Second Year Progress Report

Language Interoperability and Logic Programming Languages

Jonathan Cook, LFCS, July 2002

Abstract

The LLP/Prolog to Java translator Prolog Café was altered to produce C# instead. New built-in predicates were added to enable the user to exploit the concurrency support in C# from the Prolog side. I now intend to work on optimising the translated code by detecting common programming idioms and by allowing the programmer to add mode annotations to their code. I also intend to add a module system and support for floating point arithmetic.

1 Introduction

.NET [16] is a framework developed by Microsoft intended to support the interaction of web services and clients via XML, with a view to enabling these services to be called across languages and platforms. C# [1, 13] is Microsoft's flagship language and is related to .NET, each being to some extent designed to work well with the other.

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

Translating Prolog [5] to C# source code provides one way of using Prolog within the .NET Framework. By translating to MSIL via C# we can exploit the C# compiler's ability to produce well optimised MSIL. It is also possible to exploit language features of C#, in particular its rich graphical, networking and other libraries, by building equivalent features in Prolog.

This report begins by describing how Prolog Café [2, 3, 4, 12, 17, 18, 19] was altered to produce C# rather than Java. This, our new Prolog implementation, is called P#. Then, we discuss the addition of concurrency support to P#, and finally we explore possible future directions.

2 Work Completed

2.1 Modifying Prolog Café to Produce C#

The LLP/Prolog to Java translator, Prolog Café, was altered to produce C# instead, see [7].

Prolog Café consists of a runtime system written in Java, and a Prolog to Java translator written in Prolog. This Prolog code is translated to Java and coupled with the runtime system to form Prolog Café. Thus, the compiler is bootstrapped.

First, the translator was modified to produce C# instead of Java. This was straightforward and because of the close similarly between Java and C# only required changes to the syntax, for example the `extends` keyword of Java becomes a colon in C#.

Then, the runtime system was ported by hand from Java to C#. This required changes to the syntax and the names and classes of library method calls. C# contains some language features not present in Java which are intended to allow clearer code to be written. Where possible these new constructs were used in P#: for example property setters and getters were used where appropriate.

The naming conventions were changed in an attempt to convert idiomatic Prolog names to idiomatic C# names. For example `list_to_string/2` becomes `ListToString_2`. Symbols are translated into a more easily de-codable form. For example `-->/2` becomes `dash_dash_gtr_2`, in contrast to Prolog Café which translates this into `PRED_$454562`. This was implemented in such a way as to ensure that name clashes could not occur. For efficiency reasons the translation is performed in one pass through the name.

By a two stage bootstrapping process we were able to bootstrap P# in the same way that Prolog Café is bootstrapped. This could be achieved by running only Java programs: another Prolog implementation was not required. The T-diagram below illustrates the process.

