn calls `A_1_2`. This executes the second clause of the predicate. The `exec()` methods of `A_1_1` and `A_1_2` both construct a chain of `Predicate` objects, whose `exec()` methods are executed in sequence. Thus after `A_1_1.exec()` has been called, the `Predicate` object created by `new Predicates.B_1(a1, p1)` will be called, and so on.

```
namespace JJC.Psharp.Predicates {

using JJC.Psharp.Lang;
using JJC.Psharp.Lang.Resource;
using Predicates = JJC.Psharp.Predicates;
using Resources = JJC.Psharp.Resources;

public class A_1 : Predicate {
    static internal readonly Predicate A_1_1
        = new Predicates.A_1_1();
    static internal readonly Predicate A_1_2
        = new Predicates.A_1_2();
    static internal readonly Predicate A_1_sub_1
        = new Predicates.A_1_sub_1();

    public Term arg1;

    public A_1(Term a1, Predicate cont) {
        arg1 = a1;
        this.cont = cont;
    }

    public A_1(){}

    [... code to set the arguments ...]

    public override Predicate exec( Prolog engine ) {
        engine.aregs[1] = arg1;
        engine.cont = cont;
        return call( engine );
    }

    public virtual Predicate call( Prolog engine ) {
        engine.setB0();
        return engine.jtry(A_1_1, A_1_sub_1);
    }

    [... code to return the arity and string representation ...]
}
```

Figure 3.5: C# code generated from the simple predicate

```
sealed class A_1_sub_1 : A_1 {

    public override Predicate exec( Prolog engine ) {
        return engine.trust(A_1_2);
    }
}

sealed class A_1_1 : A_1 {
    static internal readonly IntegerTerm s1 = new IntegerTerm(2);

    public override Predicate exec( Prolog engine ) {
        Term a1;
        Predicate p1;
        a1 = engine.aregs[1].Dereference();
        Predicate cont = engine.cont;

        p1 = new Predicates.C_1(s1, cont);
        return new Predicates.B_1(a1, p1);
    }
}

sealed class A_1_2 : A_1 {

    public override Predicate exec( Prolog engine ) {
        Term a1;
        Predicate p1;
        a1 = engine.aregs[1].Dereference();
        Predicate cont = engine.cont;

        p1 = new Predicates.E_2(a1, engine.makeVariable(), cont);
        return new Predicates.D_1(a1, p1);
    }
}
}
```

Figure 3.5 (continued): C# code

```
predicate(a/1,1,static,private,user,[
    try_me_else(1),
    allocate(0),
    call(b/1),
    put_integer(2,0),
    deallocate,
    execute(c/1),

label(1),
    trust_me_else_fail,
    allocate(1),
    get_variable(y(0),0),
    put_value(y(0),0),
    call(d/1),
    put_value(y(0),0),
    put_void(1),
    deallocate,
    execute(e/2)]).
```

Figure 3.6: WAM code produced by GNU Prolog

3.4 Example Web Application: Noughts and Crosses

One of the most appropriate uses of our tool is to combine a Prolog back-end with a Windows front-end, such as a Web Service. As an example we discuss how we implemented a Graphical User Interface (GUI) front-end to a Prolog program which allows the user to play a game of noughts and crosses (or Tic Tac Toe).

As a variation on the usual game we used a 4×4 grid, rather than a 3×3 grid. One player inserts O's in the grid and the other inserts X's. The two players insert their symbols alternately until one player wins by obtaining a line of four of their symbol. Often the game ends in a draw with all the squares filled.

We implemented the game as a Web Application, where at every point in the game the user is allowed either to take the move themselves or to ask the Prolog program to take the next move. The user can play the computer or another user, or the computer can play itself.

The current board is stored at the C# level, and passed as a parameter to a Prolog predicate on each move. The code which requests that the back-end perform a move itself is as follows:

```
private void ComputerMove( ) {
    PrologInterface sharp = new PrologInterface( );
    Term a2 = new VariableTerm( );
    Predicate compMove = new Rule_3( currentPlayer, currentBoard,
                                     a2, new ReturnCs( sharp ) );
    sharp.SetPredicate( compMove );
    sharp.Call( );

    Term result = a2.Dereference();
    this.currentBoard = a2.Dereference();
    DrawBoard( (ListTerm)currentBoard );
    SwapPlayer( );
    CheckForWinningLine( );
}
```

The predicate returns the new board, which the C# code then draws. The predicate `rule/3` takes as arguments the current player and the current board and returns the modified board after the computer's move.

The `rule/3` predicate encodes three rules. The first rule plays the winning move if the game can be won by the computer in the next move. The second rule blocks a potential win by the other side, and the third rule calculates a score for each potential move and chooses the first maximum scoring move.

The following code draws the board. It does this by unpacking the Prolog list data structure:

```
private void DrawBoard( ListTerm newBoard ) {
    Term result = newBoard;
    Term elm;
    for (int i = 0; i < 16; i++) {
        elm = ((ListTerm)result).car.Dereference();
        Cells[ i ].Text = Convert( elm.ToString() );
        result = ((ListTerm)result).cdr.Dereference();
```

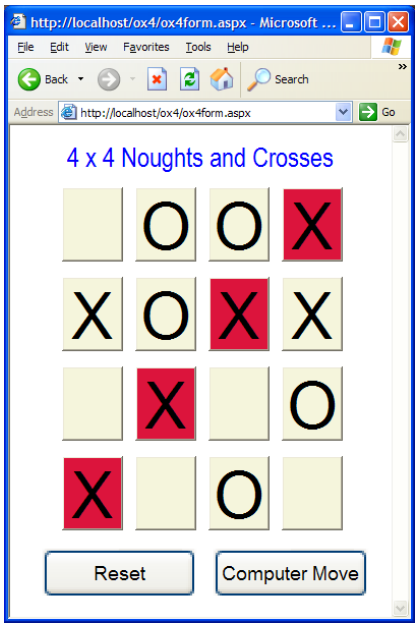

Figure 3.7: A Web Application

```
            if( !( result is ListTerm ) )
                break;
        }
    }
```

The calling mechanism used is that inherited from Prolog Café: the arguments are constructed as Prolog terms, and these are used to construct a `Predicate` object which can then be called, with the call blocking until the predicate succeeds or fails. A screen-shot of a game that the X player has won is included as Figure 3.7.

The squares of the board are implemented as C# Buttons, with a large font size chosen for the labels.

Chapter 4

Concurrency

4.1 Features of Concurrent P#

In order to be able to create new threads, we add a primitive predicate called **`fork`**`/1`. The **`fork`**`/1` predicate takes a structure as an argument and then forks a new thread, which evaluates the structure.

A **`fork`**`/2` primitive is also available. This predicate forks the first argument and returns a Prolog representation of a C# object representing the new thread. This object can then be returned to the C# part of the program where it can be used to stop that thread. A predicate, named **`stop`**`/1` is provided, which can be used to stop a thread from the Prolog side.

Having called **`fork`**`/1` with a structure containing an uninstantiated variable, anywhere in the syntax tree of the structure, a thread can use that variable to interact with the newly forked thread.

4.1.1 Communication between Threads

The **`wait_for`**`/1` predicate takes as an argument a variable that is shared with an already forked thread. It then waits until one of the threads instantiates that variable

and succeeds with the given instantiation. Except for this the instances of variables on different threads do not interact.

Consider the following program:

```
a( 2, 7 ).

and( Y ) :-
    fork( a( 1, Y ) ), fork( a( 2, Y ) ), fork( a( 3, Y ) ),
    wait_for( Y ).
```

Three threads are forked, each calling the predicate `a/2` with different values of the first argument. Only the second will instantiate `Y`, the second argument to 7. The call to **`wait_for`**`(Y)` will wait until this happens and then `and/1` will succeed with `Y = 7`. It is also acceptable for a forked thread to wait for the thread which forked it or for forked threads to fork more threads.

The following example shows that forking integrates with backtracking.

```
alpha( 'a' ).
alpha( 'b' ).
alpha( 'c' ).
alpha( 'd' ).

correct( X, Y ) :-
    \+ var( X ), % prevent cheating
    X = 'c',
    Y = X.

guess( Z ) :-
    alpha( X ),
    fork( correct( X, Z ) ),
    fail.
guess( Z ) :-
    wait_for( Z ).
```

The `guess/1` predicate has the task of discovering which of `'a'`, `'b'`, `'c'` or `'d'` is the correct answer. It can only find out this by calling `correct(X, Y)` with the correct letter as `X`, in which case `Y` is instantiated to that letter. The `guess/1` predicate forks a thread for each letter and waits for one of them to succeed. The variable `Z` correctly

retains its concurrent information on backtracking, as it comes into existence as soon as `guess(Z)` is called.

The following example is similar to the last, except that tail-recursion is used instead of backtracking. The user enters a square integer between 0 and 20^2. The program forks 21 threads to try each of the possible square roots, and then waits for one of them to signal that the answer has been found.

```
sqroot( S, R ) :-
    sqroot_threads( S, R, 0 ),
    wait_for( R ).

sqroot_threads( S, R, 21 ) :-
    !.
sqroot_threads( S, R, N ) :-
    fork( ( S =:= N * N, R = N ) ),
    N1 is N + 1,
    sqroot_threads( S, R, N1 ).
```

Note that the double brackets in the **`fork`**`/1` are necessary as the fork takes only one argument, which in this case is a structure with functor `,/2`. This provides a way of writing the code to be forked ‘in-line’.

4.1.2 Queuing of Multiple Solutions

It may be that the programmer wishes to use a concurrent variable more than once, indeed if he or she cannot, then some algorithms will require unnatural implementations.

If a bound concurrent variable is later unbound by backtracking, and then bound again to the same or a different value, then that new binding is enqueued on the queue of messages to be consumed.

Thus, a producer can give multiple bindings to a concurrent variable, possibly composing a set of solutions; and a consumer can repeatedly call **`wait_for`**`/1` to take each binding.

The **wait_for**/1 predicate can be called repeatedly by using the **repeat**/0 predicate, however **wait_for**/1 also creates a choice-point and will yield the next solution on backtracking.

The following code creates two threads, a producer and a consumer. The producer generates integer values from 0 up to 10 and the consumer consumes each produced value, doubles it and outputs the corresponding result. The producer uses a predicate called pulse/2, which makes a binding and then undoes it straight away. The first clause of pulse makes the binding, and then fails. This failure causes backtracking to the last choice-point, which undoes the binding we made in the first clause and then executes the second clause which succeeds. Thus, the predicate call succeeds having made no lasting binding. This allows us to imperatively give successive bindings to the same variable.

```
main :-
    fork( prod( X ) ),
    cons( X ).

cons( X ) :-
    wait_for( X ),
    X2 is X * 2,
    write( X2 ),
    nl,
    fail.
cons( X ).
```

```
prod( X ) :-
    enum( X, 0 ).

enum( _, 11 ) :-
    !.
enum( X, N ) :-
    pulse( X, N ),
    N1 is N + 1,
    enum( X, N1 ).

pulse( X, N ) :-
    X = N,
    fail.
pulse( _, _ ).
```

When it is detected that all of those threads that have a copy of a concurrent variable are waiting for that variable, then all of those calls to **wait_for**/1 fail. Thus, in the example above both of the threads eventually terminate, and in the square root example above, if the user asks for the root of a non-square integer the query will fail. If all of the forked threads with a variable succeed or fail having sent no message then a call to **wait_for**/1 on the remaining thread will fail. However, care must be exercised. If we had defined main/0 to fork both the producer and the consumer, then the main

thread running under the interpreter would still have a copy of the variable `X` although it would never use it. This would stop the consumer thread from terminating. It is still possible to fork both threads by forking a thread which itself first forks the producer thread and then runs the consumer code. In this case the variable `X` is introduced on the consumer thread, not on the interpreter thread.

4.1.3 The Global Table

Each forked thread is equipped with a private database that it can use in the normal way. In addition we provide a global database, which is shared between all the threads. Accesses are automatically protected by a mutex. The database can be modified by primitives **`global_assert`**`/1`, **`global_retract`**`/1` and so on. To query the database a **`global_call`**`/1` predicate is provided.

For example the following code will instantiate `Y` to `[1,2,3]`. In this case we could have just passed back the information in the concurrent variable, `X`.

```
global_example( Y ) :-
    fork( global_example_thread( X ) ),
    wait_for( X ),
    global_call( info( Y ) ).

global_example_thread( X ) :-
    global_assert( info( [1,2,3] ) ),
    X = []. % arbitrary instantiation
```

4.1.4 Comparison with Existing Concurrent Prologs

Recall our earlier discussion of concurrent logic programming languages.

Our **`wait_for`**`/1` predicate is reminiscent of the and-parallelism used in FCP and of the forking and event sending used in DeltaProlog. As with FCP, the messages passed in P# can be any Prolog term. Thus, it would be possible, for example, to pass a predicate call to be evaluated. This style of message passing is very different from that

used in the new version of Prolog Café, whose runtime-system contains a class that can be instantiated to act as a message channel.

The paper [HGC95] proposes a `wait` predicate that waits until a variable is bound, together with a range of forking primitives.

Our global table is similar to the Linda blackboards of Jinni, however blackboards also provide a synchronization facility as attempting to retrieve a term can block if there is no term to retrieve.

While we use variables to communicate we do not make use of guards in P#, as languages which use guards, such as FCP, tend to be very unlike standard Prolog.

4.2 Implementation

4.2.1 Making P# Thread Safe

The version of Prolog Café that we modified has no direct support for concurrency. The necessary stacks for the LLP/Prolog engine are encapsulated in an engine object. Unfortunately, creating two or more such objects, and running the resultant engines simultaneously, resulted in chaos because various static fields were shared between the engines.

The first task in the pursuit of adding concurrency to P# was to find these problematic fields, and to alter P# so that different threads no longer interfered with each other. Changes of this nature tend to degrade performance, because it is quicker to find static fields than it is to find instance fields. For this reason, we had to plan our changes with efficiency in mind.

When we searched for problematic static data in the runtime-system, we found three problems:

Firstly, there was a problem with the choice point frame stack which is a stack of entries representing choice points. This stack is stored as an array in Prolog Café. The

object representing an entry contained a static field, which was used to optimize the operation of this stack. This problem was solved by moving the static field into the stack object, where it became an instance field. This field is passed as an argument to the entry objects when necessary.

Secondly, an integer timestamp is used to identify a `VariableTerm`. A `VariableTerm` object represents a Prolog variable. The next value to be allocated is stored in a static field. This field needed only to be protected by a mutex.

Finally, we synchronized access to the hash-table storing Prolog symbols.

Having made these changes, multiple engines were still unable to execute concurrently. There remained a problem with the `Predicate` objects. Each Prolog predicate is translated into a C# class that contains methods for setting the arguments and for actually calling the predicate. In the case of a predicate with more than one clause, each clause after the first is compiled into a subclass of the first.

Every `Predicate` object has an instance field for the engine currently running the predicates, and another instance field for the continuation, that is the `Predicate` to run next. Predicates that have more than one clause have static fields that are statically, and finally, initialized to object instances of subclasses of the `Predicate` in question. Two threads running one of these subclass `Predicate`s will be working with the same object that was statically created. Thus, when two or more threads tried to manipulate the engine and continuation fields of such an object, they interfered with one another.

