Division of Informatics, University of Edinburgh

Laboratory for Foundations of Computer Science

P#: Using Prolog within the .NET Framework

by

Jonathan Cook

Informatics Research Report EDI-INF-RR-0145

Division of Informatics **July 2002**
http://www.informatics.ed.ac.uk/

P#: Using Prolog within the .NET Framework

Jonathan Cook

Informatics Research Report EDI-INF-RR-0145

DIVISION *of* INFORMATICS
Laboratory for Foundations of Computer Science

July 2002

Abstract :

We discuss P#, our implementation of a tool which allows interoperation between a superset of Prolog and C#. We modify the existing tool Prolog Cafe, which compiles a linear logic extension of Prolog to Java, so that it produces C# instead. This enables us to create C# objects and call their methods from normal Prolog. In particular, we are able to take advantage of the graphical, networking and other libraries. We were able to extend to P#, the ability of Prolog Cafe to compile itself. To use P#, a C# compiler and runtime system are required, but no other Prolog implementation is needed.

Keywords :

P#: Using Prolog within the .NET Framework

Jonathan Cook

Laboratory for Foundations of Computer Science,
University of Edinburgh, EH9 3JZ, UK. jjc@dcs.ed.ac.uk

Abstract. We discuss P#, our implementation of a tool which allows interoperation between a superset of Prolog and C#. We modify the existing tool Prolog Café, which compiles a linear logic extension of Prolog to Java, so that it produces C# instead. This enables us to create C# objects and call their methods from normal Prolog. In particular, we are able to take advantage of the graphical, networking and other libraries. We were able to extend to P#, the ability of Prolog Café to compile itself. To use P#, a C# compiler and runtime system are required, but no other Prolog implementation is needed.

1 Introduction

Translating Prolog to a high-level language, as opposed to native code, has many advantages. One of the most important must be that we gain the ability to interoperate with the target language.

We would like a tool which generates code which executes efficiently and that is to some extent idiomatic and exploits the rich features of modern programming languages. The ideal would be to develop tools which produce readable, well-structured code which can be easily modified by a human. Fully realising this ideal is a long way off, but progress can be made towards it.

There already exist translators which translate from Prolog to C, for example wamcc [7] and those which translate to Java, for example Prolog Café [3][4][5][10][17], and the commercial product MINERVA [15]. Wamcc has an emphasis on efficiency and so produces very unnatural code involving jumps into the middle of functions. However, Java is more restrictive in the way that flow of control can be programmed, and so the output of Prolog Café is slightly more readable. As a consequence it is also slower, but the ability to easily use Java's graphical, networking and other libraries from Prolog is gained.

C# attempts to combine the efficiency of C++ with the elegance of Java. So it seems natural to modify existing translators to translate from Prolog to C#. Hopefully, in this way, we can find a compromise between speed and readability which produces reasonably efficient, well-structured code. In addition, this provides a means of using Prolog within the .NET framework. A functional logic language, called Mercury [14], will be available for use with .NET, but in many cases it could be difficult to translate existing Prolog applications to Mercury.

Another way in which Prolog could be used within the .NET framework is by translating it directly to the .NET intermediate language (MSIL). However, if

we translate through C#, the C# compiler will do much of the optimisation for us, and few languages, if any, are in a better position to produce well optimised MSIL code than C#. This is the main incentive for generating code which is as close as possible to code written by programmers—the C# compiler should be better at optimising this than at optimising machine-generated code.

2 C# and the .NET platform

2.1 C#

C# is a relatively new object-oriented programming language which has drawn on the languages C++ [20] and Java [8][13], in an attempt to combine the efficiency of C++ with the elegance and simplicity of Java. Below the essential points of C# are summarised, see [2] and [12] for more details.

Like Java, C# is compiled into an intermediate language. C# shares certain features of Java not found in C++, such as garbage collection, reflection, thread support, static inner classes, and the ability to add finally clauses to try blocks. C# supports Java-style interfaces, and abstract methods, and like Java does not allow multiple inheritance. The support for multi-threading is similar to that of Java, with locked regions and monitors.