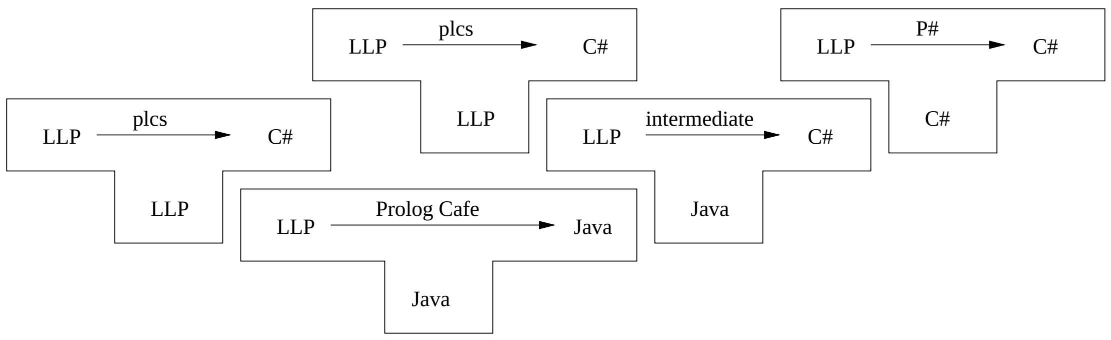


Figure 1: T-diagram

Figure 2 shows the architecture of P#, and Figure 3 shows the process by which a user can generate a standalone application based on P#. The P# runtime system and the C# generated from the part of P# which is written in Prolog are placed together in a DLL. The user's EXE file contains the user's predicates together with a special loader class. When the executable is run the loader class loads the DLL assembly and then starts up the runtime system. The runtime system detects that it has been invoked in this way and is therefore able to call back to the user's EXE assembly to execute the user's main predicate. The user's predicates and those of P# can then execute one another as usual.

P# keeps a list of the assemblies in which to look for predicates. Usually this will consist of the users EXE and the P# DLL, however a `load_assembly` predicate is provided to add an assembly to the list. Thus, the user can split their P# application across multiple assemblies.

In addition an EXE file is provided which runs the P# interpreter. This is a C# program very similar to the special loader mentioned above, which sets the start predicate to the interpreter predicate.

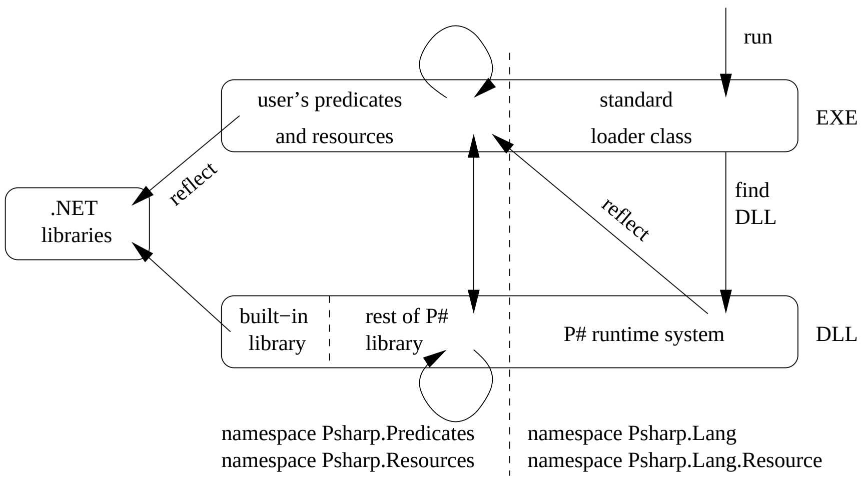


Figure 2: Separation into a DLL and an EXE

The non-LLP Prolog benchmarks provided with Prolog Café were run under P#. P# was usually faster and produced smaller executable files. This improvement is almost entirely due to difference between the languages Java and C#, in particular the compilation scheme is largely unchanged.

As a case study, a game of noughts and crosses was implemented as a C# Web Application with a P# Prolog back-end. The user is able to take any move or ask the computer to take a move. If the computer is asked to take a move, a copy of the current board is passed to a P# Prolog predicate which adds the computer's move to the board and then this is returned to the C# side to be rendered.

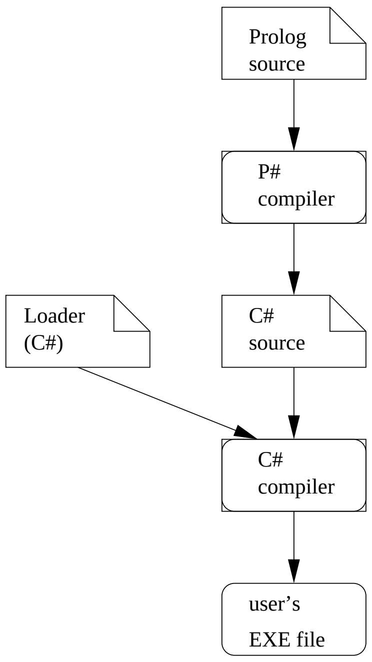


Figure 3: How the user generates their EXE file

2.2 Extending P# with Concurrency Support

P# was extended with new Prolog primitives to allow the concurrency support in C# to be exploited from the Prolog side, see [8].