There are two possible solutions to this problem. The first is instead of relying on these static fields, to create a new object every time one is required. This solution would have been inefficient. A better solution is to redesign the storage of the engine and continuation data.

The engine field is only used in the `exec()` method of the `Predicate`. This method is always called by the engine. In the original code a method that initializes the engine value was called just before a call to the `exec()` method. Instead we added an argument to the `exec()` method, whereby the engine could pass a reference to itself to each call to `exec()`.

When the continuation field is used in a the principal `Predicate` class of the translation of a predicate, there is no problem, as each thread will create a separate object. The continuation field in a subclass of this is initialized at the beginning of the `exec()` method by retrieving a value recorded in the engine. Before, this overwrote the superclass continuation field. However, there is no need for this. Instead of using the object field we create a new local variable and initialize the new variable from the engine.

There are also fields that store the arguments to the predicate, but these are already dealt with in a manner similar to our modified treatment of the continuation field. Thus, we did not have to worry about the arguments to a predicate interfering with those of the same predicate running on a different thread.

Finally a problem arose with the implementation of first call optimization (FCO).

The FCO is used when a predicate has clauses that have the property that their first body goal has the same functor as their head goal. For example, consider the second clause of the following predicate.

```
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
```

When the second clause is executed, a new choice point need not be created, instead we make a note of the last choice point when the predicate `member/2` is called and re-use this choice point when the second clause calls `member/2`. Thus, the FCO allows the first body goal to call the predicate without consuming an unnecessary choice point. The FCO is discussed in [FSW98]. When the `Predicate` class representing the first clause was entered, a reference to this `Predicate` object was stored in the *static* field of the `Predicate` class, called `entry_code`. A subclass representing a later clause could then use this value to call the first clause again. It seemed that we needed to move this data into the engine. We could not, however, place it in a simple variable in the engine to be retrieved when needed. This was because between the storing of the value and its recall, another predicate that also uses the FCO could be called and wish to store its `entry_code`. Given that usually only one such code need be stored, and never more than one per predicate in the source Prolog program, our solution was to use a hash-table in the engine that maps `Predicate` classes, that is, the type of

the `Predicate` object, to `Predicate` objects, that is, the current `entry_code` for that predicate.

Having made all of these changes, multiple engines could run different, or the same, Prolog programs concurrently, without interference.

4.2.2 Forking Threads and the Global Table

When a fork is called on a structure, a deep clone of that structure is created. For the most part this is just a copy, however, in the case of `VariableTerm`s we wish to maintain links between copies of the same variable on different threads, so that they can act as message channels.

For each variable in a forked structure, we create a `ConcurrentVariable` object, which records which of those threads with a copy of that variable are still running. Each `VariableTerm` object of a concurrent variable keeps a reference to its `ConcurrentVariable` object. If two `VariableTerm`s are unified, and one is concurrent, then the other is made concurrent, by copying this reference across.

The cloning process is recursive so that any `VariableTerm` that can be reached from the predicate call's arguments is correctly cloned in this way. The entire cloned structure is then built in a new Prolog engine, and executed.

Messages are stored on the global table. If one thread places a term in the global table, and another retrieves it, then that term passes from one engine to another. Whenever a term does this it must be rebuilt in the new engine, since `VariableTerm`s make use of the engine's trail. As a general solution, we provide a special engine for the global table. When a term is globally asserted it is first built on the global table engine, then the Prolog assertion code is run on the global table engine. We do not wish to create a new thread every time this occurs, so the assertion code is run on the global table *engine*, but on the thread that called **`global_assert`**`/1`. Thus, a call to **`global_assert`**`/1` does not exit until this has happened.

At the lowest level, all of the above is implemented by using a new internal predicate

called `$run_on_global_engine/1`, which given a structure, executes it on the current thread, but on the global engine. The arguments are cloned onto the global engine before the call. All of the solutions to the query are cloned back and asserted on the calling engine. The **`global_call`**`/1` predicate is implemented so that it then executes these solutions on the calling engine by calling this internal predicate. This is implemented in such a way that **`global_call`**`/1` creates a choice point and each solution can be retrieved in the usual way.

The definitions of the global predicates all call the normal assertion predicates wrapped in a call to `$run_on_global_engine/1`. The `$run_on_global_engine/1` predicate can only be executed by one thread at a time. The dollar sign is used to distinguish predicates internal to P# from those which are visible to the programmer using P#.

This following code shows the Prolog side of the implementation of some of the global table operations.

```
:- dynamic( '$soln'/1 ).

global_assert( ClauseToAssert ) :-
     global_table( Table ),
     '$run_on_global_engine'( assert( Table, ClauseToAssert ) ).

global_call( ClauseToCall ) :-
    global_table( Table ),
    '$run_on_global_engine'( clause( Table, ClauseToCall, ToCall ) ),
    call( '$soln'( clause( _, ClauseToCall, ToCall ) ) ),
    call( ToCall ).
```

The call **`global_table`**`( Table )` instantiates `Table` to the object encapsulating the global table. The **`global_call`**`/1` predicate first retrieves this table, and then runs on the global engine a call which converts the clause into code which can be executed (the variable `ToCall`). The code of `$run_on_global_engine/1` asserts the structure

```
'$soln'( clause( Table, SolnClause, ToCall ) )
```

which can then be called to obtain the structure `ToCall` which can then be executed

with another `call`.

The addition of the global table and its associated predicates to P# enables us to rewrite the noughts and crosses example above, so that the state of the board can be stored on the Prolog side and does not have to be passed back and forth every time a move occurs.

4.2.3 The `wait_for`/1 Predicate

When a concurrent variable is bound to a term, this binding is detected and a call to **`global_assertz`**`/1` is made, asserting the fact that the relevant variable has been bound to the relevant term. We use **`global_assertz`**`/1` as it asserts the fact at the end of the database so that the facts will form a queue and be retrieved in the order that they were asserted. In addition, the `ConcurrentVariable` object of that variable is pulsed to notify any waiting thread of the new instantiation.

The call to **`global_assertz`**`/1` is inserted into the stack of calls to be made at the first available opportunity after the binding has occurred. It is necessary to ensure that this call involves no cuts as this would destroy the logic of the code currently running at that time.

The **`wait_for`**`/1` predicate is built on top of **`global_call`**`/1`. It looks in the global table to see if an un-consumed instantiation has occurred, and if so it consumes it by calling **`global_retract`**`/1`, and succeeds having made the equivalent binding on its own thread. If not, it waits for a notification of an instantiation and when this occurs, it tries again.

Since the set of assertions in the global table for a variable are specific to that variable, the assertions for different variables do not interfere with one another.

The Prolog side of the implementation of **`wait_for`**`/1` is as follows:

```
'$on_unify'( CR, T ) :-
     lock( CR ),
     global_assertz( '$msg'( CR, T ) ),
     '$pulse_on'( CR ),
     unlock( CR ).
```

```
backtrackable_lock( L ) :-
     ( '$lock'( L, LOCKVAR )
     ;
       fail ).

wait_for( X ) :-
    '$get_concurrent_root'( X, R ),
    backtrackable_lock( R ),
    '$call_spin'( X, R ).

'$call_spin'( X, R ) :-
    global_call( '$msg'( R, B ) ),
    global_retract( '$msg'( R, B ) ),
    '$protected_unify'( X, B ).
'$call_spin'( X, R ) :-
    '$wait_on'( R ),
    '$call_spin'( X, R ).
```

The predicate `$on_unify/2` is called every time a concurrent variable is unified. It is passed the concurrent variable object which we refer to as the *concurrent root*, `CR` and the term to which that variable has been instantiated, `T`. While holding a lock on `CR` it asserts the instantiation on the global table using the internal predicate `$msg`, and then pulses the concurrent variable object to wake up any threads waiting for an instantiation to occur.

The **`wait_for`**`/1` predicate obtains the concurrent root for the concurrent variable, and then while holding a lock on the concurrent root, it calls `$call_spin/2` on the variable and its root. The `$protected_unify` predicate unifies its arguments without calling `$on_unify/2`. If there is a solution in the queue of solutions, it is retracted and then the concurrent variable is unified with the same term as that which it was unified with on the other thread. This unification is done without sending another message, hence 'protected unified'. Otherwise `$call_spin/2` waits by calling `$wait_on` for the concurrent variable object to be pulsed as a result of an instantiation, and then control returns to the first clause of `$call_spin`.

This implementation works correctly on backtracking and within failure driven loops.

4.2.4 Monitors

Our system includes two further primitives to ensure mutual exclusion among executing threads, namely, **`lock`**`/1` and **`unlock`**`/1`. Both of these take as an argument any Prolog term, and respectively acquire or release a monitor lock on the C# object representing that term. In the case of a concurrent variable, we lock on its `ConcurrentVariable` object.

On the implementation side, we are greatly helped by the fact that in C#, unlike in Java, a monitor can be entered or exited at any time by a library method call.

A **`backtrackable_lock`**`/1` predicate is also provided. This creates a variable and then binds it to an arbitrary term, setting a field in the corresponding `VariableTerm` object to indicate that this variable represents a lock. The monitor is then entered. The **`backtrackable_lock`**`/1` predicate creates a choice-point before making this binding, to ensure that on backtracking the variable will be unbound. When this un-binding is detected, the monitor is exited. The effect of calling **`backtrackable_lock`**`/1` is that everything deeper down the proof tree forms a critical region.

The P# runtime-system keeps track of each lock and unlock operation and maintains a variable, which stores the current depth of locking. When the P# Prolog thread terminates all of its locks are automatically released. This semantics mirrors that of the C# **`lock`**`/1` keyword, where a `finally` clause releases the monitor when the critical region is exited. Thus, if the thread is aborted, all of its locks are released.

4.2.5 Interoperation with C#

The locks dealt with by **`lock`**`/1` and **`unlock`**`/1` are C# locks on C# objects. Similarly **`fork`**`/1` initializes and starts a new C# thread, unification calls the `PulseAll()` method on an object and **`wait_for`**`/1` calls the `Wait()` method on an object, although they do far more besides this.

A C# program that calls a P# Prolog predicate may wrap such a call within a fork. Any variables passed to the predicate then become concurrent, allowing communication

between the C# code and the P# Prolog before the Prolog terminates.

As we saw in section 3.2, a P# Prolog predicate can call a C# method in the following way:

```
'System.Console':'WriteLine'( 'Hello World!', _ ).
```

The middle argument consists of the method name and any actual arguments. These C# arguments may include uninstantiated variables, in which case the C# will be passed a `VariableTerm` object. Thus, a concurrent variable can be passed from the P# Prolog side to the C# side. In this case the use of `:/2` should be wrapped in a fork, for example:

```
run_cs_method( In, Out, ObjectToCall ) :-
    fork( ObjectToCall:'CsThreadStart'( In, Out ) ).
```

This code would be matched on the C# side by code of the following form:

```
public object CsThreadStart( VariableTerm vt ) {
    ...

    // send message
    vt.Send( new IntegerTerm( 7 ) );

    // or await a message
    int msg = (int)( vt.Receive( ).toCsObject( ) );

    ...
    return ...
}
```

The `Send()` and `Receive()` C# methods use a temporary P# engine to respectively perform a unification and execute the **`wait_for`**`/1` predicate. Each undoes any existing binding of the concurrent variable that it is given first, and thus may be called repeatedly from the C# code. Such repetition must, however, be matched by backtracking on the P# side.

4.3 Semantics

Figure 4.1 shows the control flow logic of the unification and **wait_for**/1 operations. The symbol =# indicates the unification symbol = as used in P#, and the symbol = in the diagram represents unification as in standard Prolog. The figure, therefore, indicates that in P# unification first performs a standard Prolog unification and then in the case of unification with a concurrent variable, sends a message to a waiting thread.

The **wait_for**/1 predicate invokes the **global_call**/1 predicate to search for messages in the global table.

If it is successful, the message is consumed and then code deeper in the Prolog proof tree is executed without the lock being released. This deeper code consists of the code executed after the **wait_for**/1 predicate succeeds and up until there is backtracking back through the call to **wait_for**/1. When **wait_for**/1 is re-executed on backtracking the lock on the ConcurrentVariable object is still held, and is held until **wait_for**/1 can no longer be re-executed.

In general the way in which backtracking in predicates with side effects is handled is that there is no un-doing, see [EC98]. Thus on backtracking the side effect is performed again. The only exception to this is the **backtrackable_lock**/1 predicate, which releases the lock on backtracking. It is the **backtrackable_lock**/1 predicate that allows the lock to be held throughout the execution of deeper code.

If the call to **global_call**/1 fails, then we wait until a message appears, or in the case that all of the threads are waiting, the call to **wait_for**/1 will fail. For each concurrent variable associated with each Prolog thread, there are two Boolean flags, namely the waiting flag and the awaiting release flag. The waiting flag indicates whether the thread is currently waiting on the appropriate concurrent variable. The awaiting release flag is set when all of the threads are waiting and indicates that all the threads should be released and all of the calls to **wait_for**/1 for that variable should fail.

