P#: A concurrent Prolog for the .NET Framework

SP&E

Jonathan J Cook

Laboratory for Foundations of Computer Science, University of Edinburgh, EH9 3JZ, UK.
`Jon.Cook@ed.ac.uk`

SUMMARY

We discuss P#, our implementation of a tool that 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. We add language constructs on the Prolog side that allow concurrent Prolog code to be written. A primitive predicate is provided that 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 discuss two software engineering tools implemented using P#.

KEY WORDS: Concurrency, Prolog, C#, .NET

INTRODUCTION

Microsoft's .NET platform [1] offers an unparallelled opportunity to build systems based on a number of interoperating languages. A list of the languages currently implemented for the .NET platform can be found in Reference [2]. Logic programming is currently underrepresented, 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.

We would like such a compiler to generate code that executes efficiently and uses C#'s language constructs idiomatically. We would also like our tool to exploit the rich features that C#, being a modern programming language, possesses. 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 toward it.

There already exist translators that translate from Prolog to C, see Reference [3], for example GNU-Prolog [4, 5], 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 [6, 7, 8, 9, 10], BinProlog [11], B-Prolog [12] and the commercial product MINERVA [13]. The Prolog in 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 [14]. 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# is a relatively new object-oriented programming language that has drawn on the languages C++ [15] and Java [16, 17]. We now summarize the essential points of C#, see References [18] and [19] for more details. Like Java, C# is compiled into an intermediate language. C# shares with Java some features not found in C++, such as garbage collection, reflection, thread support, static inner classes, and the ability to add `finally` clauses to `try` blocks. C# supports Java-style interfaces and abstract methods, and like Java does not allow classes to be defined by multiple inheritance. The support for concurrency is similar to that of Java, being based on locked regions and monitors. C# also shares features with C++ that are 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++. Event handling is implemented using delegates, a construct described in Reference [18] as a "type-safe object-oriented function pointer, which is able to hold multiple methods". C# has extensive library support for XML and regular expressions. Like Java, it also has good library support for networking.

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 hope, in this way, to find a compromise between speed and readability that produces reasonably efficient, well-structured code.

.NET [1] is a framework developed by Microsoft to support the interaction of web services and clients via XML, with a view to enabling these services to be called across languages and platforms. C# and .NET are related, each being to some extent designed to work well with the other. In order to facilitate the writing of web services, a framework has been developed that 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. Thus, C# is an important language.

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, MSIL. 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 MSIL via C.

A functional logic language, called Mercury [20, 21, 22], is available for use with .NET. Mercury, despite being reminiscent of Prolog, is a fully declarative language. Thus, the developers of Mercury did not have to deal with 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 that 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 Reference [23].

Another way in which Prolog could be used within the .NET framework is by translating Prolog directly to MSIL. 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 MSIL code than C#. This provides a strong incentive for generating code that 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# [24] 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 [25] and FCP [26]. 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 wanted no non-concurrent P# Prolog program to be broken, provided that it happened not 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.

We discuss two case studies, both of which are 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.

OBTAINING A TRANSLATOR TO C# FROM PROLOG CAFE

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 paper was carried out, version 0.6.1 has been released, which includes support for multithreading of a different nature to ours. Prolog Café translates a linear logic extension of Prolog, called LLP, via a linear logic extension of the WAM, namely the LLPAM [27], to Java. The linear logic features that P# has inherited from Prolog Café are detailed with examples in the paper, Reference [28]. For more information on linear logic, see References [29, 30].

For more information on the WAM (Warren Abstract Machine), a standard compilation strategy for Prolog, see References [31, 32, 33]. For more information on the Prolog language, see Reference [34].

Prolog Café is an extension of jProlog [35], which uses a continuation passing style of compilation referred to as binarization and described in Reference [36].

Prolog Café consists of a run-time system written in Java, which essentially 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.