C# also has features of C++ not found in Java, such as operator overloading, namespaces, jumps, enums, 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 [2] 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. It also shares with Java support for networking, something that can be a difficulty when using C++.

2.2 .NET

.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# and .NET are related, each being to some extent designed to work well with the other.

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

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

3 Prolog Café

Prolog Café is a program written by Mutsunori Banbara and Naoyuki Tamura. The most recent version, 0.44, was released in 1999. Prolog Café translates a linear logic extension of Prolog (LLP), via a linear logic extension of the WAM (LLPAM) [21] to Java.

For more information on the WAM, a standard compilation strategy for Prolog, see [1][23][24]. For more information on the Prolog language, see [6].

Prolog Café is an extension of jProlog [11] which uses a continuation passing style of compilation referred to as binarization and detailed in [22].

Prolog Café consists of a run-time system written in Java, which essentially simulates the WAM derivative, and several Prolog/LLP files which 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 (the compiler is bootstrapped). Then both these sets of Java files are put in a JAR file.

The user is able to run this program as a normal, but rather 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 which can then 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.

4 Modifications to the Translator

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

5 Obtaining a Bootstrapped Translator

Prolog Café is bootstrapped, in that it is able to compile that part of itself which is written in Prolog. In fact, this part of the code uses linear logic resources in places, and so cannot be compiled by SICStus [19] Prolog. 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 which 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 which compiles LLP to C#, but which is written in Java, i.e. $P_{Java}^{C\#}$. By then running $P_{Java}^{C\#}$ on T' we obtain a program which compiles LLP to C#, but which is written in C#, i.e. $P_{C\#}^{C\#}$.

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 summarise 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 which 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 which language that program is translating from, and which 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 which translates from LLP to Java in Prolog Café.

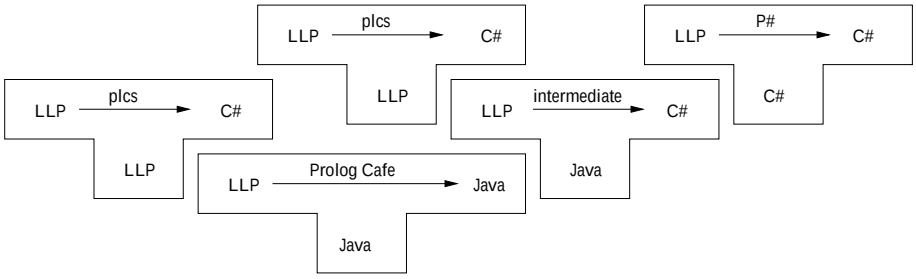

Figure 1: Obtaining a Bootstrapped Translator to C#

6 Other Modifications

The above process produces the C# files corresponding to the translator. It does not, however, produce a run-time system. This had to be hand translated from Java into C#. On the whole this was a straightforward process. The libraries of the two languages are very similar, as are the semantics. Some of C#'s keywords are semantically very similar 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`.

Prolog is a difficult language to parse because the syntax can be dynamically changed by declaring postfix, prefix and infix operators. The parser in Prolog Café heavily relies on the "unget" method in the Java standard library class PushbackReader. There does not appear to be a corresponding class in the C# library yet. Attempts at converting the code to use the "peek" method instead failed due to a subtlety, so this problem was solved by implementing a reader with a one character push back facility in C#.

A slight speed improvement was obtained by converting one of the Prolog stacks from an array implementation to a C# Stack implementation.

C# has inherited from Visual Basic, the ability to declare property setters and getters. This 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 these accesses go through methods which the programmer specifies using a special syntax. This is a pattern which occurs frequently in Prolog Café, and indeed most object oriented code, so where possible they were used in P#.

7 Names and Architecture

7.1 Namespaces

The naming in Prolog Café is fairly flat, relying on prefixes such as "`PRED_`" to denote a predicate. This leads to ugly names and a lack of structure. Prolog Café uses a dollar sign at the start of predicate names to indicate that they are internal to Prolog Café. The choice of a dollar is probably due to the fact that for historical reasons a dollar may occur in a Java class name. In C# dollar signs cannot occur in classnames.