A mechanism was provided to enable new threads to be forked, and for data to be passed between threads via variables. These features were modelled on existing concurrent versions of Prolog such as Parlog, Aurora, FCP and DeltaProlog, see [6]. Aurora is an OR-parallel Prolog and is designed more for programming parallel processors than for programming in a concurrent language. We provide a form of AND-parallelism similar to FCP and Parlog, but did not use guards as this would have resulted in a language too far removed from Prolog. DeltaProlog is based on CSP and, like P#, has facilities for forking threads and for passing messages between threads.

Firstly, P# had to be made thread-safe. This involved finding static fields and replacing them with instance fields. Also some fields had to be protected by mutexes.

A global database was provided as another means of passing data between threads. Each thread has its own private database which only it can access. All threads are able to read and modify the global database. These accesses are automatically protected by a mutex. The private database is accessed in the usual way with primitives `assert`, `retract` and so on. The global database is accessed using new primitives `global_assert`, `global_retract` and so on. A `global_call` predicate is provided to call facts asserted in the global table.

A new thread is forked by calling the `fork/1` predicate with a structure as an argument. This structure is cloned onto a new Prolog engine and then executed. Any uninstantiated variables in this structure then become concurrent variables shared between the two threads. If either thread instantiates one of these variables then the other thread can take that instantiation by calling `wait_for/1` on that variable. When an instantiation occurs it is added to a queue by asserting information about the instantiation in the global table. `wait_for` waits for such a message to appear in the global table, and when it does it makes the appropriate binding. It is possible for such a variable to be shared between more than two threads: for example if a forked thread forks another thread.

The following example illustrates the use of `fork` and `wait_for`. 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. `guess/1` forks a thread for each letter and waits for one of them to succeed.

```
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 following example shows how the scheme can be integrated to some extent with backtracking. The program calculates the square root of a square integer between 0 and 400. The program forks 21 threads, each of which tries one of the possible square roots. When one of them finds the root, this is reported back to the main thread. Because a conjunction of two goals is a structure with functor `,/2`, we can place such a conjunction in a call to fork. Extra brackets are needed to make clear that this conjunction is a single argument. If the user attempts to find the square root of a non-square integer then the query will fail.

```
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 ).
```

It is possible for a producer to give multiple bindings to a variable on backtracking, and then for `wait_for` to consume each value also on backtracking. As an example the following code consists of a producer which produces the numbers from 0 to 10, and a consumer which doubles each number and prints out the result. The `pulse/2` predicate makes a binding, which sends a message to the consumer, and then undoes it straight away so that it can be bound to a different value on the next call to `pulse`.

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

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

P# keeps track of which of those threads having a copy of a concurrent variable are still running. If it is detected that all such threads are waiting for the same variable, then all these calls to `wait_for` fail. Thus, if we are waiting for a solution and all the threads fail without instantiating the concurrent variable then the call to `wait_for` fails.

I also added predicates to allow other threading features of C# to be used. For example, calls are provided to enter and exit a monitor and to sleep for a specified time. A backtrack-able lock primitive is provided: everything deeper on the proof tree from the call to the backtrack-able lock forms a critical region.

It is possible for concurrent code on either the Prolog or the C# side to interact with code on the other side. A C# thread can call a Prolog predicate defined by the user which forks a new P# Prolog thread. The C# code can then pass messages to and receive messages from the Prolog thread by calling special methods of the object representing the relevant variable. Conversely, a P# Prolog thread can call a predicate which forks a call to a C# method.

A P# Prolog predicate can call a C# method in the following way:

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

The middle argument consists of the method name and any actual arguments. These C# arguments may include uninstantiated variables. Thus, a concurrent variable can be passed from the Prolog side to the C# side. The use of `cs_method/3` should be wrapped in a fork, for example:

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

This would be matched on the C# side by something like:

```
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.