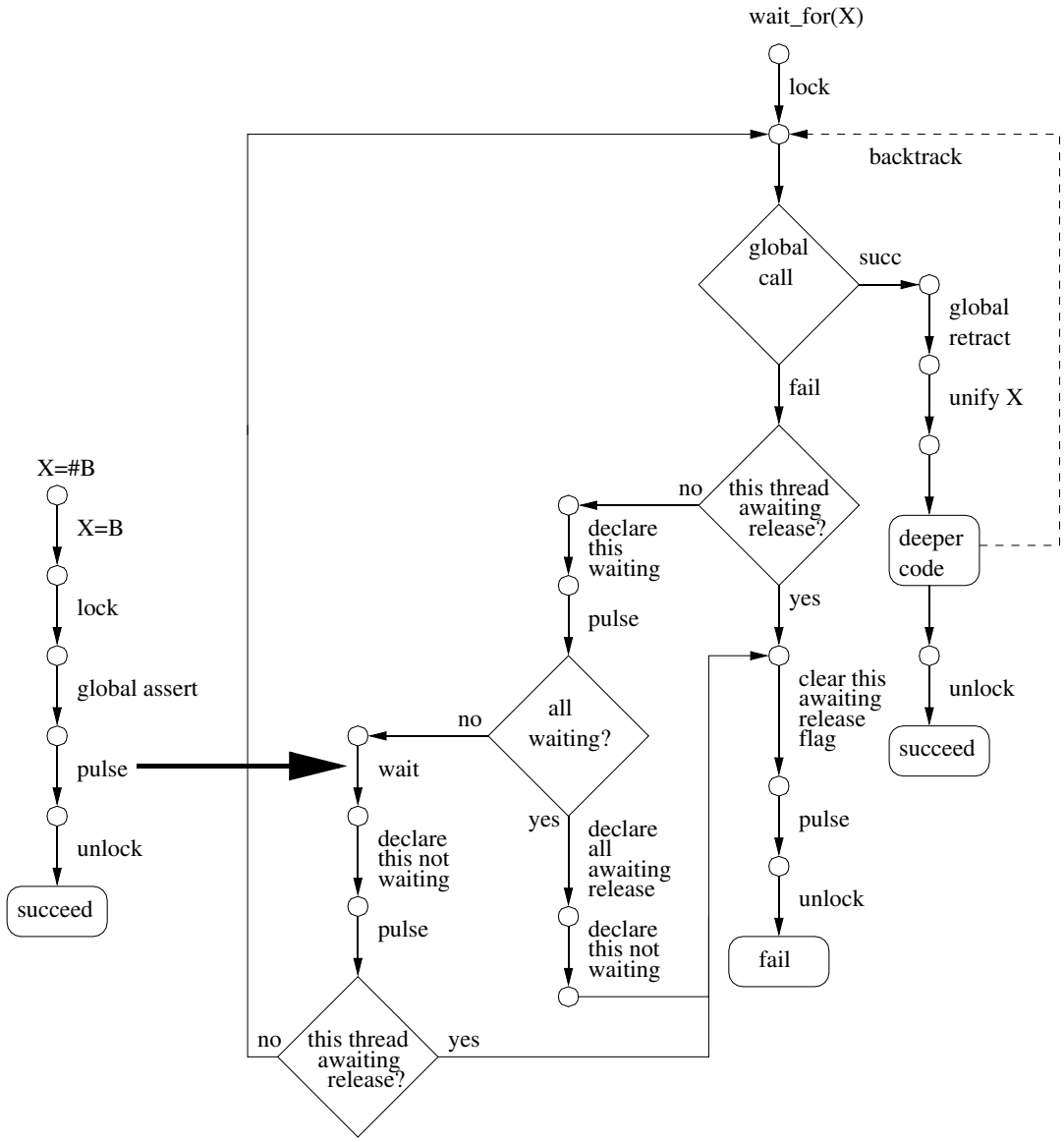

Figure 4.1: Control flow logic of unification and wait_for

Chapter 5

Case Studies and Performance Measurement

In this chapter, we consider three case studies which illustrate the use of P#: a disconnected shared database, an object-oriented assistant and a class hierarchy viewer. We also give a performance measurement taken before the developments reported in the next chapter.

5.1 A Disconnected Shared Database

We now give an example that illustrates the usefulness of our concurrency support in Prolog. Suppose that we have a central database that stores a generic set of facts. Several users are able to connect to this database, and to alter its facts. Each user can, at any time, disconnect from the database and manipulate their private copy. When they reconnect, their private copy must be synchronized with the shared database and with those of all the other connected parties.

Synchronization of two databases consists of determining which, if any, facts conflict; and in this case asking the user which fact to use. Much of the work in solving this problem occurs at the Prolog level, and concurrency is useful here. While a user is

trying to decide which fact is correct, we want to be checking other facts for consistency. Interoperation with C# is useful as we require network communication between the agents, and we would like a, possibly web based, graphical user interface.

Each predicate can be considered separately. The user can mark some predicates as being only allowed to have one fact: detecting inconsistencies here is trivial. Also the user can mark one or more of the arguments of a predicate as being key fields. In this case there cannot be two facts for that predicate that share the same key. The front-end can take care of what the facts are supposed to mean.

The central database runs as a server on a Linux machine. This is possible as P# runs on Mono version 0.26 or higher. Mono is an 'open source implementation of the .NET Development Framework', which runs on Windows and several distributions of Linux.

On starting, the server is passed an XML file, which details the list of predicates allowed, and for each predicate which fields are key fields. Having stored this data in the Prolog database, it waits for connections from clients.

Each agent runs as a client on a separate Windows machine. The client is able to send messages to and receive messages from the server by using SSH port forwarding. The user is able to connect to and disconnect from the server by selecting appropriate menu items. When disconnected, facts may be asserted in the local copy of the database. On connecting all these facts are united with the facts on the server, and then conflicts are detected. For each conflict, the connecting agent asks the user which one of the set of conflicting facts should be used. When the user has specified this fact, all of the other facts in the conflicting set are retracted. Finally, the new database is broadcast to all the currently connected agents.

Conflicts are found as follows. For each predicate we first look to see if there are any conflicts by searching until the first is found. If there are no conflicts, we move on to the next predicate. If there are conflicts, then a new thread is forked to form a complete list of conflicts and to ask the user to choose between the conflicting facts.

Recall that a **`wait_for`** call detects when all the remaining threads are waiting for a certain variable. We exploit this by passing a variable to all forked threads and

then waiting for that variable on the main thread. This ensures that the main thread waits until all conflicts have been resolved before it broadcasts the new database to the agents.

The code which maintains the database and to detect and resolve conflicts is written in Prolog. The code to manage the GUI and the socket communication is written in C#. The Prolog and C# code interact using the scheme inherited from Prolog Café. That is, the C# code can call a Prolog predicate by using a method provided in the P# runtime-system, and the Prolog can call an arbitrary C# method by using the `:/2` predicate or equivalently the **`cs_method`**`/3` predicate.

5.2 An Object-Oriented Assistant

When a programmer begins learning to use a new C# namespace or Java package, they have to investigate how the classes interoperate. Often, having constructed an object, they need to find how to use it. They wish to know, for example, which methods it can be passed to, or which methods can be invoked upon it. Alternatively, they discover that they require an object of a certain type and need to find out how to obtain one: either by using a constructor or by invoking a method that returns an object of that type.

We discuss our implementation of a tool that allows a programmer to issue queries on a set of C# namespaces or Java packages. P#'s principal intended use is to couple a Prolog back-end to a C# front-end. The back-end, in this case, searches a database representing a namespace: a classic use of Prolog. The front-end consists of a graphical user interface: a standard use of C#.

If the tool was to be used to investigate only the C# namespaces, then we could use reflection to search for the fields and for the methods. We want, however, to develop a generic tool that can be used for multiple object-oriented languages; in particular we want the tool to search Java packages in addition to C# namespaces.

We now briefly summarize how these requirements are achieved, and then give more

details below. First, we compile the C# namespace data or Java package data into a file containing a set of Prolog predicates representing the types of the methods and the fields. Two such programs are required: one for C# and one for Java. Both of these use reflection and are written in C# and Java respectively. Then, in order to avoid stack overflow, this flat database must be converted into a tree structured database. Next, these facts are compiled into a set of C# classes, and these classes are compiled into a Windows DLL. Each namespace or package is compiled into a separate DLL. Finally, the graphical front-end is executed and the user enters a query. The DLLs corresponding to the required namespaces are loaded using **`load_assembly`**`/1` and a Prolog thread is spawned to execute the search. This thread passes solutions back to the user interface which displays a list for the user to select from. The solutions are passed via a synchronized queue. When the user has selected a field or method to investigate, an instance of the Internet Explorer Web browser is spawned which displays the documentation for that method or field.

Our database generation program reflects on the fields and methods of a namespace or package, storing this data in predicates of the form:

- classmember(⟨namespace⟩, ⟨class name⟩, static|instance, field, ⟨field name⟩(⟨field type⟩)).
- classmember(⟨namespace⟩, ⟨class name⟩, static|instance, method, ⟨method name⟩(⟨return type⟩, ⟨argument types⟩,...)).
- interfacemember(⟨namespace⟩, ⟨class name⟩, static|instance, field, ⟨field name⟩(⟨field type⟩)).
- interfacemember(⟨namespace⟩, ⟨class name⟩, static|instance, method, ⟨method name⟩(⟨return type⟩, ⟨argument types⟩,...)).
- derives(⟨superclass⟩, ⟨class⟩).

In the case of C# there are additional constructs beyond methods and fields to consider. C# **`struct`**s, **`enum`**s and **`delegate`**s are treated as classes (which is, roughly speaking, how they are implemented). Properties are dealt with as fields, however the get and set methods are also included as methods.

If we try to compile a large flat database to C# and then compile the corresponding C# classes into a P# program, we find that the stack overflows when the compiled P# Prolog program is executed. This is because very large disjunctions cause deep recursion. We solved this problem by writing a Prolog program which takes as input a generic Prolog database, and converts it into a form which will not cause stack overflow. This is a significant improvement of P#, as it enables users to query larger databases.

We convert the flat database into a balanced tree. The facts in the database are separated into those having different predicates. We then choose a number which we refer to as the *arity*, and which we will write as α. For each predicate we split the set of facts into sets containing α nodes each, with the exception that possibly one of the sets contains less than α nodes. For each of these sets of α or less nodes, we create a new clause which has as its head a newly chosen predicate and has as its body a disjunction of the members of the set of α nodes. We then take this new set of clauses, and split it up into sets containing α nodes, possibly with one of the sets containing less than α nodes. We continue constructing this tree until we reach a single node for which we create a clause having a head which is the predicate name that we started with. This is best illustrated by an example:

Suppose that we begin with the set of seven clauses in the left hand box of Figure 5.1, and choose an arity of 2. This is translated into the set of all the clauses in the outer right hand box. The four boxes inside this outer box correspond to different stages of the generation algorithm. The names of the predicates in the generated clauses can be decoded as follows. The first number appended to the original predicate name is the stage of the algorithm, that is the level in the tree. The second number is the number of the original clause that the new clause corresponds to.

By entering the above clauses into any Prolog implementation, the reader can verify that the query `?- a(X).` yields the solutions: $X = 1$, $X = 2$, $X = 3$, $X = 4$, $X = 5$, $X = 6$, $X = 7$ in that order. The stack no longer overflows as the execution path no longer involves large disjunctions.

We exploit the concurrency features of P# to allow solutions to the user's query to be passed from P# Prolog to C# via a synchronized queue. When a query is entered a

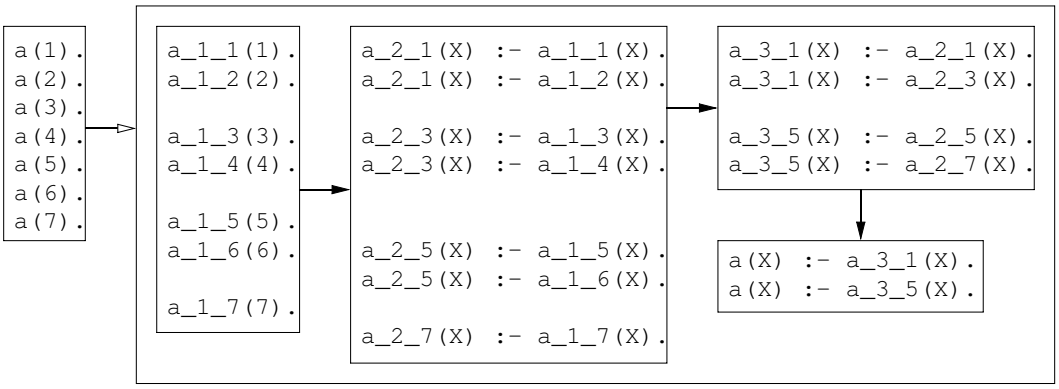

Figure 5.1: Example of the treeification algorithm at work

P# Prolog predicate is called which forks a thread which searches for solutions. This thread is passed a new uninstantiated variable which is used to coordinate communication between the P# Prolog and C# threads. The C# code then spawns a new C# thread which receives the solutions by repeatedly calling the `Receive()` method.

The user interface allows the user to select one of the following options:

- Find methods: This finds all of the methods in the specified classes or namespaces.
- Find methods which take arguments (unordered subset): The user enters a subset of the arguments of the method in any order.
- Find methods which take arguments (unordered): The user enters all of the arguments in any order.
- Find methods which take arguments (ordered): The user enters all of the arguments in the correct order.
- Find methods which return type: The user enters the required return type.
- Find fields: This finds all of the fields in the specified classes or namespaces.
- Find fields of type: The user enters the type of the required fields.

- Find classes: This finds all of the classes in the specified namespace.
- Find classes which define member: The user enters the name of the member being sought.
- Find superclasses of: The user enters a class and is given the chain of classes from that class to the root of the hierarchy.
- Find direct subclasses of: The user enters a class and is given its immediate subclasses.

Having entered this data, the user may specify a limit on the number of solutions to be returned and the set of namespaces and/or classes to search. If no namespaces are specified then the entire database is searched.

Solutions are passed from the P# Prolog thread to the C# thread and are added to a combobox (see Figure 5.2) as they are received. The user can then select an item of this box and click on a button which spawns an instance of Internet Explorer with the appropriate URL for the required item. In the case of C# this URL has protocol `mshelp://`. In the case of Java the URL is an anchor in a Javadoc generated HTML file.

The concurrency features of P# were exploited in order to improve efficiency by loading and then priming the databases on starting the application. As soon as the application starts the namespace databases are loaded one at a time. When they are all loaded, they are each primed by issuing a query that has no solution. This querying process has the effect of causing all of the C# classes in the database to be JIT compiled as each has to be accessed to ascertain that there is no solution. Because this process takes several minutes, each namespace database is separately locked by a mutex. When a query is issued by the user, this mutex is acquired by that process and locked until the query is completed. Thus, if a query is issued the priming process temporarily stops and then resumes when the query is completed. This optimization has little effect in cases where the user issues a query localized to a single namespace. If the query is over the entire C# API, however, there is a significant speed improvement as a result of the priming process. In the case of the query shown in the screen-shot, if the query

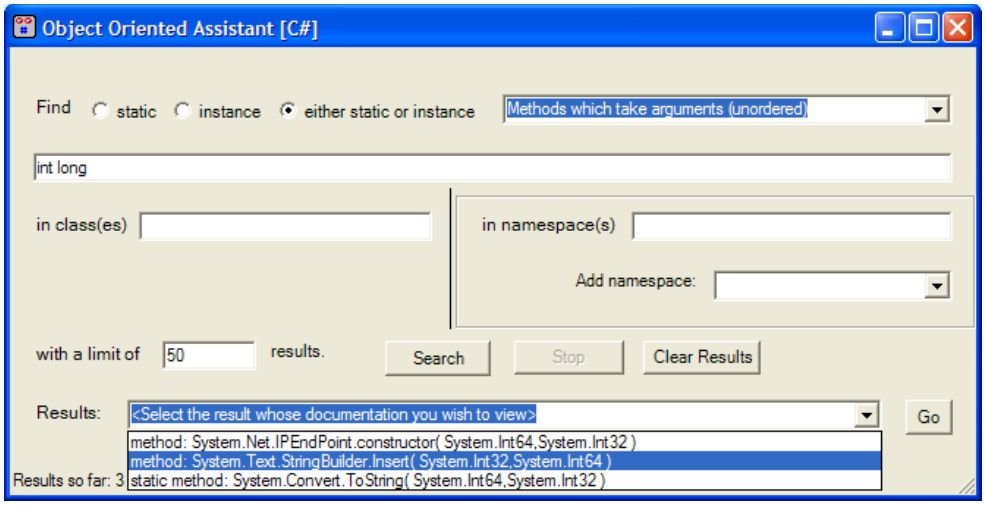

Figure 5.2: Screen-shot of the object-oriented assistant

is entered as soon as the application is started it takes roughly two minutes, and in the case that we wait for the databases to be primed first it takes two seconds. As a result of this scheme, the time during which a query is not being executed is not wasted.