We instead place all predicates in a single namespace, and all resources in a separate namespace. This allows us to use just the predicate name. We do still have to use special representations for punctuation symbols in predicate names, and for instances where a predicate is compiled to more than one class. This is done in such a way as to convert idiomatic Prolog predicate names such as `list_to_string` into idiomatic C# names such as `ListToString`, and `-->` to `dash_dash_gtr`. In contrast Prolog Café translates `-->` into a concatenation of the decimal values of the ASCII characters, `PRED_$454562`, which is difficult for a human to decode. By placing underscores in carefully chosen places when non-idiomatic Prolog names are given we were able to obtain an injective mapping from Prolog names to C# names, which generates the new name in one pass

through the Prolog name. The rules simply rewrite types of character, based on whether the character is at the start of the name, within the name following an underscore or within the name not following an underscore. For example, a lower case letter within the name following an underscore becomes upper case, and the underscore is lost.

7.2 Assemblies

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 which is actually called first, is in control, and calls the user's code. Furthermore, the user may wish to split their code across several files.

The P# interpreter must use reflection to locate 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 a different assembly to 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 a separate assembly. Thus, we added to P# a class storing a list of assemblies, and a predicate which loads a given assembly. This predicate can be called both during an interpreter session and from Prolog compiled to C#. To dynamically find a class P# firstly looks in its own assembly and then tries each of the assemblies in the assembly list.

We decided it was important to protect the user from the issues involved in generating a DLL and then having to make it visible. Thus, the P# runtime system and libraries were placed in a DLL. The user, having used P# to generate C# files for their predicates, 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, i.e. the user's assembly, and then adds that assembly to the list of assemblies we mentioned above. P# can now find the user's main predicate by reflection and call it. This process is summarised in Figure 2. Usually after the first two reflections have occurred the predicates which will be called can be determined statically, thus there is very little overhead associated with the use of reflection. The one exception to this is where the user reads or alters a C# field or invokes a C# method. In this case we use reflection. As with Prolog Café, some of the more usual 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.