As a case study for the concurrency support in P# we implemented a system of disconnected collaborative agents. A central server keeps a record of a set of facts. A number of agents are each able to connect to the server, to change its facts, and then to disconnect from the server and to change their own records of the facts. When the agent reconnects the server has to be synchronized with the agent. This involves finding which facts conflict and then asking the agent which of the conflicting facts, if any, to use. To provide a notion of conflict, some arguments of a predicate in the database can be marked as key fields.

Having done all of this, as an experiment we timed both the new concurrent version of P# and the non-concurrent version of P# compiling the translation engine Prolog file. We chose this example because it is a long running natural Prolog program. We found that a compilation of the original Prolog to C# translator runs at roughly 90% of the speed that it did before.

2.3 Graphical Interface

I have developed a simple GUI for P#. This allows Prolog code to be edited and either interpreted or compiled to C# and run. The C# libraries allow the C# compiler to be invoked and for the executable to be built in memory rather than on disk. Thus, the GUI has a menu item which compiles a file into memory and allows its predicates to be invoked from the interpreter. As would be expected, compiled P# code runs significantly faster than interpreted code. A screen-shot is included as Figure 4.

2.4 Experiments with Delegates

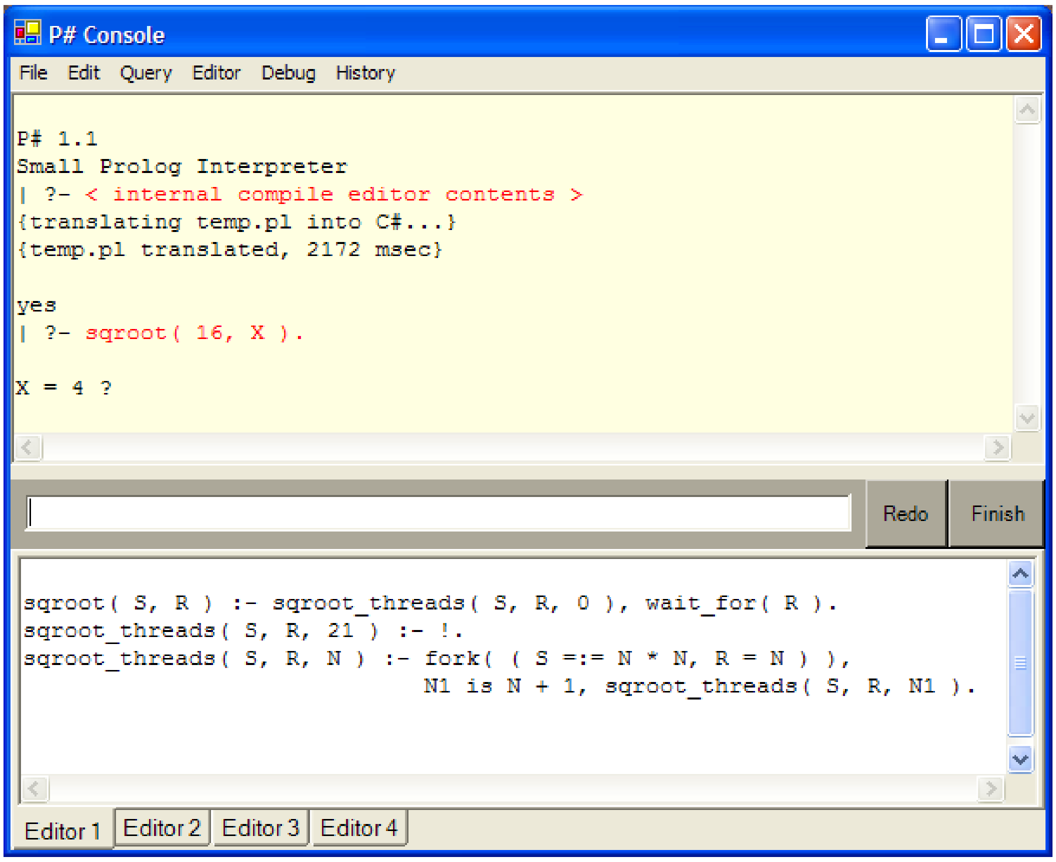
Some experiments were performed regarding the use of delegates and `struct`s to achieve faster compiled code.