5.3 A Class Hierarchy Viewer

We discuss an implementation of a tool that allows the part of the Java or C# class hierarchy surrounding a given class to be displayed. This tool operates as follows. The user is asked to provide a single class from the C# or Java class hierarchy and is then provided with a graphical inheritance diagram having their chosen class at the centre. In this application the Prolog back-end determines a suitable subset of the inheritance tree and then computes coordinates for each of the lines and text labels. The back-end then calls C# methods that render these graphically.

We need an algorithm to lay out the tree on the screen. The paper Functional Pearls: Drawing Trees [Ken96] gives an SML program for drawing trees in an aesthetically pleasing manner. The MLj homepage [MLj] includes full source code and an on-line

demonstration of this program. We first translated the SML program into Prolog by hand. This task was surprisingly straightforward, because of similarities between the functional and logical paradigms.

For example the SML type declaration

```
datatype 'a Tree = Node of 'a * ('a Tree list)
```

might have as an instance

```
Node( 5, [ Node( 4, [] ), Node( 3, [] ) ] ).
```

which could be represented in Prolog as

```
node( 5, [ node( 4, [] ), node( 3, [] ) ] ).
```

and the SML function declaration

```
fun movetree (Node((label,x), subtrees), x':real) =
    Node((label, x+x'), subtrees)
```

can be translated into the Prolog

```
movetree( node( pair( Label, X ), Subtrees ),
          Xprime,
          node( pair( Label, XSum ), Subtrees ) ) :-
    XSum is X + Xprime.
```

Higher-order functions can also be implemented in Prolog. For example, the `map` function maps a function and a list to a list containing the list of functional applications of the function to the items of the list. For example, `map( f, [a,b,c] )` evaluates to `[f(a), f(b), f(c)]`. We can implement the `map` function in Prolog as follows:

```
map( _, [], [] ) :- !.
map( F, [H1|T1], [H2|T2] ) :-
        ToCall =.. [ F, H1, H2 ],
        call( ToCall ),
        map( F, T1, T2 ).
```

In the Prolog case there is less type safety. Although higher-order programming of this type is possible in Prolog, it is a functionality that is little used in practice.

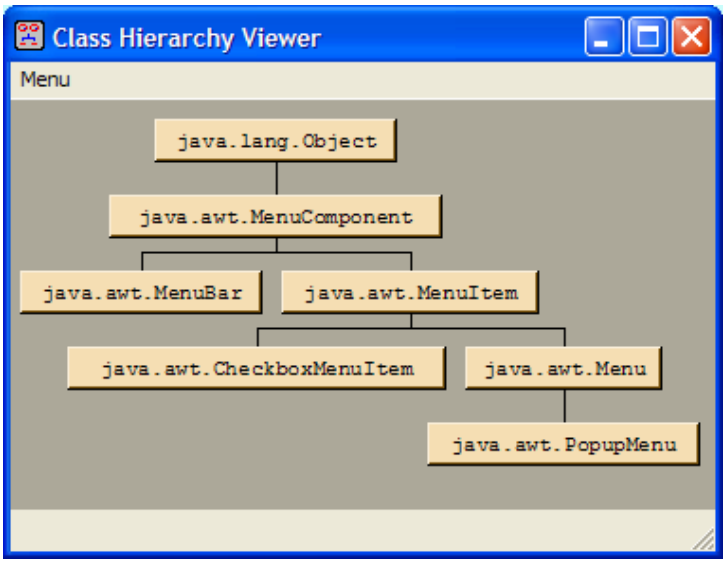

Figure 5.3: Screen-shot of the class hierarchy viewer

We considered various contrived schemes for deciding which nodes to draw. The simplest sensible approach that does not lead to an immense tree always being drawn is to draw the tree below the class of interest down to a certain depth, and to show the path to the `Object` class above the class of interest. We then position the scrollbars so that the class that we are interested in is central on the screen. If the user clicks on a class in the tree, the tree is redrawn with that class in the centre.

A screen-shot of the class hierarchy viewer is included as Figure 5.3.

5.4 Performance Measurement Before Optimization

We benchmarked concurrent P# against the Java-based Prolog systems Jinni 8.48 [Jin], MINERVA 2.4 and Prolog Café 0.4.4, and against the non-concurrent version of P#, as shown in Table 5.1.

The bottom row of the table indicates where appropriate the geometric mean average over the tests of the ratio of the time taken to execute the benchmark with P# to the time

taken to execute the benchmark with the tool heading the column. This is a measure of the speed of each tool relative to P#.

Some of the original benchmarks supplied with Prolog Café ran too quickly on our machine for the time to be measured. These benchmarks were replaced by identical benchmarks with the main code of the benchmark being run repeatedly by a tail-recursive loop. The second column of the table indicates, where this was done, how many runs of the benchmark the timing was taken over.

We used Sun's JVM (Java Virtual Machine), version 1.4.0 for Windows for the tests. The tests were carried out on a 2 GHz Pentium 4 machine with 512 Mb of memory running Windows XP Professional. All times are in milliseconds.

The results indicate that, on these benchmarks, P# has a speed comparable with Jinni and MINERVA and roughly double the speed of Prolog Café.

There is a small loss of performance due to the addition of concurrency support. This penalty is, in our opinion, acceptable, given the greatly enhanced possibilities for interoperation with C#, some of which we have presented in this thesis. Prolog programs that do not involve concurrency experience a relatively small overhead from these changes. In a concurrent program, the most significant overhead is due to extra work that must be done when a unification involves a concurrent variable. In particular, fields of the `VariableTerm` objects involved may need to be updated, and a call to **`global_assertz`**`/1` made. On the consuming side, after a wait has been woken it must check the other threads to see if a binding has occurred. It is, in our opinion, acceptable for there to be an extra overhead when a unification occurs, as the programmer knows that the unification will lead to a message being passed between threads.

We also investigated the relative sizes of the Java class files and C# executable file. There was little point comparing the size of the Java and C# source files as they are very similar. The class files produced by Prolog Café were on average 1.5 times larger than the executables produced by P#. The Prolog Café Jar file, which contains the compiled class files for the runtime-system, and library is roughly 3 Megabytes in size. The corresponding file for P#, the DLL, is roughly 1 Megabyte in size.

Table 5.1: Comparison of P# with other tools before optimization

Benchmark	repeat factor	P# time	Jinni time	MINERVA time	Prolog Cafe time	non concurrent P# time
boyer		2870	2370	844	3766	2792
browse		1380	1640	724	1052	1130
chat_parser	×10	1042	391	104	2932	1062
crypt	×10	68	47	120	250	78
fast_mu	×40	156	52	140	297	99
meta_qsort	×10	245	312	239	531	224
mu	×80	135	67	120	266	115
nreverse		57	172	94	120	31
poly_10		146	110	146	475	146
prover	×40	125	78	94	401	109
qsort	×20	47	47	99	130	31
queens (8 all)		78	52	167	151	68
queens (10 all)		1703	1177	1386	818	1443
queens (16 first)		1036	594	2198	500	891
query	×10	161	58	125	500	157
reducer		282	130	130	1078	380
tak		406	521	1474	495	365
zebra	×10	469	427	229	589	458
average speed, with P# normalized to 1.00		1.00	1.36	1.12	0.55	1.13

Chapter 6

Optimizing P#

Much effort has been invested in the optimization of Prolog implementations. Indexing [Han92] is an example of an optimization of the WAM.

In this chapter, we discuss a major optimization of P#. This optimization is based on the exploitation of semi-deterministic predicates. A predicate is *semi-deterministic* if it always either fails or succeeds with exactly one solution. If a predicate is semi-deterministic then there may be backtracking from one clause of the predicate to the next, if an earlier clause fails at some point. A semi-deterministic predicate which only calls other semi-deterministic predicates has the property that an individual clause will not be executed more than once by backtracking. In such cases we can do away with the Prolog stacks, which govern backtracking, and simulate in C# the fairly simple flow of control which is permitted for such a predicate. A predicate is *non-deterministic* if it may produce more than one solution.

A more specific class of predicates than the semi-deterministic predicates is that of the *deterministic* predicates. A predicate is deterministic if it always succeeds exactly once. Deterministic predicates occur frequently in idiomatic Prolog. Often, they are the result of coding a function in Prolog. When one wishes to code a predicate which will be used as a function, one generally expresses this as a Prolog predicate, some of whose arguments are *input* arguments, with the other arguments being *output* arguments. Input arguments are arguments which are known to be instantiated on entry into

the predicate, and output arguments are those which are not instantiated on entry into the predicate, but which will be instantiated on exit from the predicate. The property of an argument of being input or output is referred to as its *mode*.

Developer experience suggests that these functional predicates often perform some simple utility, that is, they frequently occur as leaf predicates. Such a predicate might, for example, concatenate two lists, or takes as arguments integers m and n and return the list of integers: $[m, m+1, \ldots, n-1, n]$. Another feature of leaf predicates is that, often, they are called frequently. Thus, if we can reduce the time taken to execute such predicates, we may effect a significant optimization.

6.1 Idiomatic Compilation

We now discuss how the compilation scheme presented in the preceding chapters can be changed so that certain predicates are compiled to C# code that is closer to the code which a C# developer would have produced. This new compiler has several phases.

The first phase compiles each predicate into an abstract syntax tree representation of a C# class containing a method called `idiomatic()` which consists of a block for each clause. Within each block, input variables are first extracted from the arguments, then each goal is executed in turn and finally output variables are copied back into the arguments. The second phase attempts to convert any recursive calls in the `idiomatic()` method into jumps back to the first block of the idiomatic method. If this proves possible, the third phase attempts to convert the body of the method into a **`while`** loop. The fourth phase then performs a liveness analysis, and the final phase converts the abstract syntax tree to C# code. The abstract syntax representation of C# just after the first phase can also be converted to C# code. Thus, we will be able to give an example of the state of the code at that stage.

We will refer to predicates which are compiled to more idiomatic C# as *idiomatic predicates*, and to those which continue to use the original compilation scheme as *non-idiomatic predicates*. If we are going to translate some semi-deterministic predi-

cates into more idiomatic C#, then we will find that when control passes from a non-idiomatic predicate to an idiomatic predicate, the arguments must be converted accordingly. A significant improvement in efficiency should be obtained if idiomatically compiled predicates work with native **`int`**s rather than with `IntegerTerm`s. Thus, when calling an idiomatic predicate which uses integers, from non-idiomatic code we should convert the `IntegerTerm`s to **`int`**s. In order to call a non-idiomatic predicate from an idiomatic one, we would have to start a new Prolog interpreter—which would lead to an unacceptable performance penalty.

In light of the above, it seems sensible to idiomatically compile a large a slice as possible at the bottom of the call stack. Thus, whenever a non-idiomatic predicate calls an idiomatic one, we convert its arguments; and an idiomatic predicate never calls a non-idiomatic one. When an idiomatic predicate calls another idiomatic predicate there is also sometimes a need for the conversion of arguments, for example, when a predicate passes an `int` to a predicate which takes a `term` as an argument.

The `exec()` method of a non-idiomatic predicate is replaced in the idiomatic case by a method similar to the following which calls the `idiomatic()` method.

```
public override Predicate exec( Prolog engine ) {
    int outarg3 = 0;
    bool success = idiomatic( ( arg1.Dereference() ),
        ((IntegerTerm)( arg2.Dereference() )).value( ),
        out outarg3 );
    if( success ) {
        arg3.Unify( new IntegerTerm( outarg3 ), engine.trail );
        return cont;
    } else
        return engine.fail();
    }
}
```

Arguments are always dereferenced before being passed on to the `idiomatic()` method, as they may be `VariableTerm`s instantiated to some other type of term. Also observe that `IntegerTerm`s are converted to native `int`s when that argument of the predicate is declared to be an `int`. If the argument is declared to be a `term`, no conversion is necessary. Floats and atoms are also treated in this way. The `exec()` method

shown above illustrates the restriction of idiomatic compilation to semi-deterministic predicates. The predicate can either fail or instantiate some of its arguments to new values. This is the only interaction the idiomatic predicate can have with the predicates above it in the call tree. Thus, there cannot be calls to the standard Prolog `assert/1` predicate or to other database modification predicates.

In order to call an idiomatic version of a predicate from another idiomatic predicate it is essential that we know its type and mode signature. Thus, when multiple files are compiled at different times, it is necessary for each file to see the type and mode signatures of the others.

6.1.1 Generating Naïve Idiomatic Code

We now discuss the first phase of the translation into idiomatic code. This produces code which despite being more recognizable as C# code is still not particularly idiomatic.

In the following, we assume that each clause of a predicate which is to be idiomatically compiled is a conjunction of goals. Some disjunctions can be converted into an equivalent predicate definition with more clauses. Other disjunctions and the if-then-else construct can be compiled by creating dummy predicates. See the section 6.1.6 for a discussion of dummy predicates.

A semi-deterministic predicate executes as follows. The first head goal which matches the calling query is found and then that clause begins to execute. If the end of the clause is reached with all the goals having succeeded then the predicate succeeds and exits at that point. If at any point one of the conjoined goals fails, and we have not encountered a cut, then we backtrack to the call to the predicate and execute the next clause that matches the calling query. If one of the goals fails and we have encountered a cut, then the predicate fails at that point. When all the matching clauses have been tried and have failed, the predicate fails and the call returns.

Figure 6.1 illustrates this control flow. The thick horizontal lines represent the execution of the different clauses of the predicate. The lines beginning in the middle of the

horizontal lines indicate control flows which can occur between goals in the clause. The lines beginning at the end of the horizontal lines indicate control flows which can occur just after the final goal of the clause, that is, success.

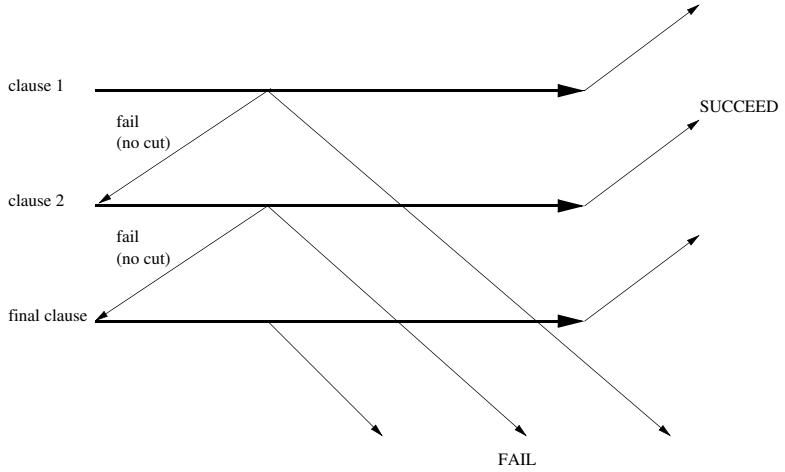

Figure 6.1: Control flow for a semi-deterministic predicate that only calls other semi-deterministic predicates

The difference with a predicate that may produce more than one solution because it calls predicates which themselves may have more than one solution, is that there may be backtracking to before a previous goal within a clause. This greatly complicates the control flow.