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
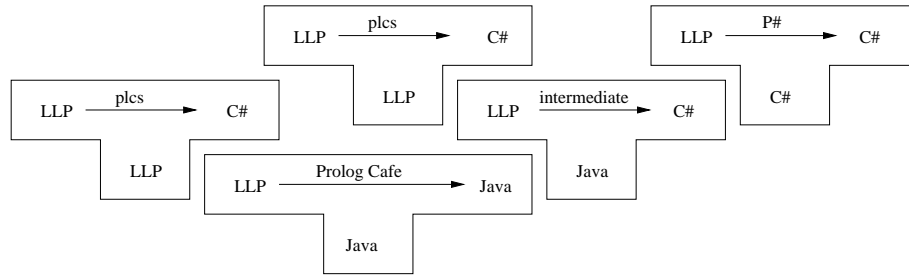
Figure 1: Obtaining a Bootstrapped Translator to C#

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 [37] 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 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 is 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.

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`.

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 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 2. Usually after the first two reflections have occurred the predicates that are to be called can be determined statically, thus there is very 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.
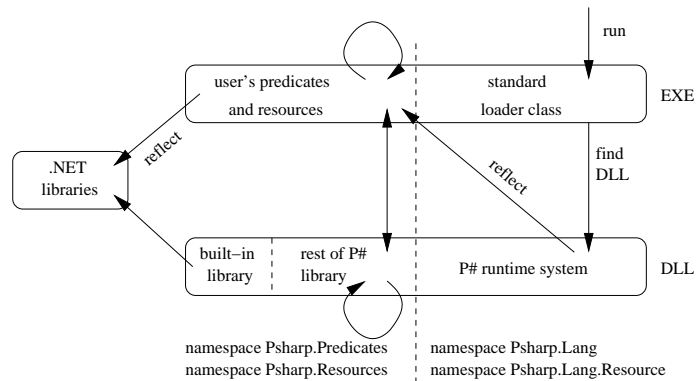
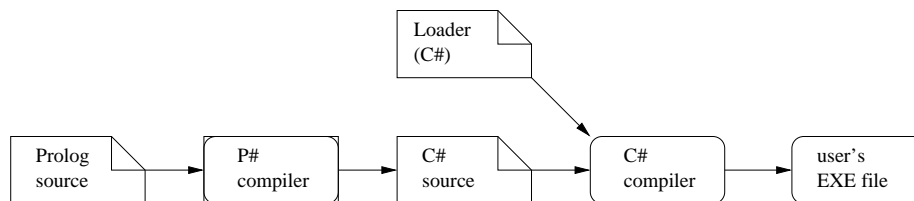**Figure 2**: Separation into a DLL and an EXE

Figure 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 mentioned above. This loader runs the DLL directly in 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 shows how the process by which a user is able to generate a stand-alone C# application from a Prolog/LLP source file.

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 to execute next, 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 very 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 initialised to contain just a call to `a`. The timings were taken by using the C# `System.DateTime.Now.Ticks` property.

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 because the C# compiler is good at optimising code that makes heavy use of objects, but is unable to optimise so effectively

---

Table I. 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

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 MSIL.

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 I 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, 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 be called on an object as follows, for example:

```
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 `plcomp/1` predicate that compiles a given Prolog file into memory.

Example code generated by P#

Figure 4 shows a simple Prolog predicate, chosen to illustrate the way in which both conjunction and disjunction are compiled, and Figure 5 shows part of the C# class into which it is compiled. Figure 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 it then calls `A_1_sub_1`, 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`s to be 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.

CONCURRENT P#: FEATURES AND EXAMPLES

In order to be able to create new threads, we add a primitive called `fork/1`. The `fork` 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` 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.

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

Figure 4: A simple Prolog predicate

```
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 5: C# code

```
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 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 6: WAM code produced by GNU Prolog

Communication between threads

The `wait_for` 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`. `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' ).                           guess( Z ) :-
alpha( 'b' ).                             alpha( X ),
alpha( 'c' ).                             fork( correct( X, Z ) ),
alpha( 'd' ).                             fail.
                                        guess( Z ) :-
correct( X, Y ) :-                        wait_for( Z ).
  \+ var( X ), % prevent cheating
  X = 'c',
  Y = X.
```

The `guess/1` predicate knows that the correct answer is 'a', 'b', 'c' or 'd'. However it can only find out which 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` 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".