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

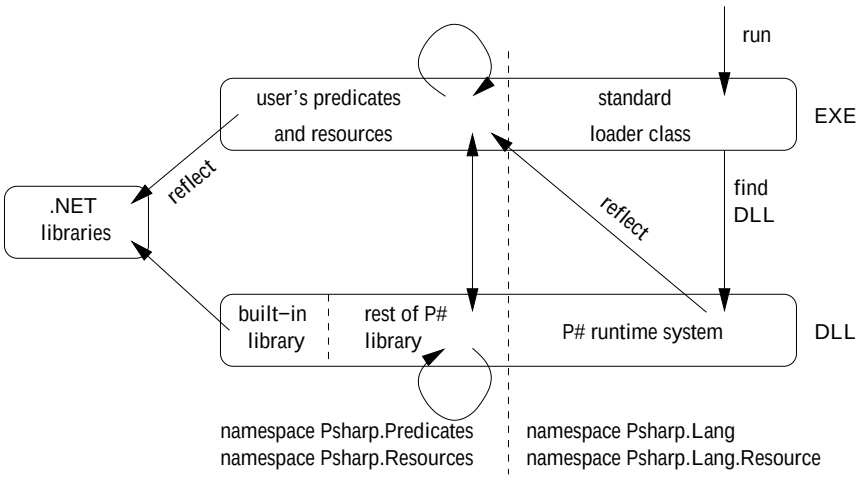

Figure 2: Separation into a DLL and an EXE

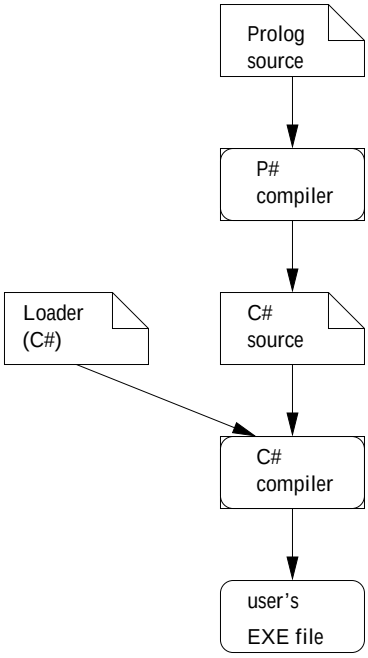

Figure 3: How the user generates their EXE file

which contains only a class very similar to the Loader mentioned above. This loader runs the DLL directly in interpreter mode. Thus, we have essentially allowed P# to be used as either a DLL or an EXE.

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.

8 Benchmarking

We benchmarked P# using the benchmarks provided with Prolog Café. For each benchmark, we first compiled the Prolog into Java and ran the Java using Prolog Café, recording both the times for compilation and execution of each benchmark. The compilation time is recorded in the column "Café comp." in Table 1. In the case of execution we performed the test using both Sun's Java SDK for Windows and Microsoft's JVM. These times are recorded in the columns "Café exec (Sun)" and "Café exec (MS)" respectively. For the compilation we used Sun's SDK only.

We then compiled each Prolog benchmark into C# and ran the C# using P#, again recording all the times. These times are shown in the columns "P# comp." and "P# exec.". All times are in milliseconds.

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 sizes of files in the table are given in bytes. The Prolog Café JAR file, which contains the compiled class files for runtime system and library is roughly 3 Megabytes in size. The analogous file for P#, the DLL, is roughly 1 Megabyte in size.

The bottom row of the table indicates where appropriate the average over the tests of the ratio of the Prolog Café value to the P# value, expressed as a percentage improvement. For the raw data a smaller value is better (less time or less code). Thus, this average gives an indication of the improvement (average speedup or average reduction in size of executable code).

Part of the speedup we found could be due to the fact that C# compiles all MSIL to native code before starting to run the benchmark, whereas Java often JIT compiles code on demand. Thus, the Java timings may include some JIT compilation, whereas the C# timings should include none.

9 Simple Example

Figure 4 shows a simple Prolog clause, and Figure 5 shows part of the C# class that it is compiled into.

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

Figure 4: A simple Prolog clause

Table 1. Benchmark results

Benchmark	P# comp.	Café comp.	P# size	Café size	P# exec.	Café exec (Sun)	Café exec (MS)
boyer	8,281	8,985	196,608	393,027	3085	3445	2964
browse	4,609	4,906	45,056	85,550	1282	1091	1122
chat_parser	30,109	34,359	389,120	1,027,170	701	1402	891
crypt	3,703	3,890	32,768	55,253	61	131	50
divide10	2,547	2,968	16,896	29,196	20	30	30
fast_mu	3,625	3,750	32,768	41,242	40	60	50
flatten	6,110	6,469	65,536	141,194	371	551	381
log10	2,578	2,797	16,896	29,214	30	30	30
meta_qsort	3,672	4,047	45,056	86,659	80	230	100
mu	2,609	2,891	28,672	36,687	31	50	50
nreverse	2,266	2,453	9,728	15,480	40	121	60
ops8	2,609	2,828	16,896	29,249	30	30	30
poly_10	4,734	4,250	49,152	94,523	150	370	200
prover	3,609	4,000	45,056	73,411	70	100	60
qsort	2,640	2,735	12,288	18,939	10	30	21
queens (8 all)	3,828	3,454	32,768	50,998	81	121	60
queens (10 all)					1702	1001	1552
queens (16 first)					1072	631	1012
query	2,828	3,250	40,960	85,499	80	221	100
reducer	9,157	10,359	131,072	298,555	431	751	491
serialise	2,968	3,468	28,672	33,191	30	40	40
tak	2,610	2,360	6,656	6,294	391	561	531
times10	2,563	3,312	16,896	29,191	20	50	21
unify	10,500	12,563	77,824	164,246	120	240	130
zebra	4,094	4,641	28,672	26,986	150	270	130
average improvement of P#		↑ 9%		↑ 72%		↑ 73%	↑ 16%

```
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 BrotherOrSister_2 : Predicate {

    public Term arg1, arg2;

    public BrotherOrSister_2(Term a1, Term a2, Predicate cont) {
        arg1 = a1;
        arg2 = a2;
        this.cont = cont;
    }

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

    public override Predicate exec() {
        engine.setB0();
        Term a1, a2, a3;
        Predicate p1;
        a1 = arg1.dereference();
        a2 = arg2.dereference();

        a3 = engine.makeVariable();
        p1 = new Predicates.Parent_2(a3, a2, cont);
        return new Predicates.Parent_2(a3, a1, p1);
    }

    ...[methods to return the predicate's functor and arity]...
}
}
```