There are several ways in which this can be implemented. The most elegant implementation approach is probably to use exceptions. Each clause is compiled into a **`try`** block and a failure manifests itself as an exception throw. The **`catch`** block is either empty, allowing control to pass to the next **`try`** block; or contains the code for the next clause. The use of exceptions does, however, carry a performance penalty, and since our main objective is to improve efficiency, this is unacceptable. Such an approach would produce code similar to the following:

```
public bool idiomatic( ... ) {
    // clause 1
    try {
        if( !clause_1_goal_1() )
```

```
            throw new FailException( );
        if( !clause_1_goal_2() )
            throw new FailException( );
        return true;
    } catch( FailException ) {
    }
    // clause 2
    try {
        if( !clause_2_goal1() )
            throw new FailException( );
        return true;
    } catch( FailException ) {
    }
    return false;
}
```

Another approach would be to use **do** ... **while**(**false**) blocks. Each clause would be compiled into such a block, which executes its contents exactly once. We would then use the C# **break** statement to jump to the next block from the middle of a block. This is a dishonest approach, in that the **do** ... **while** construct is not intended to be used in this way. If we were translating to Java, this is probably the approach that would be used. Such an approach would produce code similar to the following:

```
public bool idiomatic( ... ) {
    // clause 1
    do {
        if( !clause_1_goal_1() )
            break;
        if( !clause_1_goal_2() )
            break;
        return true;
    } while( false );
    // clause 2
    do {
        if( !clause_2_goal1() )
            break;
        return true;
    } while( false );
    return false;
```

```
}
```

A final approach, and the one for which we opted, is to use the **`goto`** construct. The **`goto`** construct is shunned by many because of its tendency to produce unstructured code. For an example of this point of view, see [Dij68]. However, it is arguably acceptable to use the **`goto`** construct in a tightly controlled structured way, see [Knu74]. Indeed the C# language places significant restrictions on the use of the **`goto`** construct. It is not permitted to jump into a different block, for example. In particular it is not permitted to jump into or out of a loop. Thus, this is both an honest and flexible approach.

We structure the code in the following way. Each clause is compiled into a block of C# code. Each block is preceded by a label of the form `clause1`, `clause2`, ..., and so on.

The declaration of the idiomatic method should reflect which arguments are input arguments and which are output arguments. For example, if we are considering a predicate of arity two, whose first argument is an input argument, and whose second argument is an output argument, and both are integers, then we can use the following declaration:

```
public static bool idiomatic( int arg1, out int arg2 )
```

The first thing that must be done in each clause block is to find whether that clause matches the arguments that have been passed to the predicate. In this stage, we also want to unpack any lists or structures and extract the values of input variables embedded in such structures.

For each argument of the predicate, we recurse through the structure of the argument, keeping track at each point of the path through the structural tree of that argument to the current point. Thus, on encountering the head: `p( f([X|Y], 5) )`, when processing the first argument, on reaching the variable `Y` we have a path of `list( tail, structure( f/2, 1, arg(1) ) )`. This means that we start with the first argument. This is a structure with functor `f/2`, we take its first argument. This is a list, and we take its tail. In the C# code this is translated into: `Term Y = arg1[1].Tail`.

In reaching this point, we will have already generated code which checks that the first argument really does have a functor of `f/2`, and that its first argument is a list. The code, assuming that it occurs in the first clause, is as follows:

```
if( arg1.Functor() == null ) goto clause2;
if( !( arg1.Functor().Equals( "f" ) ) goto clause2;
if( arg1.Arity() != 2 ) goto clause2;
if( !arg1[1].IsList() ) goto clause2;
```

These **`if`** statements, which share a *then* clause, will be combined into one neat **`if`** statement in a later phase.

When the value of 5 is reached a C# statement is produced which tests that this part of the argument is 5: **`if`**`( !( arg1[2].Equals( 5 ) ) )` **`goto`** `clause2;`.

In order that all this works, the class `Term` in the runtime-system of P# is extended with several methods and properties. The `Head` and `Tail` properties return the head and tail of a `ListTerm`, failing with an exception if passed something other than a `ListTerm`. The `Term` class becomes an indexer, so that if the `Term`, `t`, is a `StructureTerm`, then `t[2]` returns the second argument of the structure. Methods called `Functor()` and `Arity()` are added to allow the string valued functor and arity of a structure to be determined.

An `Equals(string s)` method is added to the `Term` class. This returns true if the `Term` is a `SymbolTerm` representing a symbol with functor `s` and arity 0. A similar method is provided, which deals with integers. Finally an `Equals(Term t)` method is provided, which performs a content equality test.

Next, all of the variables which might be used within the clause are declared and those which are input values are initialized to the arguments passed to the idiomatic method.

As the clause is a conjunction of goals, each is executed in turn, using a **`goto`** to jump to the next clause if a goal fails. At the point that a cut occurs a Boolean variable called `cut`, which is initially set to false, is set to true. At the start of each clause after the first, we test the `cut` variable, and return false if `cut` is true, indicating that the predicate has failed. If there are any output arguments, then they must be set to default values before

the method returns, thus in order to fail we jump to a block at the end of the method labelled `fail:`.

For example, if `a/2` is idiomatic, then the call `a( 1, 2 )` is translated to:

```
if( !Predicates.A_2.idiomatic( 1, 2 ) ) goto <nextclause>;
```

When calling a C# method, some of whose arguments are marked `out` or `ref`, it is necessary to include the same `out` or `ref` keyword in the call. Thus, if `a/3` is idiomatic, with mode signature **`mode`**`( a( in, in, out ) )` then a call to `a/3` may be similar to the following:

```
if( !Predicates.A_3.idiomatic( Term.ToInt( X ),
                               new IntegerTerm( Y ),
                               out Z ) )
    goto <nextclause>;
```

Some type conversions may be necessary. In this case, the variable `X` is an `IntegerTerm` in the calling method, but an **`int`** in the method `A_3.idiomatic()`. The variable `Y` is an `int` in the calling method and an `IntegerTerm` in the method `A_3.idiomatic()`.

When one of the arguments being passed to the predicate being called is an `out` argument which is a structure, for example a call to `callee_predicate( a(X, [Y|Z]) )`, we need to create a dummy variable to hold the result. This code fragment is translated into something similar to the following:

```
Term formal_goal1_arg1;
if( !( Predicates.CalleePredicate.idiomatic(
        out formal_goal1_arg1 ) ) ) {
    goto fail;
}
if( !( ((formal_goal1_arg1).Functor( ) != null
     && ( (formal_goal1_arg1).Functor( ) ).Equals( "a" )
     && (formal_goal1_arg1).Arity( ) == 2
     && ((formal_goal1_arg1)[2]).IsList( )) ) ) {
    goto fail;
}
Z = ((formal_goal1_arg1)[2]).Tail;
Y = ((formal_goal1_arg1)[2]).Head;
X = (formal_goal1_arg1)[1];
```

We translate the Prolog relational infix predicates: `=:=/2`, `=\=/2`, `</2`, `=</2`, `>/2` and `>=/2` directly into their corresponding C# operators which have the same semantics.

We translate the infix **`is`**`/2` predicate which instantiates an uninstantiated variable on its left hand side to the result of evaluating the arithmetic expression on its right hand side. If the term on the left hand side of the **`is`**`/2` predicate is instantiated then **`is`**`/2` acts in the same way as `=:=/2`. That is, it tests for equality. Thus, depending on the state of instantiation of the left hand side we need to translate a call to **`is`**`/2` to either an assignment or an arithmetic comparison. In order for it to be able to make this decision the compiler keeps track of the state of knowledge regarding whether variables are instantiated or not throughout the course of the execution of the clause. At the beginning of the clause we know that `in` arguments are fully instantiated and that `out` arguments are uninstantiated variables. If during the course of the execution of the clause a predicate is called all of whose arguments are either `in` arguments or `out` arguments, then we know that all the arguments are instantiated after the call. If a variable occurs which has not previously appeared at all in either the head or a previous goal to the one under consideration, we know that this variable is uninstantiated.

A similar issue arises with the translation of the unification infix predicate, `=/2`. We recurse through the structure of the term on both sides of the unification, failing at any point if we find unmatched functors.

If we have a compound term on one side and on the other side we have a term which is not compound and also not a variable, an integer for example, the unification will fail and we translate it to a jump to the next clause. If we reach a fully instantiated variable on one side and a fully instantiated compound term on the other, we translate this into an equality test. If one side is a fully instantiated variable and the other is a partially instantiated compound term, for example:

```
Y = f( 1, 2 ),
f( A, B ) = Y.
```

then this is a situation similar to that encountered when arguments to a clause are unpacked at the beginning of the clause. Indeed, we use this unpacking code to generate the following:

```
if( !( ((Y).Functor( ) != null &&
       ( (Y).Functor( ) ).Equals( "f" ) &&
         (Y).Arity( ) == 2) ) ) {
     goto <nextclause>;
}
Term A = (Y)[1];
Term B = (Y)[2];
```

If there is an uninstantiated variable on one side and a compound term on the other, we translate this into an assignment, where the compound term is assigned to the variable.

The only remaining cases are those where neither side is a compound term. If we know that both of the terms are instantiated variables or ground terms, we test them for equality, succeeding if they are equal and failing if they are not. If we know that one side is instantiated and the other is an uninstantiated variable, then we assign the instantiated value to the uninstantiated variable. Finally, if both sides are uninstantiated variables we currently fail to idiomatically compile the predicate. This situation occurs rarely, and a translation which changed all of the following occurrences of one of the variables into the other could lead to aliasing problems if another predicate was called with these two variables.

After all of the goals have been executed, we need to assign the values held by variables representing output arguments to the output arguments of the idiomatic method. The idiomatic method returns true or false on success or failure respectively.

At this stage of the compilation process, the following predicate, which can be used to find the length of a list,

```
len( [], Z, Z ).
len( [_|T], A, Z ) :-
    A1 is A + 1,
    len( T, A1, Z ).
```

has been translated into the following code:

```
namespace JJC.Psharp.Predicates {

using JJC.Psharp.Lang;
```

```
using Predicates = JJC.Psharp.Predicates;

public class Len_3 : Predicate {

    public Term arg1, arg2, arg3;

    public Len_3(Term a1, Term a2, Term a3,
                 Predicate cont) {
        arg1 = a1;
        arg2 = a2;
        arg3 = a3;
        this.cont = cont;
    }

    public L_3(){}
    public override void setArgument(Term[] args,
                                     Predicate cont) {
        arg1 = args[0];
        arg2 = args[1];
        arg3 = args[2];
        this.cont = cont;
    }

    public static bool idiomatic( Term arg1, int arg2,
                                  out int arg3 ) {
        bool cut = false;
        clause1: {
            if( !( ( arg1 ).Equals( "[]" ) ) ) {
                goto clause2;
            }
            int Z = arg2;
            cut = true;
            arg3 = Z;
            return true;
        }
        clause2: {
            if( !( (arg1).IsList( ) ) ) {
                goto fail;
            }
            Term _1 = (arg1).Head;
            Term T = (arg1).Tail;
            int A = arg2;
```

```
            int Z;
            if( cut ) {
                goto fail;
            }
            int A1 = ( A + 1 );
            if( !( Predicates.Len_3.idiomatic( T,
                    A1, out Z ) ) ) {
                goto fail;
            }
            arg3 = Z;
            return true;
        }
        fail: {
            arg3 = 0;
            return false;
        }
    }

    public override Predicate exec( Prolog engine ) {
        int outarg3 = 0;
        bool success = idiomatic(
           ( arg1.Dereference() ),
           ((IntegerTerm)
                ( arg2.Dereference() )).value( ),
          out outarg3 );
        if( success ) {
            arg3.Unify( new IntegerTerm( outarg3 ),
                        engine.trail );
            return cont;
        } else
            return engine.fail();
    }

    public override int arity() { return 3; }

    public override string ToString() {
        return "len(" + arg1 + ", " + arg2 + ", " +
               arg3 + ")";
    }
}

}
```

6.1.2 Coalescing Adjacent `if` Statements

The way in which structures and lists are unpacked in the initial processing for each clause leads to a number of **`if`** statements all of which branch to the same place (the next clause) and have no **`else`** clause. In order to rewrite the code as a **`while`** loop where possible, it helps to rewrite these **`if`** statements as a single **`if`** statement.

Written in the intermediate language, the **`if`** statements are of the form:

```
if( not( Cond1 ), goto( nextclause ) )
if( not( Cond2 ), goto( nextclause ) )
...
if( not( Condn ), goto( nextclause ) )
```

These are equivalent to the **`if`** statement:

```
if( not( Cond1 ) || not( Cond2 ) || ... || not( Condn ),
    goto( nextclause ) )
```

where `||` is the or shortcut operator, which stops evaluating its arguments when it becomes apparent what the final result will be.

By considering all of the possible execution paths it can be shown that De Morgan's rule applies, despite the use of the shortcut operators, and that this **`if`** statement is equivalent to the more succinct statement:

```
if( not( Cond1 && Cond2 && ... && Condn ),
    goto( nextclause ) )
```

When the initial unpacking code, in the intermediate language, of the clause block is generated the **`if`** statements are often separated by declarations of variables. Typically, these declared values are used in the conditions of later **`if`** statements as well as in the subsequent code.

We could coalesce only **`if`** statements which are adjacent, leading to several **`if`** statements separated by declarations of variables. We want, however, all of the **`if`** statements in the prologue to collapse to a single **`if`** statement, so that it may be possible to rewrite some of the blocks as a **`while`** loop. This can be achieved by collecting all of

the **`if`** statements together. It is necessary not to use the declared variables in the **`if`** statements, but rather to extract the values directly from the arguments again each time they are required in the condition of an **`if`** statement. This results in **`if`** conditions which are longer and also degrades performance slightly. Both of these penalties are acceptable, we feel, if the code can be written as a **`while`** loop.

6.1.3 Tail-Recursion Converted to Iteration

Often a predicate will contain a recursive call. In this case we have to decide between translating the call into recursion or iteration in the C# code. Recursion is easier from an implementation point of view, however iteration is preferable as it avoids the risk of stack overflow and is often more efficient. The only case where translation to iterative code can be done in a natural way is when the recursive call occurs as the final goal of a clause. This is because at such a point we can merely jump back to the first clause having modified the arguments appropriately and maintain the same semantics for the code. Fortunately, for efficiency reasons, tail-recursion is the most common form of recursion in Prolog code.

In order to translate a recursive call into iterative code, it is also necessary that any output argument in the head of the tail-recursive clause, is matched by an identical output argument in the recursive call itself. For example, if we have a predicate `p/2` where only the second argument is an output argument, then the clause: `p( X, Y ) :- X1 is X - 1, p( X1, [Y] )` cannot be converted to iterative code (using the current scheme), whereas `p( X, Y ) :- X1 is X - 1, p( X1, Y )` can. In this latter case the code generated for the recursive call, prior to liveness analysis and rewriting as a **`while`** loop, will be: `arg1 = X1;` **`goto`** `clause1;`.

The conversion to iteration would not be performed by the C# compiler if it were not performed by the idiomatic code generator.