Queueing 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` to take each binding.

The `wait_for` predicate can be called repeatedly by using the usual `repeat/0` predicate, however `wait_for` 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 `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` 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` on the remaining thread will fail. However, care must be exercised. If we had defined `main` 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 the interpreter thread.

IMPLEMENTATION OF CONCURRENCY

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, initialised 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 main `Predicate` class there is no problem, as each thread will create a separate object. The continuation field in a subclass of this is initialised 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).

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 FCO, allows the first body goal to call the predicate without consuming an unnecessary choice point. FCO is discussed in Reference [38]. 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 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 hashtable 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.

Forking threads and the global table

When a fork is called on a structure a deep clone of that structure is created. In 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.

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.

   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 VariableTerms 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. Thus, a call to global_assert 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, 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 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 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. $run_on_global_engine can only be executed by one thread at a time.

The wait_for predicate

When a concurrent variable is bound to a term, this is detected and a call to global_assertz is made, asserting the fact that the relevant variable has been bound to the relevant term. We use global_assertz 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 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 predicate is built on top of global_call. 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, 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.

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 this 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` 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` 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` 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.

Interoperation with C#

The locks dealt with by `lock` and `unlock` are C# locks on C# objects. Similarly `fork` initializes and starts a new C# thread, unification calls the `PulseAll()` method on an object and `wait_for` 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 with 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.

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. This time 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` 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.

SEMANTICS OF CONCURRENCY

Figure 7 shows the control flow logic of the unification and `wait_for` 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` predicate invokes the `global_call` 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` predicate succeeds and up until there is backtracking back through the call to `wait_for`. When `wait_for` is re-executed on backtracking the lock on the `ConcurrentVariable` object is still held, and is held until `wait_for` 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 Reference [39]. Thus on backtracking the side effect is performed again. The only exception to this is the `backtrackable_lock` predicate, which releases the lock on backtracking. It is the `backtrackable_lock` predicate that allows the lock to be held throughout the execution of deeper code.

If the call to `global_call` fails, then we wait until a message appears, or in the case that all of the threads are waiting, the call to `wait_for` 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` for that variable should fail.