P# inherits from Prolog Café a supervisor function scheme for continuation style code. If the C# compiler made use of tail calls this would not be necessary, however, I have not been able to find any code for which this optimisation is used by the C# compiler.

The supervisor function looks approximately like this:

```
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 which stores a pointer to a static `exec()` method, and these can be passed around instead of objects.



```
P# 1.1
Small Prolog Interpreter
| ?- < internal compile editor contents >
{translating temp.pl into C#...}
{temp.pl translated, 2172 msec}

yes
| ?- sqroot( 16, X ).

X = 4 ?
```

```
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 ).
```

Figure 4: Screen-shot

We now have the problem, however, that we cannot pass the arguments by setting the fields in the Predicate object. Instead we can pass them, together with the continuation, as arguments to the `exec()` method. These are wrapped in a object or a struct.

A test program was written to make many simple Prolog calls using the original scheme and using the delegate scheme. It was found that the delegate scheme was slower in the Release build and about the same speed in the Debug build. This is probably because the C# compiler optimises the object oriented code, but is unable to optimise effectively the less natural (from the point of view of usual C# programming) use of delegates. This strengthens our view that attempting to produce code "as a human would write it" could bring efficiency benefits. Also, although the methods pointed to by the delegates are static, the call to the delegate is translated into a virtual call in the MSIL.

2.5 Time Line

May 2001–July 2001: Research into Prolog implementation and writing Thesis Proposal.
August 2001–September 2001: Re-learning Prolog, investigating the code of Prolog Café.
October 2001: Bootstrapping.
November 2001–February 2002: Porting Prolog Café to a naïve compiler to C#, writing paper [7].
March 2002: Delegate Experiment.
March 2002–May 2002: Work on concurrent P#, writing paper [8].
May 2002: Work on the GUI.
July 2002: Work on disconnected agents example.

3 Future Work

3.1 Generating More Idiomatic C#

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 would be possible, it may be the case that much more efficient C# could be generated.

In particular tail recursive predicates which 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 ).
```

This could be compiled into:

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

In fact much tail recursive code can be characterised 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 which 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 which returns a list's tail. Nevertheless making the generated code more idiomatic should ensure that we are working with the C# compiler's optimiser rather than against it.

3.2 Mode Inference

Related projects, such as Mercury, 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 which have not been provided.

Consider the code:

```
:- mode p( in, out ).
```

```
p( X, Y ) :-
    q( X, Z ),
    r( Z, Y ).
```

X must be instantiated on entry into p, because the first argument of p has mode in, so we know that in this case it is instantiated when q is called. Similarly Y must be instantiated after the call to r, so r must instantiate it. Furthermore Z is uninstantiated on entry into q so the second argument of q is not in.

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

I wrote a simple Prolog program which takes as data facts describing some of the modes of a Prolog program and infers as many of the others as possible.

3.3 Integrating Concurrency Support with Linear Logic

At present normal Prolog variables are used as message channels between concurrently executing threads. It may be possible to use linear logic [9, 10] resources as well. This would provide a message channel which can be used only once. The linear logic constructions available may then give rise to interesting programming possibilities. I intend to familiarise myself with linear logic programming in order to decide whether this would be a sensible extension. There is a danger that using linear logic in this way could lead to obscure code.

3.4 Adding Support for Modules

At present the P# namespace is still rather flat. Ideally we would like to provide a full module system, perhaps along the lines of that of SICStus Prolog. We would like to map Prolog modules to C# namespaces. There are some issues here regarding name clashes. Also, we may lose the current good support for separate compilation. At present the programmer has too much access to the internals of P# and could thereby subvert various features. By adding a module system the internal predicates of P# could be better hidden from the programmer.