6.1.4 Rewriting Blocks as a `while` Loop

In cases where the tail-recursion optimization has been applied, the intermediate code generated often contains code of the following form (where we write the code in its C# form):

```
clause1: {
    if( <Condition> ) goto clause2;
    <Code Block 1>
    return true;
}
clause2: {
    <Code Block 2>
    goto clause1;
}
```

Code of this form can be rewritten as the following, provided that there are no other **`goto`** statements in the code:

```
while( <Condition> ) {
    <Code Block 2>
}
<Code Block 1>
return true;
```

Note that there is no code before the **`if`** statement in `clause1` because of the way we have coalesced the **`if`** statements and collected them at the beginning of the block. If there were we would have difficulties with scopes of variables, because such code would have to appear both before and within the while loop. We would also run into difficulties when considering tail-recursive predicates which have more than one base case.

The rewrite can be performed because the **`while`** loop above can be rewritten as:

```
loop: if( !<Condition> ) goto end;
      <Code Block 2>
      goto loop;

end:  <Code Block 1>
      return true;
```

which can be rewritten as:

```
loop: if( <Condition> ) goto label;
      <Code Block 1>
      return true;
label:<Code Block 2>
      goto loop;
```

Usually, there are other **goto**s, namely statements of the form

```
if( cut ) goto fail;
```

which occur in the C# code for all but the first clause. Since the `fail:` block is usually small, it is acceptable to in-line this block wherever a **goto** `fail` occurs.

We refer to a clause which ends in a tail-recursive call as a *step case*, and any other clause as a *base case*. Recursive predicates which have two clauses tend to have a base case as their first clause and a step case as their second clause, as above. If a recursive predicate has more than two clauses, it is usual for the predicate to be in the form of a number of base cases followed by a number of step cases. Even when this is not the case, it is common for the clauses to be mutually exclusive, that is, for any set of arguments, only one clause can match. In this case, we can re-order the clauses so that all of the base cases come before all of the step cases.

A predicate with a number of base cases followed by a number of step cases is first translated into code which has the following general form:

```
// base cases
clause1: {
    if( <Condition B1> ) goto clause2;
    <Code Block B1>
    return true;
}
clause2: {
    if( <Condition B2> ) goto clause3;
    <Code Block B2>
    return true;
}
...<more base cases>...
```

```
clause<n>: {
    if( <Condition Bn> ) goto clause<n+1>;
    <Code Block Bn>
    return true;
}

// step cases
clause<n+1>: {
    if( <Condition S1> ) goto clause<n+2>;
    <Code Block S1>
    goto clause1;
}
clause<n+2>: {
    if( <Condition S2> ) goto clause<n+3>;
    <Code Block S2>
    goto clause1;
...<more step cases>...
clause<m>: {
    if( <Condition Sm> ) goto fail;
    <Code Block Sm>
    goto clause1;
}

fail: {
    <Fail Block>
    return false;
}
```

This can be re-written as the following code:

```
while( <Condition B1> && <Condition B2> && ... && <Condition Bn> ) {
    if( !<Condition S1> ) {
        <Code Block S1>
    }
    else if( !<Condition S2> ) {
        <Code Block S2>
    }
    ...
    else if( !<Condition Sm> ) {
        <Code Block Sm>
    }
    else {
```

```
        <Fail Block>
        return false;
    }
}
if( !<Condition B1> ) {
    <Code Block B1>
}
else if( !<Condition B2> ) {
    <Code Block B2>
}
...
else if( !<Condition Bn> ) {
    <Code Block Bn>
}
return true;
```

Observe that the situation where we have just base cases is not a special case of this scheme because of the way in which the final clause block would branch to the fail block. Indeed, there is nothing to be gained from applying the re-write in this case, and in fact the most logical extension to this case results in rather odd code. Thus, in this case we do not apply the re-write.

One major impediment to the generation of **`while`** loops is instances where there are other branches to the next clause within the code for a clause. This occurs when a non-tail-recursive call is made at some point in the body of one of the clauses. This is not a problem, provided that the non-tail-recursive call cannot fail. If it cannot fail, then the **`goto`** will never be executed. Thus, we have a motivation for distinguishing deterministic predicates from those that are merely semi-deterministic.

6.1.5 Liveness Analysis

The code produced by the above tends to have more variables than are necessary. Often code similar to the following is produced.

```
int X = arg1;
int _1 = arg2;
int Z;
```

```
Z = X + 1;
arg1 = Z;
```

where `_1` is never used. This can be replaced by `arg1 = arg1 + 1` by applying liveness analysis [App98] to the code. This is a standard technique which involves analyzing which variables are live at the same time at each point during the execution of the program. If two variables are never live at the same time, then one of the variables can be consistently changed to the other.

A control flow graph is generated, then the set of liveness equations is solved iteratively with the order of nodes being the reverse of the order that statements were encountered in the generation of the control flow graph. This significantly reduces the number of steps needed to solve the equations. Then for each type, an interference graph is generated for the variables of that type, and coloured with the argument variables: `arg1`, and so on, pre-coloured. A map is then obtained which maps the set of variables to a smaller set of variables, and this map is applied to the code being rewritten and the number of variables used is usually decreased. There is practically no limit to the number of registers available, so during the colouring we start with the node of minimum degree and work up to the node of maximum degree. Finally, dead code, including assignments of the form `x = x` are removed.

We also strip out statements involving the `cut` variable, if they are superfluous. The `idiomatic` method always begins with the statement **`bool`** `cut =` **`false`**`;`, and the only other statements involving `cut` that can occur in the code are `cut =` **`true`**`;` and **`if`**`( cut )` **`goto`** `...`. If we find a example of the latter which can never be proceeded by an instance of `cut =` **`true`**`;` then we remove that **`if`** statement.

Using the **`while`** loop rewrite and liveness analysis, the `len` predicate, which we mentioned earlier:

```
len( [], Z, Z ).
len( [_|T], A, Z ) :-
    A1 is A + 1,
    len( T, A1, Z ).
```

has the idiomatic method shown below:

```
public static bool idiomatic( Term arg1, int arg2, out int arg3 ) {
    while( !( (( arg1 ).Equals( "[]" )) ) ) {
        if( !( ((arg1).IsList( )) ) ) {
            arg3 = 0;
            return false;
        }
        arg1 = (arg1).Tail;
        arg2 = ( arg2 + 1 );
    }
    {
        arg3 = arg2;
        return true;
    }
}
```

The extraneous brackets, `{` and `}`, at the end of this method are necessary in some cases in order to avoid a difficulty with C#'s variable scoping rules. Notice also that the liveness analysis has determined that the head of `arg1` should be removed, and that `arg2` should be incremented in each iteration of the loop.

6.1.6 Compiling Disjunctive Constructs and the `not` Construct

Recall that to compile the disjunction operator, denoted by `;` and the if-then-else construct,
`(if) -> (then) ; (else)` we need to create dummy predicates. Similarly for the **`not`** construct: **`not`**`(.)` or `\+(.)`.

For example, the predicate

```
a(X, Y) :- b, ( c(X) ; d(Y) ), e.
```

would be converted into

```
a:- b, dummy_a_0_1( X, Y ), e.
dummy_a_0_1( X, Y ) :- c( X ).
dummy_a_0_1( X, Y ) :- d( Y ).
```

This is exactly what Prolog Café does to simplify its compilation algorithm. However we have an additional problem. We will refer to a predicate in `a`'s position as the *calling predicate*. If we are to compile the calling predicate idiomatically, we must also compile the dummy predicate idiomatically. Thus, we need to know the modes and types of the arguments of the dummy predicate. If we cannot infer some of the types, we can use the generic type, `term`. However, if some of the modes cannot be inferred from the calling clause we have to abandon the idiomatic compilation of the calling predicate.

An argument marked `in` is assumed to be fully instantiated on entry into the relevant predicate. An argument marked `out` is assumed to be an uninstantiated variable on entry into the relevant predicate and fully instantiated on exit from the predicate if the predicate succeeds.

We use the original P#/Prolog Café predicates which take as input a clause possibly containing disjunctive constructs or the **`not`** construct, and return a modified clause together with a set of new dummy clauses, neither of which contain these constructs.

For each argument in a call to a dummy predicate, we try to infer either that it can be given the `in` mode or that it can be given the `out` mode.

If all of the variables occurring in the argument occur as part of `in` arguments in the calling predicate, then we can infer that the argument to the dummy predicate is an `in` argument.

With the above definitions of the `in` and the `out` modes, any variable which has occurred in a previous goal in the same clause is known to be fully instantiated. This is because if it is an `in` argument it is known to have been instantiated before the earlier goal, and the goal cannot uninstantiate it. If it is an `out` argument it is known to be instantiated after the earlier goal, and no subsequent goal in the relevant clause can uninstantiate it. Therefore if all of the variables occurring in an argument in the call to the dummy predicate

1. occur as `in` arguments in the head of the calling predicate; or
2. occur earlier in the body of the calling clause

then we can infer that that argument to the dummy predicate is an `in` argument.

We can infer that a single variable argument of a dummy predicate is an `out` argument when it:

1. is an `out` argument of the calling predicate; and
2. is known to be uninstantiated on entry into the dummy predicate; and
3. is known to be instantiated on exit from the dummy predicate.

We can infer condition 2 when the variable does not occur earlier in the calling clause.

We can infer condition 3 when the variable does not occur later in the calling clause, since it must be instantiated when the clause exits. We can also infer condition 3 when the variable occurs in the directly following goal as an `in` argument.

This only leaves two cases where an inference cannot be made from the above:

The first such case is where there is more than one variable embedded in the argument to the dummy clause and they are not all input variables.

The second case is where the argument is a single variable which:

1. is an `out` argument of the calling predicate; and
2. does not occur earlier in the calling clause; and
3. occurs later in the calling clause, but not as an input argument of the directly following goal.

In this case we can look at the predicates involved in the disjunction or **`not`** construct.

In the construct: `b(X); c(X)`, where `X` is an input argument of both `b/1` and `c/1`, `X` must be an input argument of the dummy predicate. If it is an output argument of both, then `X` must be an output argument of the dummy predicate. If one is an input argument and the other is an output argument we cannot infer either mode and therefore cannot compile such code.

The construct: `\+( a( X ) )` has to have `X` as an input argument since the **`not`** construct cannot instantiate its variables. This is because this construct is equivalent to

```
( a( X ), !, fail ); true.
```

The construct: `a( X ) -> b( X )` is equivalent to

```
( a( X ), !, b( X ) ); fail
```

If `X` is an input argument of `a/1` or of `b/1`, then it is also an input argument of the dummy predicate. If it is an output argument of both then the code is not consistent with the modes and it will not be compiled.

6.1.7 Multiply Moded Idiomatic Predicates

Often we wish to give a predicate more than one mode. An example of this is the built-in `=..` predicate. This can take a structure and gives us a list whose head is that structure's functor and whose tail is the structure's arguments. For example:

```
?- f( 1, 2, 3 ) =.. L.

L = [f,1,2,3].
```

In this usage, the `=..` predicate has the mode signature: `in, out`.

Alternatively the predicate takes a functor and its arguments and gives us the corresponding structure. For example:

```
?- S =.. [f, 1, 2, 3].

S = f(1,2,3).
```

In this usage, the `=..` predicate has the mode signature: `out, in`.

If more than one mode is declared for a predicate, we translate it into several `idiomatic()` methods, distinguished by the use of the `out` modifier in their declarations. For the `=..` predicate, these would have signatures:

```
public static idiomatic( Term arg1, out Term arg2 );
public static idiomatic( out Term arg1, Term arg2 );
```

In the C# language, these are different methods. Because in C# we must also use the `out` modifier when calling the method, we are able to select the correct method to call based on the state of instantiation of the arguments to be passed to the method. This manifests itself as an **`if`** statement in the `exec()` method of the idiomatic code for the predicate. When calling a multiply moded predicate from an idiomatic method, if we know the state of instantiation of its arguments, we do not require an **`if`** statement.

6.1.8 Type Consistency

In the intermediate code, variables are represented as tuples giving their name and type and if appropriate which clause they occur in. When the intermediate code is translated to C# care is taken to check that these types match up. A predicate in the idiomatic compiler finds the overall type of each relevant expression and a conversion is inserted if two expressions do not have the same type when they should.

This phase also deals with the question of whether the `==` operator or the `Equals()` method should be used for comparisons.

6.2 Example Code—The Eight Queens Problem

Suppose that we are using the following code to solve the Eight Queens Problem:

```
queens(N,Qs) :-
    range(1,N,Ns),
    queens(Ns,[],Qs).

queens([],Qs,Qs).
queens(UnplacedQs,SafeQs,Qs) :-
    select(UnplacedQs,UnplacedQs1,Q),
    not_attack(SafeQs,Q),
    queens(UnplacedQs1,[Q|SafeQs],Qs).

mode( not_attack( in, in ) ).
type( not_attack( term, int ) ).
not_attack(Xs,X) :-
```

```
    not_attack(Xs,X,1).

mode( not_attack( in, in, in ) ).
type( not_attack( term, int, int ) ).
not_attack([],_,_) :-
    !.
type( not_attack/3, 2, [N1=int] ).
not_attack([Y|Ys],X,N) :-
    X =\= Y+N,
    X =\= Y-N,
    N1 is N+1,
    not_attack(Ys,X,N1).

select([X|Xs],Xs,X).
select([Y|Ys],[Y|Zs],X) :-
    select(Ys,Zs,X).

mode( range( in, in, out ) ).
type( range( int, int, term ) ).
range(N,N,[N]) :- !.
type( range/3, 2, [M1=int] ).
range(M,N,[M|Ns]) :-
    M < N,
    M1 is M+1,
    range(M1,N,Ns).
```

This code, less the mode and type declarations, forms one of the benchmarks provided with Prolog Café. Observe that we have given mode and type declarations to the predicates `not_attack/2`, `not_attack/3` and `range/3` as these are the predicates which can be idiomatically compiled. The type declarations with one argument refer to the types of the arguments of the predicate as a whole. The type declarations with three arguments: the functor, the clause number and the types of the variables, give the types for variables in the clause of that number which cannot be inferred from the type declaration for the clause as a whole. In fact, it is possible to infer that `N1` and `M1` are `int`s, see the section on future work. If a type is not given, the generic type, `term`, is used.

The predicate `select/3` cannot be idiomatically compiled as it is non-deterministic. Since `select/3` occurs below `queens/2` and `queens/3` in the call tree, these two predicates cannot be idiomatically compiled. All non-idiomatic predicates continue to be compiled using the original Prolog Café/P# compilation scheme.