EXAMPLE: 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 the 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. Mono is an "open source implementation of the .NET Development Framework", which runs on Windows and several implementations 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.
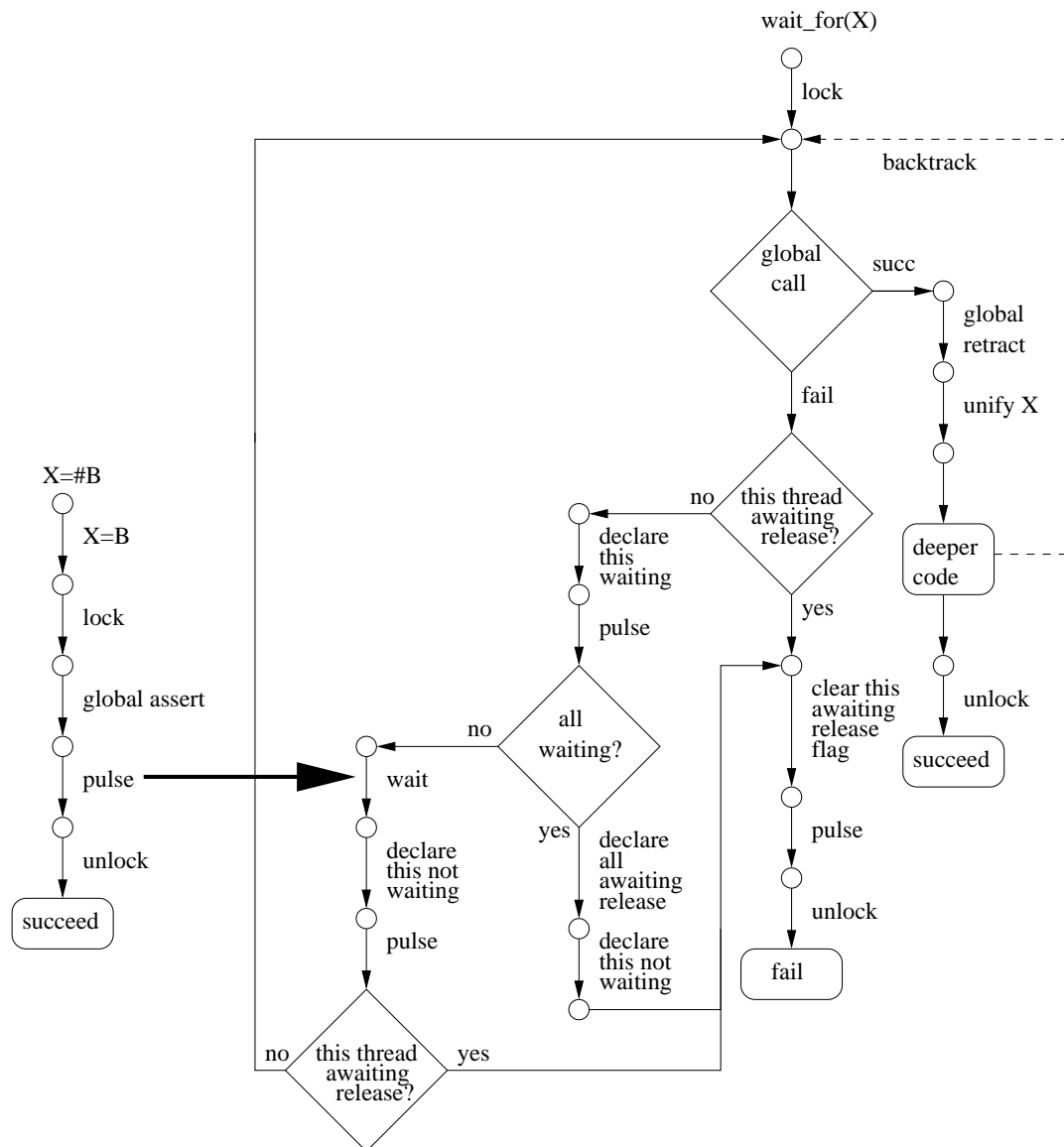
Figure 7: Control flow logic of unification and wait_for

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 to maintain 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` predicate.

EXISTING CONCURRENT PROLOG DERIVATIVES

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, Reference [40], 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. The concurrent logic languages Parlog, Guarded Horn Clauses and Concurrent Prolog are discussed in Part I of Reference [41].

DeltaProlog [25] is an extension of Prolog based on CSP [42]. 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 [39] is an multithreaded extension of SICStus Prolog. SICStus MT has a predicate that spawns a thread, named `spawn/2`, which is similar to our `fork` predicate. Messages can be sent from one thread to another thread, by specifying in the call to the `send/2` predicate the destination 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 are the `self/1` predicate that returns an identifier for the currently executing thread, a predicate that waits for a period of time, similar to our `sleep` predicate, and finally a predicate that kills a thread, similar to our `stop` predicate.

Jinni also has support for concurrency, based on Linda blackboards. Terms can be read from and written to the shared blackboard. Our global table is similar to this, however blackboards also provide a synchronization facility as attempting to retrieve a term can block if there is no term to retrieve.