Figure 5: C# code

The C# code is almost identical to that produced by Prolog Café, 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 which is to be called next, and then calls the next predicate. So after the `exec()` method exits the predicate generated by `new Predicates.Parent_2(a3, a1, p1);` will be called, and if successful then `p1` will be called. If `p1` is successful, `cont` will be called.

10 Case Study: Noughts and Crosses

One of the most likely uses of our tool is to combine a Prolog back-end with a Windows front-end, such as a Web service. As a case study we discuss how we implemented a GUI front-end to a Prolog program which allows the user to play a game of noughts and crosses.

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

We implemented the game as a Web Application, where at every point in the game the user is allowed to either take the move themself or to ask the Prolog program to take the next move. Thus, it is possible to discover how the program will respond to any configuration of the board. Also, the user can play the computer or another user, or the computer can play itself.

The current board is stored at the C# level, and passed as a parameter to a Prolog predicate on each move. The predicate returns the new board, which the C# code then draws. The calling mechanism is that inherited from Prolog Café: the arguments are constructed as Prolog terms, and these are used to construct a Predicate object which can then be called, with the call blocking until the Predicate succeeds or fails. A screenshot of a game that the X player has won is included as Figure 6.

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

11 Future Work

Recall our function to convert Prolog names into C# names. Code to perform the name translation is required in both the translator (written in Prolog) and the run-time system (now written in C#). 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 detect instances where such a translation can be done automatically and perform the translation to produce more idiomatic C#. As with similar projects (HAL [9], Mercury) we may support this by allowing the Prolog code to