The predicate `range/3` is compiled into the following method. This predicate takes a pair of integers and produces the list of consecutive integers starting with the first integer and ending with the second.

```
public static bool idiomatic( int arg1, int arg2, out Term arg3 ) {
    bool cut = false;
    clause1: {
        if( !( (arg2 == arg1) ) ) {
            goto clause2;
        }
        arg3 = new ListTerm( new IntegerTerm( arg1 ), Term.Nil );
        return true;
    }
    clause2: {
        Term Ns;
        if( !( arg1 < arg2 ) ) {
            goto fail;
        }
        int M1 = ( arg1 + 1 );
        if( !( Predicates.Range_3.idiomatic( M1, arg2,
                                             out Ns ) ) ) {
            goto fail;
        }
        arg3 = new ListTerm( new IntegerTerm( arg1 ), Ns );
        return true;
    }
    fail: {
        arg3 = null;
        return false;
    }
}
```

Notice that we could not compile this into a **`while`** loop because the output argument in the recursive call is not the same as in the head of the clause containing the call.

The predicate `not_attack/3` is translated into a **`while`** loop. This predicate checks for a given configuration of queens generated by `select/3`, that no two queens are attacking one another.

```
public static bool idiomatic( Term arg1, int arg2, int arg3 ) {
    while( !( (( arg1 ).Equals( "[]" )) ) ) {
        if( !( ((arg1).IsList( )) ) ) {
            return false;
        }
        int Y = Term.ToInt( (arg1).Head );
        arg1 = (arg1).Tail;
        if( !( arg2 != ( Y + arg3 ) ) ) {
            return false;
        }
        if( !( arg2 != ( Y - arg3 ) ) ) {
            return false;
        }
        arg3 = ( arg3 + 1 );
    }
    {
        return true;
    }
}
```

6.3 Comparison with Mercury

Work has been done on translating Mercury [HCS+] to high-level C, see [HS02]. That paper lists the advantages of translation to higher-level code. Generating low-level code usually leads to the Prolog compiler having to do more work, producing less readable code and often ending up working against the compiler of the language that Prolog is being compiled to.

As with Mercury we exploit type and mode information, but we do not attempt to idiomatically compile non-deterministic predicates. This is because our existing scheme for compilation of such predicates is as efficient as a more idiomatic compilation. P# generates highly idiomatic and readable code in many instances, often being able to

translate a simple predicate into a **`while`** loop. As such simple predicates often occur as leaf predicates that are called frequently, this significantly reduces the number of method calls, and allows us to avoid this overhead.

As we do not need to implement cuts for non-deterministic predicates, there is no need to unwind the stack when a P# cut occurs as there is when a commit occurs in a non-deterministic Mercury predicate. Thus, with P# Prolog, testing a cut flag after each failure is an acceptable solution.

6.4 Performance Measurement After Optimization

We benchmarked both the idiomatic and original versions of P#, against Jinni 2004 [Jin] (using Sun's Java SDK 1.4.2), MINERVA 2.4 [MIN] and SICStus Prolog 3.10.1 [SIC] as shown in Table 6.1. The speed-up column gives the factor by which P# is speeded-up by the optimization.

Table 6.1: Speed-up due to the use of idiomatic code and mode/type declarations (times in ms)

Benchmark	idiomatic P# time	original P# time	P# speed-up	Jinni time	Minerva time	SICStus time
browse	125	1360	11	1250	703	63
poly_25	1609	6781	4.2	6172	3500	226
queens (10 all)	438	1954	4.5	1250	1609	78
queens (16 first)	203	1234	6.1	734	1078	63
nreverse (2000)	485	4922	10	8047	921	62
tak	31	10969	350	11094	3938	437
zebra	140	140	1.0	62	78	16

The tests were carried out on a 2 GHz Pentium 4 machine with 512 Mb of memory running Windows XP Professional. All times are in milliseconds.

The performance of the `zebra` benchmark is not changed as its computational predicates are all non-deterministic. The speed-ups of the `queens` and `browse` benchmarks are almost entirely due to the idiomatic compilation of the `not_attack/3` predicate

and the list concatenation predicate respectively. All of the predicates of the `poly_25` benchmark, which computes $(1+x+y+z)^{25}$ symbolically, could be idiomatically compiled.

It should be noted that the optimizations we employed are of particular benefit to numerical code and that Jinni, MINERVA and SICStus Prolog did not have the benefit of mode or type declarations. Observe that the 'most numerical' benchmark, `tak`, which computes the Takeuchi function with arguments 24, 16 and 8, experienced the most significant speed-up. However, some of the other benchmarks involve list operations, and these too were speeded up. The idiomatically compiled list operation predicates can operate on heterogeneous lists.

Chapter 7

Conclusions

7.1 Translating Prolog to C# Source Code

Prolog is, as a language, particularly suited to solving problems involving logical deduction from a set of facts. There are many cases where a program such as this requires a modern user interface or sophisticated networking capabilities. By allowing interoperation between Prolog and C#, this can be easily achieved.

We achieved such interoperation by porting the Prolog to Java translator Prolog Café to a similarly bootstrapped Prolog to C# translator. Both the translator, written in Prolog/LLP and the run-time system, written in Java, had to be ported. In the former case, this involved changing code which produces Java to code which produces C#. In the latter case, this involved changing Java code into equivalent C# code.

We investigated the possibility of using the constructs which C# possesses and Java does not, to improve the C# code. While some of these constructs could be used to cosmetically improve the code of P# and the code that it generates, few were of any use for improving its efficiency. One that could have been used to improve efficiency, namely, pointers, would have resulted in unsafe code—something that we wished to avoid.

We have given several examples in which a Prolog back-end is coupled with a graphical

C# front-end. These demonstrate that P# is a useful tool for Prolog–C# interoperation. It is possible in addition to exploit other features of C# in this way, such as its rich support for networking.

A performance measurement shows that P# has a similar level of efficiency to other Prolog implementations available that are based on an modern object-oriented language.

Possible extensions are detailed in the following subsections.

7.1.1 Security

The C# language and the .NET platform have sophisticated mechanisms for enforcing security requirements. Among these is the ability to add security meta-data to a C# class. This might, for example, indicate that a method is only allowed to write to a single, named, file. Recent versions of the Windows operating system have an area of the file-system referred to as *isolated storage*. Such areas are intended for the storage of data such as application initialization data files. An application is granted permission to modify its own isolated storage files, without being allowed to write to those of another application. A possible future research question is: what is the best way to add language constructs to P# that allow such security requirements to be expressed and then added to the generated C# classes?

The C# language also has support for versioning and strong names. Work could be done on adding such a capability to P#.

7.1.2 Interoperation with Other APIs

The success of a new language hinges on its interoperation with APIs which are used by programmers. One future direction of research could be integrating Prolog for the ADO.NET [Sce01] database architecture of the .NET Framework. This would be beneficial for those wishing to use P# in a setting where large quantities of data need to be processed. We would also like to support the use of legacy database technology.

It is desirable for a language to be able to easily interoperate with foreign data formats. A P# programmer currently has access to the extensive XML namespaces of the .NET libraries. However, it has been argued that XML support should be incorporated into the core of the C# language, hence the Cω [Cw] language. Thus, perhaps, XML support ought to be incorporated into the P# language.

7.2 Concurrency

We have modified the P# engine in order to make it thread safe; and have added several built-in predicates to P# in order to allow concurrent code to be written. These predicates are as follows:

- **`fork`**`/1` and **`fork`**`/2`: these run their first argument on a new thread. Any uninstantiated variables in this argument become potential message channels. Such a forked call cannot return terms through output variables in the usual way, so such uninstantiated variables are only required in the structure for establishing a message channel.

- **`stop`**`/1`: this can be used to stop a thread.

- **`sleep`**`/1`: this causes the current thread to sleep for a given time period.

- **`global_call`**`/1`, **`global_assert`**`/1` and so on: these act in a similar way to the **`call`**`/1`, **`assert`**`/1` and so on, predicates but deal with facts on a global database which is shared between all of the threads in a P# program.

- **`wait_for`**/1: this takes as an argument a variable which has been passed in a fork, and waits for that variable to become instantiated on another thread. When this occurs the binding that occurred on the other thread is copied to the thread that called **`wait_for`**`/1`. Thus the argument of **`wait_for`**`/1` becomes instantiated. Every time the variable is instantiated on another thread a new message is enqueued on the message queue, which is stored in the global table.

- **`lock`**`/1`, **`unlock`**`/1` and **`backtrackable_lock`**`/1`: these acquire or release locks on terms.

We have also added methods that can be called from C#, which send and receive messages.

Our concurrent version of Prolog is well suited to interoperation with C#. Our approach, like DeltaProlog, retains a Prolog feel and retains Prolog as a subset. Our use of forking and event sending is similar to that of DeltaProlog, except that the sending mechanism is closer to FCP. We do not use guards in any way as this would lead to a language too far removed from Prolog. All of the new features are implemented by defining new built-in predicates. There are no syntactic changes to the language. This means that it is often straightforward for a developer to translate another dialect of Prolog to P#.

With Prolog, complex programming possibilities arise as a consequence of a simple underlying set of rules. It is interesting to observe that as a consequence of this, our simple changes to these rules have yielded many new possibilities. Examples of these are the way in which a conjunction of goals can be passed to a fork, and the fact that a message may consist of any Prolog term. The higher order and type-less nature of Prolog afforded a freedom which allowed many of the features mentioned in this thesis to be easily implemented partly in Prolog.

The concurrency features of P# could be improved by allowing the P# programmer to work with a higher level view of concurrency. We could support concurrent data-types such as semaphores and mutexes, building these on top of the concurrent primitives which we have discussed in this thesis and which are in turn built on top of C#'s concurrency primitives. This work could mirror the work that has been done on concurrency for ML [Rep99].

7.3 Idiomatic Compilation

The efficiency of P# and the readability of the code it produces can be significantly improved by compiling to more idiomatic C# with the assistance of mode and type declarations. This is because the C# compiler is designed to compile code written by human programmers. Particularly significant improvements are observed for Prolog programs that are predominantly numerical. Our compilation scheme avoids the overheads of the Prolog stacks used by the WAM in situations where they are not necessary. Our technique tends to be able to compile those predicates that are called most frequently. Thus, even when only a few of the program's predicates can be idiomatically compiled, efficiency will often still be improved.

We attempt to compile tail-recursive predicates, an example of a common Prolog idiom, into iterative loops in C#, the equivalent idiom for an imperative language. Where possible, we translate these iterative loops into **`while`** loops. We perform a liveness analysis, which prunes the code in such loops down to the basic function of the loop. In many cases these optimizations result in code far closer to that which a C# developer would have produced.

Up until the work on optimizing P# it appeared that for our purposes C# was not a significant advance on Java. Whenever we considered a C# feature that was not present in Java, we found that it would not lead to improvement. With the work on optimization, however, features of C# such as the **`goto`** statement and indexers have proved useful.

It is interesting to observe that our idiomatic compilation scheme resembles, in some ways, the operation of the WAM. Indeed, it has been suggested to the author by an anonymous referee that the WAM is such a good compilation technique for Prolog compilation, that the more the idiomatic compilation scheme is sophisticated, the closer it will get to following WAM principles. The code at the beginning of each block representing a clause resembles the `get` and `unify` WAM instructions. Also, liveness analysis is usually performed during register allocation in a WAM-based Prolog compiler.

There remain many interesting potential extensions to the idiomatic compiler. These are detailed in the following subsections, during which we draw attention to the limitations of what can and should be idiomatically compiled.

7.3.1 Idiomatic Translation of Database Primitives

We would also like to compile primitives that modify the database (**`assert`**`/1`, **`retract`**`/1` and so on) to more idiomatic C#, but there is a problem due to the fact that the database can be modified from non-idiomatic code and then called from idiomatic code or *vice versa*. However, where the programmer has specified that this does not happen and the accesses of the database are of a simple nature, we would be able to do this.

We could also investigate the translation of stateful operations, such as database access, into functional operations, such as threading an argument through the code, in order to be able to idiomatically compile code which uses such stateful operations. In some cases this would be likely to lead to inefficient code, however the use of idiomatic translation would counteract this effect to some extent.

If database operations could, somehow, be accommodated by the idiomatic translator, we would be able to idiomatically translate the idiomatic translator itself, which is written in P# Prolog. This might significantly improve the efficiency of idiomatic compilation. We would also be able to idiomatically compile the Prolog to C# translator that was derived from Prolog Café's Prolog to Java translator, as this too is written in Prolog.

7.3.2 Idiomatic Translations of Concurrent Code

We now discuss how we could idiomatically compile predicates involving the concurrent primitives: **`fork`**`/1`, **`wait_for`**`/1` and unifications involving concurrent variables.

We can only idiomatically compile such a predicate when it is semi-deterministic. This rules out predicates which produce multiple solutions and in particular those which produce or consume a queue of solutions through a concurrent variable by backtracking. Predicates which queue solutions using the `pulse/2` predicate, which makes a temporary binding in order to send a message and then undoes it, can be idiomatically compiled.

There would need to be some way to indicate to the idiomatic compiler the fact that a variable passed to a predicate is a concurrent variable. This could either take the form of a mode declaration or a type declaration.

Knowing that a variable is concurrent, we would need to detect all instances where that variable is instantiated. Wherever in the code the variable was assigned to, as part of a unification for example, a call to the C# `Send()` method of the `VariableTerm` class would have to be made. Whenever the **`wait_for`**`/1` predicate was encountered we would need to call the `Receive()` method of the `VariableTerm` representing the concurrent variable.

It would be straightforward to implement the predicates: **`stop`**`/1`, **`sleep`**`/1`, **`lock`**`/1` and **`unlock`**`/1`. Given that we know that the predicates involved are semi-deterministic, it should also be straightforward to implement the **`backtrackable_lock`**`/1` predicate.

With regard to the global table operations, similar comments apply to those given above for the database modification primitives.

Finally, we would have to implement the **`fork`**`/1` predicate. This is a case where it would be acceptable to start a new Prolog interpreter as a fork always does this. Thus, it would not matter if the thread to be forked had a non-idiomatic predicate as its entry predicate, which is likely to be the case.

7.3.3 Idiomatic Translations of Failure Driven Loops

When a Prolog program is required to read in a file, or to do some other repetitive action associated with input/output streams, failure driven loops are often used. An example taken from the implementation of P# looks similar to the following:

```
interpreter :-
    write('Prolog Interpreter'), nl,
    repeat,
        write('| ?- '),
        read_with_variables(Goal, Vs),
        '$cafe_execute'(Goal, Vs),
        Goal == end_of_file,
        !,
    nl, write(bye), nl.
```

We will, however, rarely be able to convert such a loop to an iterative C# loop, as it is likely that one of the predicates called in the loop, possibly via other predicates, cannot be compiled to idiomatic code for some reason.

7.3.4 Support for More Modes

Ideally, we would like to be able to idiomatically compile a larger class of predicates including those with modes other than input and output.

In particular we would like to be able to idiomatically compile predicates that have arguments which are uninstantiated or partially instantiated on entry into the predicate and may or may not be further instantiated by the predicate call.

One solution is to use the C# `ref` modifier, which is similar to the C# `out` modifier except that the variable passed must be initialized before the call and may or may not be changed during the call.

While we are unpacking the arguments to the predicate in order to generate the start of the block representing each clause, if we reach an uninstantiated variable, a C# variable is created whose name is based on the position of that variable in the argument.

This variable stores the uninstantiated `VariableTerm` which has been passed to the predicate.