Languages that have implicit parallelism can be divided into or-parallel languages, such as Aurora, independent and-parallel languages such as &-Prolog and dependent and-parallel parallel languages such as Andorra, the latter being related to committed choice languages. The paper, Reference [43], is a recent survey of languages with implicit parallelism.

Aurora [44] is an or-parallel Prolog, well suited to the parallel programming of parallel processors. This means that a predicate that is marked as parallel can have its *clauses* evaluated in parallel. Parallelism in Aurora is implicit.

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 aims of Aurora are very different from those of P#. We were interested instead in building a concurrent Prolog based on C#'s concurrency primitives. Or-parallelism does not seem to make sense in this setting. We use a construct similar to the asynchronous `assert`.

The parallel Prolog language &-Prolog automatically parallelises 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, which 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) [26] 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 "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 Reference [40]:

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

FCP has, from our point of view, a serious disadvantage. This disadvantage is that it is very unlike Prolog. Indeed it is non-trivial even to simulate Prolog programs in FCP. It is for this reason that while we use variables to communicate we do not make use of guards in P#.

Parlog [45] is similar to FCP. Instead of read-only annotations, each predicate is given a mode declaration, which indicates which variables are input variables and which are output. As with FCP, communication is via unification of shared variables.

FCP, Parlog and FGHC (Flat Guarded Horn Clauses) are examples of committed choice languages [46]. 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" nondeterminism.

Our `wait_for` 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.

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 [47] also has support for concurrency consisting of a extensive choice of predicates for spawning new threads and for occasioning 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 was as a blackboard, an idea proposed in Reference [48]. P# also uses a Prolog database in this way, namely the global table, as message sending by unification and the `wait_for` predicate use the global table to pass messages. The paper, Reference [49], proposes a `wait` predicate that waits until a variable is bound, together with a range of forking primitives.

CASE STUDIES

In this section, we consider two case studies, which illustrate the use of P#: an object-oriented assistant and a class hierarchy viewer.

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 as well as C# namespaces.

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, for efficiency reasons, 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` 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 Internet Explorer is spawned, which displays the documentation for that method or field. A screen-shot of the object-oriented assistant is included as Figure 8.

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 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 grabbed 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 optimisation 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 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 when a query is not being executed is not wasted.

Class hierarchy viewer

We discuss an implementation of a tool, which 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
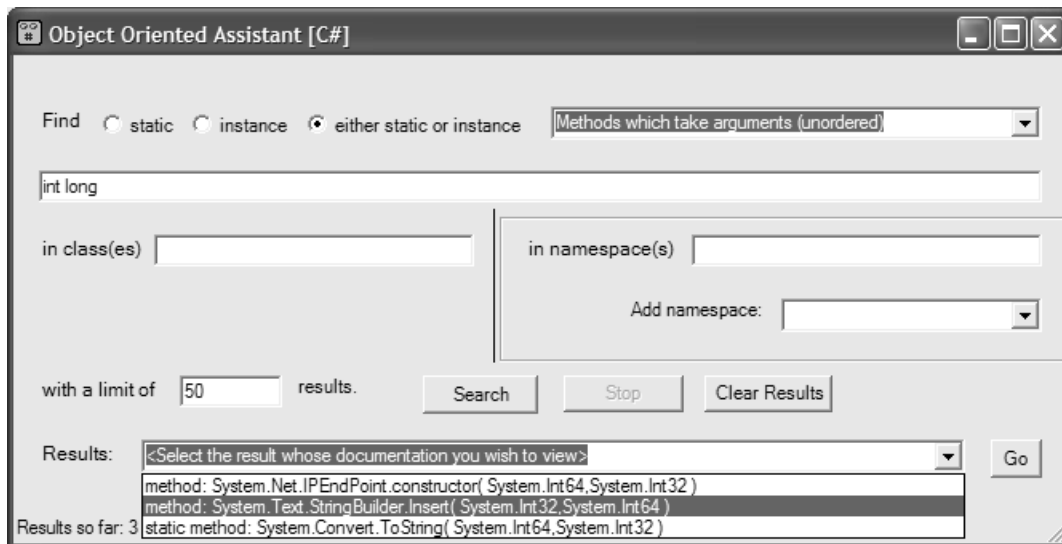
Figure 8: Screen-shot of Object-Oriented Assistant