3.5 Other Work

A new beta version, 0.5.0, of Prolog Café was recently released. Floating point and exception support has been added and the code tidied up considerably. When the final version is released I intend to upgrade P# to match it.

Some simple P# programs will be written for the P# web page. Specifically a graphical tutorial on Dynamic Clause Grammars (DCG), as well as some more conventional examples such as the Eight Queens problem and pentominos.

3.6 Case Study

As a major case study to demonstrate the usefulness of P#, I intend to implement an object-oriented programming "wizard". This would be a program, intended to be integrated into an IDE, which would help the programmer to navigate the programming language's class hierarchy, methods and fields. A programmer might, for example, wish to use a method which takes as an argument an object of a certain type. They might ask the assistant how they would obtain such an object, and the assistant may then search its database for methods which return objects of that class. The implementation of this for C# would involve the Prolog code calling C# reflection methods.

4 Related Projects

4.1 Mercury

Mercury [14] is a functional logic language, and the logic language which users of .NET are recommended to use. Mercury, in contrast to Prolog, is a fully declarative language. Thus, the developers of Mercury did not have to deal with some issues that are a problem for us: specifically cuts. We hope that this will provide scope for original work in developing P#.

The basic syntax of Mercury is similar to Prolog, with added notation for mode declarations and function declarations. The declarative nature of Mercury means that I/O has to be programmed by passing a variable around which represents the current "state". Instead of cuts, users of Mercury are advised to use the `if-then-else` construct which in Mercury does not involve a Prolog style cut.

4.2 HAL

HAL [11] has more of an emphasis on constraint logic programming (CLP), in particular it is designed specifically with the design of constraint solvers in mind. Like Mercury, HAL allows mode declarations, and also allows determinism declarations. For example the user can declare that a predicate will succeed at most once or even never.

4.3 MINERVA

MINERVA [15] is a commercial Prolog to Java translator, and thus performs a similar function to that intended for P#. That is, applications which require a Java/C# front-end and would like a logic language back-end.

5 Timetable

I plan to start the third year by considering how to generate more efficient and natural C#. Prolog code will be written which detects a certain form of tail recursive code and translates it into more efficient code than that produced by the standard P# translation scheme inherited from Prolog Café. Even if this produces no improvement in efficiency, it should improve the readability of the code. This work will probably be the subject of a funding proposal to Microsoft.

When the final version of Prolog Café 0.5.0 is released, P# will be updated to match.

Work may be done on mode inference, and on detecting Prolog idioms other than tail recursion. Also, I will have to familiarise myself with programming in linear logic in order to assess whether useful features can be built using it.

6 Thesis Chapters

Below, I summarise an estimate of the chapter structure of the thesis.

- Introduction : introduction to language translation issues
- Language Interoperability : survey of existing language translation and .NET
- Prolog and Linear Logic
- Producing Naïve C# : minimal modifications of Prolog Café to produce C# instead of Java
- Implementing Concurrent Prolog
- Producing More Efficient Code
- Producing More Readable Code
- Case Study: Object-Oriented Programming Wizard
- Conclusion

Of these the chapters on producing naïve C# and implementing concurrent Prolog have been written in the form of papers, see [7] and [8].

References

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

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

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

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

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

[6] Ciancarini, P., (1992) Parallel Programming with Logic Languages: A Survey, In Journal Computer Languages 17(4) 213–239.

[7] Cook, J. J., (2002) P#: Using Prolog within the .NET Framework. To appear as a University of Edinburgh Technical Report. Available from `http://www.dcs.ed.ac.uk/home/jjc`.

[8] Cook, J. J., (2002) A Concurrent Prolog for Interoperation with C#. Submitted for publication. Available from `http://www.dcs.ed.ac.uk/home/jjc`.

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

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

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

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

[13] Liberty, J. (2001) Programming C#. O'Reilly.

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

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

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

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

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

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