When the final part of the block representing the clause is reached where values are copied back into the arguments, either

1. in the case that the variable is still fully uninstantiated, we return the original `VariableTerm`; or
2. in the case that the variable is now fully instantiated, we return the instantiation; or
3. in the case that the variable is now partially instantiated, we return the structure with new `VariableTerm`s in the place of the holes.

The problem with this approach is aliasing. Consider the following code:

```
mode( p( ref, ref ) ).
mode( q( ref, ref ) ).
p( X, Y ) :- X = Y, q( X, Y ).
q( [H|T], Y ) :- H = 1.
```

If we call the predicate `p/2` with two uninstantiated variables, then they become the same variable and the first becomes partially instantiated to a list beginning with the integer 1. On returning from `p/2`, we need also for the variable `Y` to become the same partially instantiated list. With the above scheme it will not be.

A solution to this problem is to extend our treatment of unification to unification of two uninstantiated variables in the way suggested earlier on page 97 and then the call to `q` becomes: `q( X, X )`. However, this would only work when the aliasing occurred during a call to an idiomatic predicate. It would still be possible for the aliasing to occur in a non-idiomatic predicate and then for aliased variables to be passed to an idiomatic predicate. One way to deal with this is to search for all aliasing during the `exec()` method of the idiomatic predicate, which could be prohibitively time consuming.

One solution to the problem of aliasing is to introduce a mode called, say, `noalias` which is used when the programmer knows that the variable declared to be `noalias` has no relation via aliases to any other variable being passed to the predicate. In fact,

there is a similar problem with the `out` mode, our `out` mode must mean ‘out with no aliasing’.

Now, as we know that the variables are unrelated, case 3 above is the correct translation.

If more modes were supported, we would encounter situations when dealing with arithmetic relational operators and unifications where we do not know the state of instantiation of some of the terms involved. In this case we need to use **`if`** statements in the C#, which first test whether the variable is instantiated and then act accordingly.

A problem arises with such **`if`** statements. We cannot test whether a variable representing a native **`int`** is instantiated or not without adding a Boolean variable to keep track of the state. As such Boolean variables would make the code very unnatural and error prone to modify, our solution to this would be to only idiomatically compile such predicates when the variable whose state of instantiation is ambiguous is declared to be a `term`.

Our experience of attempting to idiomatically translate predicates suggests that the need for such **`if`** statements would not occur often.

It is not clear, however, that idiomatically compiled code which supports more modes than just `in` and `out` would be any more efficient than the original P#/Prolog Café compilation scheme.

We would like for it to be possible for a P# programmer to specify that, in a structure occurring in the head of a clause, some of the variables are input variables and some are output variables.

7.3.5 Other Extensions to the Idiomatic Compiler

It is possible to infer more of the types than is done at present. For example, if `A` is an `int` then so is `A+1`, and given `A1` **`is`** `A + 1`, we can infer that `A1` is an `int`. Also, the programmer currently has to specify which predicates can be idiomatically compiled.

Tools could be developed that aid the programmer to modify their Prolog programs in order to make them suitable for idiomatic translation.

Work has been done on marking Mercury programs tail-recursive [ROS99]. We could adapt this work to make P# predicates tail recursive, and hence be able to translate more of them to iterative constructs in C# such as **`while`** loops.

7.4 Closing Remarks

We have discussed one way in which language interoperation can be achieved, that is, by source-to-source language translation. We have shown in our specific case, the translation of Prolog to C#, that such a translation is feasible and the resultant tool is efficient and useful. We have been able to use the tool successfully to implement the case studies detailed in this thesis. The usefulness of the tool is also evidenced by the fact that interest in the tool is not limited to its developer and his institution. A number of students and external developers from both academia and industry have contacted the author, expressing an interest in using P#, asking questions about P# and thanking the author for developing P#.

The reader may wish to obtain our tool, which is available from

`http://www.lfcs.ed.ac.uk/psharp`

Bibliography

[AK91] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT press, 1991.

[Alb] B. Albahrari. A comparative overview of C#. http://genamics.com/developer/csharp_comparative.htm.

[App98] Andrew W. Appel. *Modern compiler implementation in C*. Cambridge University Press, 1998.

[BC02] Roberto Bagnara and Manuel Carro. Foreign language interfaces for Prolog: A terse survey, 2002. The Association for Logic Programming Newsletter, vol 15. Association for Logic Programming.

[Bec] R. Becket. Mercury tutorial. http://www.cs.mu.oz.au/research/mercury/information/documentation.html.

[Bin] BinProlog home page. http://www.binnetcorp.com/BinProlog/.

[BK99] Nick Benton and Andrew Kennedy. Interlanguage working without tears: Blending SML with Java. In *International Conference on Functional Programming*, pages 126–137, 1999.

[BKR99] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 129–140, 1999.

[BKR04] Nick Benton, Andrew Kennedy, and Claudio Russo. Adventures in in-

teroperability: The SML.NET experience. In *Proceedings of the 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP).*, 2004.

[Box98] D. Box. *Essential COM*. Addison Wesley, 1998.

[BPr] BProlog home page. http://www.cad.mse.kyutech.ac.jp/people/zhou/bprolog.html.

[BT97] M. Banbara and N. Tamura. Java implementation of a Linear Logic Programming language. In *Proceedings of the 10th Exhibition and Symposium on Industrial Applications of Prolog*, pages 56–63, 1997.

[BT98] M. Banbara and N. Tamura. Compiling resources in a Linear Logic Programming language. In *Proceedings of Post-JICSLP'98 Workshop on Parallelism and Implementation Technology for Logic Programming Languages*, 1998.

[BT99] Mutsunori Banbara and Naoyuki Tamura. Translating a Linear Logic Programming language into Java. *Electronic Notes in Theoretical Computer Science*, 30(3), 1999.

[Cam84] J. A. Campell, editor. *Implementations of PROLOG*. Ellis Horwood, 1984.

[CD95] Philippe Codognet and Daniel Diaz. WAMCC: Compiling Prolog to C. In *International Conference on Logic Programming*, pages 317–331. MIT Press, 1995.

[CH99] Manuel Carro and Manuel V. Hermenegildo. Concurrency in Prolog using threads and a shared database. In *International Conference on Logic Programming*, pages 320–334, 1999.

[CIA] CIAO home page. http://clip.dia.fi.upm.es/Software/Ciao/.

[Cia92] Paolo Ciancarini. Parallel programming with logic languages: A survey. *Computer Languages*, 17(4):213–239, 1992.

[CM94] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer, 4th edition, 1994.

[Coo02] Jonathan J. Cook. Reverse execution of Java bytecode. *The Computer Journal*, 45(6):608–619, 2002.

[Coo04a] Jonathan J. Cook. P#: A concurrent Prolog for the .NET Framework. *Software: Practice and Experience*, 34(9):815–845, 2004.

[Coo04b] Jonathan J. Cook. Optimizing P#: Translating Prolog to more Idiomatic C#. In *Proceedings of CICLOPS 2004*, 2004.

[Cor01] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, 2001.

[CSH] Thomas Conway, Zoltan Somogyi, and Fergus Henderson. *The Prolog to Mercury Transition Guide*. http://www.cs.mu.oz.au/research/mercury/information/documentation.html.

[Cw] Cω home page. http://research.microsoft.com/Comega/.

[Dij68] E. W. Dijkstra. Goto considered harmful. *Communications of the ACM*, 11(3):147–8, 1968.

[dlBDMS02] M. García de la Banda, B. Demoen, K. Marriott, and P. Stuckey. To the gates of HAL: a HAL tutorial. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming, number 2441 in LNCS*, pages 47–66. Springer-Verlag, 2002.

[DM94] Bart Demoen and Greet Maris. A comparison of some schemes for translating logic to C. In *ICLP Workshop: Parallel and Data Parallel Execution of Logic Programs*, pages 79–91, 1994. Published as UPMAIL Technical Report No. 78, Upsala Univesity, Computing Science Department.

[Dyb96] R. Kent Dybvig. *The Scheme programming language*. Prentice Hall, Inc., 1996.

[EC98] Jesper Eskilson and Mats Carlsson. SICStus MT — A multithreaded execution environment for SICStus Prolog. *Lecture Notes in Computer Science*, 1490:36–53, 1998.

[FLMJ99] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. Calling hell from heaven and heaven from hell. In *International Conference on Functional Programming*, pages 114–125, 1999.

[FSW98] Juliana Freire, Terrance Swift, and David S. Warren. Beyond depth-first strategies: Improving tabled logic programs through alternative scheduling. *Journal of Functional and Logic Programming*, 1998(3), April 1998.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir95] J.-Y. Girard. Linear logic: Its syntax and semantics. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic (Proc. of the Workshop on Linear Logic, Cornell University, June 1993)*, number 222. Cambridge University Press, 1995.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Brancha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

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

[Gor98] Rob Gordon. *Essential JNI Java Native Interface*. Prentice Hall PTR, 1998.

[GPA+01] Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of Prolog programs: a survey. *Programming Languages and Systems*, 23(4):472–602, 2001.

[Gre87] S. Gregory. *Parallel Logic Programming in PARLOG, The Language and Its Implementation*. Addison Wesley, 1987.

[HAL] The HAL home page. http://www.csse.monash.edu.au/~mbanda/hal/.

[Han92] Werner Hans. A complete indexing scheme for WAM-based abstract machines. In *PLILP 1992*, pages 232–244, 1992.

[HCS+] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, Chris Speirs, Tyson Dowd, and Ralph Becket. *The Mercury Language Reference Manual*. http://www.cs.mu.oz.au/research/mercury/information/documentation.html.

[HG91] M. Hermenegildo and K. Greene. The &-Prolog system: Exploiting independent and-parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HGC95] Manuel V. Hermenegildo, Daniel Cabeza Gras, and Manuel Carro. Using attributed variables in the implementation of concurrent and parallel logic programming systems. In *International Conference on Logic Programming*, pages 631–645, 1995.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HS02] Fergus Henderson and Zoltan Somogyi. Compiling mercury to high-level C code. In *Computational Complexity*, pages 197–212, 2002.

[HWTK98] Joshua S. Hodas, K. M. Watkins, Naoyuki Tamura, and Kyoung-Sun Kang. Efficient implementation of a Linear Logic Programming language. In *IJCSLP*, pages 145–159, 1998.

[Jin] Jinni home page. http://www.binnetcorp.com/Jinni/.

[jPr] jProlog home page. http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/.

[Jyt] Jython home page. http://www.jython.org/.

[Kaw] Kawa Scheme home page. www.gnu.org/software/kawa/.

[Ken96] Andrew Kennedy. Drawing trees. *Journal of Functional Programming*, 6(3):527–534, 1996.

[KN90] A. Krall and U. Neumerkel. The Vienna Abstract Machine. In *Proceedings of PLILP 1990*, 1990.

[Knu74] Donald E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6(4):261–301, 1974.

[LBD+88] Ewing Lusk, Ralph Butler, Terence Disz, Robert Olson, Ross Overbeek, Rick Stevens, D.H.D Warren, Alan Calderwood, Peter Szerdi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumil Hausman. The Aurora or-parallel Prolog system. In *Proceedings of the 3rd International Conference on Fifth Generation Computer Systems*, pages 819–830. Addison-Wesley, 1988.

[Li96] X. Li. Program sharing: A new implementation approach for Prolog. In *Proceedings of the 8th International Symposium, PLILP '96, LNCS 1140*, pages 259–273, 1996.

[Lib01] Jesse Liberty. *Programming C#*. O'Reilly, 2001.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley Longman Inc., 2nd edition, 1999.

[Mer] The Mercury home page. http://www.cs.mu.oz.au/research/mercury/.

[Mie84] C. Mierowsky. Design and implementation of Flat Concurrent Prolog. Technical Report TR CS84-21, Weizmann Institute, 1984.

[MIN] MINERVA home page. http://www.ifcomputer.com/MINERVA/.

[MLj] MLj home page. http://www.lfcs.ed.ac.uk/mlj.

[MM] Erik Meijer and Jim Miller. Technical overview of the common lan-

guage runtime (or why the JVM is not my favorite execution environment). docs.msdnaa.net/ark/Webfiles/WhitePapers/CLR.pdf.

[Mon] Mondrian home page. http://www.mondrian-script.org/.

[.NE] .NET languages. http://www.gotdotnet.com/team/lang/.

[NET] The Microsoft developer .NET home page. http://msdn.microsoft.com/net.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of ECOOP 2003*, 2003.

[Oct] Octopus .NET translator home page. http://www.remotesoft.com/octopus/.

[P#] P# manual. http://www.lfcs.ed.ac.uk/psharp.

[PC] Prolog Café home page. http://pascal.cs.kobe-u.ac.jp/~banbara/PrologCafe/index-jp.html.

[Pla02] D. S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 2nd edition, 2002.

[PN84] L. M. Pereira and R. Nasr. Delta-Prolog: A distributed logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Languages*, pages 283–291, 1984.

[Pop97] Alan Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison Wesley, 1997.

[PTB+97] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java for applications—a way ahead of time (WAT) compiler. Technical report, Dept. of Computer Science, University of Arizona, Tucson., 1997.

[Pus96] Cornelia Pusch. Verification of compiler correctness for the WAM. In J. Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th*

International Conference on Theorem Proving in Higher Order Logics (TPHOL'96), Turku, Finland, 1996. Springer-Verlag LNCS 1125.

[Rep99] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[RMH90] Mads Tofte Robin Miler and Robert W. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[ROS99] Peter Ross, David Overton, and Zoltan Somogyi. Making mercury programs tail recursive. In *Logic Program Synthesis and Transformation*, pages 196–215, 1999.

[Sce01] D. Sceppa. *Microsoft ADO.NET*. Microsoft Press International, 2001.

[Sha87] E. Shapiro. *Concurrent Prolog—Collected Papers*. MIT Press, 1987.

[SIC] SICStus Prolog home page. http://www.sics.se/sicstus/.

[SML] SML.NET home page. http://www.cl.cam.ac.uk/Research/TSG/SMLNET/.

[Str00] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 2000.

[TB90] P. Tarau and M. Boyer. Elementary logic programs. In P. Deransart and J. Maluszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, pages 159–173. Springer, LNCS 456, 1990.

[Tic95] E. Tick. The deevolution of concurrent logic programming languages. *Journal of Logic Programming*, 23(2):89–123, 1995.

[TK96] N. Tamura and Y. Kaneda. Extension of WAM for a Linear Logic Programming language. In *Proceedings of the Second Fuji International Workshop on Functional and Logic Programming*, 1996.

[TK97] Naoyuki Tamura and Yukio Kaneda. A compiler system of a Linear Logic Programming language. In *Proceedings of the IASTED Interna-*

tional Conference on Artificial Intelligence and Soft Computing, Banff, Canada, pages 180–183, 1997.

[TL01] Thuan Thai and Hoang Q. Lam. *.NET Framework Essentials*. O'Reilly, 2001.

[TLA92] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

[TV00] Andrey A. Terekhov and Chris Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, 2000.

[War83] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, CA., 1983.

[War88] D. H. D. Warren. Implementation of Prolog, 1988. Tutorial No. 3, 5th International Conference and Symposium on Logic Programming, Seattle, WA.