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 [50] gives an SML program for drawing trees in an aesthetically pleasing manner. The MLj homepage [51] includes full source code and an online 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.
```

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.

PERFORMANCE

We benchmarked concurrent P# against the Java-based Prolog systems Jinni 8.48 [52], MINERVA 2.4 and Prolog Café 0.4.4, and against the non-concurrent version of P#, as shown in Table II.

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 paper. 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` 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.

Prepared using **speauth.cls**

Table II. Comparison of P# with other tools

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# normalised to 1.00		1.00	1.36	1.12	0.55	1.13

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 runtime system, and library is roughly 3 Megabytes in size. The corresponding file for P#, the DLL, is roughly 1 Megabyte in size.

FUTURE WORK

Generating more idiomatic C# and mode inference

Both the runtime system, which is written in C#, and the translator, which is written in Prolog, of P# contain code that translates Prolog names to C# names. The same algorithm

is used: but tail recursive in the Prolog and iterative in the C#. We wrote the Prolog version first, and manually translated it to idiomatic C#. We felt that in this case the translation to an iterative version was fairly mechanical. In future work, we hope to modify the translator so that it can detect instances where such a translation can be used to automatically translate Prolog to more idiomatic C#. As with similar projects, such as HAL [53] and Mercury, we may support this by allowing the Prolog code to be annotated with mode declarations. In this case the renaming predicate would have one input and one output parameter.

The current compilation scheme leaves open the possibility of compiling a predicate, and all predicates deeper than it in the tree, into more idiomatic C#. Thus, if we could detect instances where this improvement would be possible, it may be the case that much more efficient C# could be generated.

In particular tail recursive predicates that involve no cuts could be compiled into `while` loops. For example consider the usual Prolog code for finding the length of a list:

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


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

This Prolog code could be compiled into:

```
a = 0;
while( !list.isEmpty( ) ) {
    list = list.tail( );
    a++;
}
return a;
```

A number of stages would be involved in such a compilation, for example mode inference to determine that `Length` is an output parameter of `len/2` and a liveness analysis to determine that `a` should be incremented by one in each iteration of the loop.

In fact much tail recursive code can be characterized as conforming to the following general pattern:

```
p( ..., Z, Z ). % base 1
p( ..., Z, Z ). % base 2
...             % base i


p( ..., A, Z ) :- ..., p( ..., A1, Z ). % step 1
p( ..., A, Z ) :- ..., p( ..., A1, Z ). % step 2
...                                      % step j
```

It would be possible, though maybe quite involved, to detect code that looks like this and then to translate it into iterative code. However, it is not clear that the code would run

significantly faster since much of the translated code is the same as it was before. In the list length example above, we are still calling the method that returns a list's tail. Nevertheless making the generated code more idiomatic should ensure that we are working with the C# compiler's optimizer rather than against it.

Related projects, such as Mercury [22] and HAL [53], support mode declarations. A mode declaration provides extra information to the compiler regarding which arguments of a predicate are input arguments and which are output arguments. Providing such information can be optional, only being used to provide more efficient compiled code when some mode declarations have been provided.

Notwithstanding backtracking, as the program runs forwards variables change from being uninstantiated to instantiated. Together with any mode data, we can then infer some of the modes that have not been provided.

Even if only the external interfaces to the Prolog program are given mode declarations, a large number of modes could be inferred.

Initial experiments suggest that compiling to more idiomatic code could speed up the execution of tail recursive predicates by an order or magnitude. These experiments involved manually modifying the C# generated by P# to more idiomatic code that a compiler could conceivably generate.

Other possible extensions

One of the main innovations of GNU Prolog was to exploit a feature of a dialect of C to allow very fast indirect branching. Indirect branching refers to those WAM instructions that branch to a location stored in a register. Essentially the idea was to jump into functions, thereby avoiding the overhead of function calls. Delegates and `goto`s may be useful in obtaining fast direct and indirect branching, indeed invoking a delegate closely models an indirect branch.

It should be possible to eliminate all object creation of `Predicate`s, but not `Term`s, by storing pointers to static `exec()` methods in delegates. Initial experiments suggest, however, that this change would not improve efficiency as it seems to defeat optimizations that are performed by the C# compiler or JIT compiler on code that involves frequent object creation. This observation strengthens our view that attempting to produce code "as a human would write it" could bring efficiency benefits.

The `goto` construct in C# is only able to jump to a label in the same block and certainly could not be used for jumping into the middle of methods. It may be useful, however, to have the ability to jump within methods generated for clauses. Considerable importance is placed on the issue of jumps in the existing literature on Prolog implementation.

We may be able to exploit operator overloading to obtain more readable code. For example, the = sign might be used for unification and the == sign might be used for equality of terms, as in Prolog.

We also intend to add a module system, probably based on that of SICStus Prolog. This addition will allow us to deal with those predicates that are internal to P# in a more natural way. Modules would be translated into C# namespaces.

In the object-oriented assistant case study above, we manually translated ML to Prolog. We intend to investigate the automation of such translations.

CONCLUSION

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.

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

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

We have implemented a concurrent version of Prolog that 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.

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 that allowed many of the features mentioned in this paper to be easily implemented partly in Prolog.

We have given two examples where 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.

REFERENCES

1. 'The Microsoft developer .NET home page'. `http://msdn.microsoft.com/net`.
2. '.NET languages'. `http://www.jasonbock.net/dotnetlanguages.html`.
3. B. Demoen and G. Maris. 'A comparison of some schemes for translating logic to C'. In 'ICLP Workshop: Parallel and Data Parallel Execution of Logic Programs', 79–91 (1994).
4. 'GNU Prolog home page'. `http://pauillac.inria.fr/~diaz/gnu-prolog/`.
5. P. Codognet and D. Diaz. 'WAMCC: Compiling Prolog to C'. In 'International Conference on Logic Programming', 317–331 (1995).

6. 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', 56–63 (1997).

7. 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).

8. M. Banbara and N. Tamura. 'Translating a Linear Logic Programming language into Java'. *Electronic Notes in Theoretical Computer Science*, **30** (1999).

9. J. S. Hodas, K. M. Watkins, N. Tamura and K.-S. Kang. 'Efficient implementation of a Linear Logic Programming language'. In 'IJCSLP', 145–159 (1998).

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

11. 'BinProlog home page'. http://www.binnetcorp.com/BinProlog/.

12. 'BProlog home page'. http://www.cad.mse.kyutech.ac.jp/people/zhou/bprolog.html.

13. 'MINERVA home page'. http://www.ifcomputer.com/MINERVA/.

14. R. Bagnara and M. Carro. 'Foreign language interfaces for Prolog: A terse survey'. Newsletter of the Association for Logic Programming.

15. B. Stroustrup. *The C++ Programming Language*. Addison Wesley (2000).

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

17. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley Longman Inc., 2nd edition (1999).

18. B. Albahrari. 'A comparative overview of C#'. http://genamics.com/developer/csharp_comparative.htm.

19. J. Liberty. *Programming C#*. O'Reilly (2001).

20. R. Becket. 'Mercury tutorial'. http://www.cs.mu.oz.au/research/mercury/information/documentation.html.

21. F. Henderson et al. *The Mercury Language Specification*. http://www.cs.mu.oz.au/research/mercury/information/documentation.html.

22. 'The Mercury home page'. http://www.cs.mu.oz.au/research/mercury/.

23. T. Conway et al. *The Prolog to Mercury Transition Guide*. http://www.cs.mu.oz.au/research/mercury/information/documentation.html.

24. 'P# home page'. http://www.lfcs.ed.ac.uk/psharp.

25. L. M. Pereira et al. 'Delta-Prolog: A distributed logic programming language'. In 'Intl Conf 5th Gen Comp Sys', (1984).

26. C. Mierowsky. 'Design and implementation of Flat Concurrent Prolog'. Technical Report TR CS84-21, Weizmann Institute (1984).

27. N. Tamura and Y. Kaneda. 'Extension of WAM for a Linear Logic Programming language' (1996).

28. N. Tamura and Y. Kaneda. 'A compiler system of a Linear Logic Programming language'. In 'Proceedings of the IASTED International Conference on Artificial Intelligence and Soft Computing, Banff, Canada', 180–183 (1997).

29. J.-Y. Girard. 'Linear logic'. *Theoretical Computer Science*, **50**, 1–102 (1987).

30. 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).

31. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT press (1999). Out of print, available from http://www.isg.sfu.ca/~hak.

32. D. H. D. Warren. 'An abstract Prolog instruction set'. Technical Report 309, SRI International, Menlo Park, CA. (1983).

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

34. W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer, 4th edition (1994).

35. 'jprolog home page'. http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/.

36. P. Tarau and M. Boyer. 'Elementary logic programs'. In P. Deransart and J. Maluszyński, editors, 'Proceedings of Programming Language Implementation and Logic Programming', 159–173. Springer, LNCS 456 (1990).

37. 'SICStus Prolog home page'. http://www.sics.se/sicstus/.

38. J. Freire, T. Swift and D. S. Warren. 'Beyond depth-first strategies: Improving tabled logic programs through alternative scheduling'. *Journal of Functional and Logic Programming*, **1998** (1998).

39. J. Eskilson and M. Carlsson. 'SICStus MT — A multithreaded execution environment for SICStus Prolog'. *Lecture Notes in Computer Science*, **1490**, 36–53 (1998).
40. P. Ciancarini. 'Parallel programming with logic languages: A survey'. *Computer Languages*, **17**, 213–239 (1992).
41. E. Shapiro. *Concurrent Prolog—Collected Papers*. MIT Press (1987).
42. A. R. Hoare. *Communicating Sequential Processes*. P-H (1985).
43. G. Gupta, E. Pontelli, K. A. M. Ali, M. Carlsson and M. V. Hermenegildo. 'Parallel execution of Prolog programs: a survey'. *Programming Languages and Systems*, **23**, 472–602 (2001).
44. E. Lusk et al. 'The Aurora or-parallel Prolog system'. In 'Proceedings of the 3rd Internation Conference on Fifth Generation Computer Systems', 819–830. Addison-Wesley (1988).
45. S. Gregory. *Parallel Logic Programming in PARLOG, The Language and Its Implementation*. Addison Wesley (1987).
46. E. Tick. 'The deevolution of concurrent logic programming languages'. *Journal of Logic Programming*, **23**, 89–123 (1995).
47. 'CIAO home page'. `http://clip.dia.fi.upm.es/Software/Ciao/`.
48. M. Carro and M. V. Hermenegildo. 'Concurrency in Prolog using threads and a shared database'. In 'International Conference on Logic Programming', 320–334 (1999).
49. M. V. Hermenegildo, D. C. Gras and M. Carro. 'Using attributed variables in the implementation of concurrent and parallel logic programming systems'. In 'International Conference on Logic Programming', 631–645 (1995).
50. A. Kennedy. 'Drawing trees'. *Journal of Functional Programming*, **6**, 527–534 (1996).
51. 'MLj home page'. `http://www.lfcs.ed.ac.uk/mlj`.
52. 'Jinni home page'. `http://www.binnetcorp.com/Jinni/`.
53. 'The HAL home page'. `http://www.csse.monash.edu.au/~mbanda/hal/`.