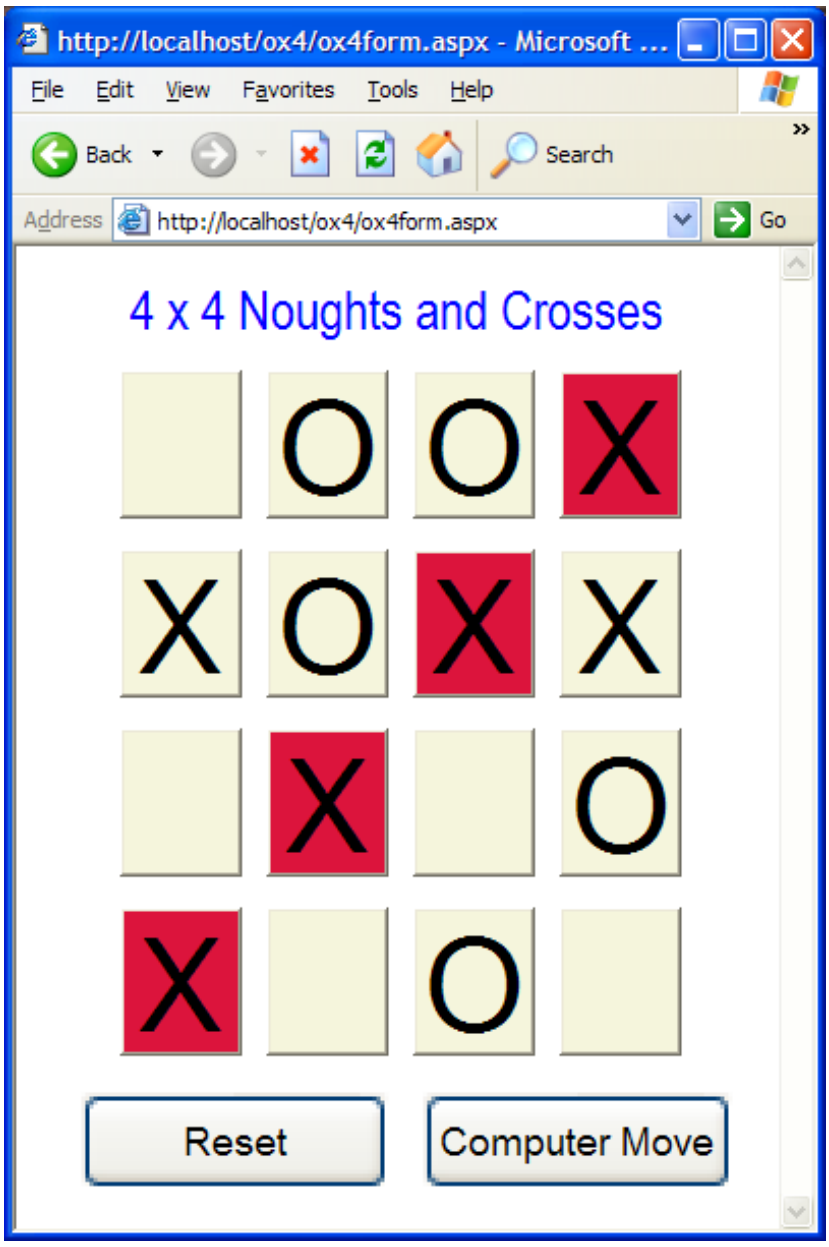

Figure 6: A Web Application

be annotated with mode declarations. In this case the renaming predicate would have one input and one output parameter.

Greater efficiency may be obtained by using `structs` rather than classes in some places, as these can be placed on the stack rather than on the heap.

One of the main innovations of wamcc was to exploit a feature of a dialect of C to allow very fast indirect branching (some WAM instructions branch to a location stored in a register). Essentially the idea was to jump into the middle of functions, thereby avoiding the overhead of function calls. Delegates and `gotos` 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 Predicates, but not Terms, by storing pointers to static `exec()` methods in delegates. Initial experiments suggest, however, that this would not improve efficiency as it seems to defeat optimisations which are performed by the C# compiler or JIT compiler on code which involves frequent object creation. This strengthens our view that attempting to produce code "as a human would write it" could bring efficiency benefits.

`gotos` in C# are 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.

If we cannot obtain very readable C# code, then we can at least make the interfaces to the code simple and clear so that programs can easily incorporate the translated code into a C# project.

The correctness of the WAM has been proved in Isabelle/HOL, as described in the paper [18]. It may be possible to extend this to cover any modifications we make to the WAM in pursuit of translation to C#. Certainly if programmers are ever to use code from such a translator, and they only work with the interfaces, we would like to be confident that the generated code is correct.

There is scope for new work on implementing some of the many existing extensions of Prolog. In particular it should be interesting to improve support for concurrency by taking advantage of the support for multi-threading which C# shares with Java. We could model this on existing concurrent versions of Prolog. We would want the concurrency to be explicit, with the programmer explicitly stating in the Prolog source code where it is to be used. In general we would 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 the tool. This could be achieved, for example, by adding a predicate whose effect is to begin executing another predicate on a new thread, with an entirely new set of stacks. Unlike Java, C# has methods which can be used to enter or exit a monitor at any point. In Java it is only possible to either mark a block of code as synchronized or as a special case

of this to mark an entire method as synchronized. Thus, it should be possible to add to P# predicates for performing these two operations. One challenge lies in finding the most appropriate way to store and allow concurrent access to shared data.

We also intend to add a full module system, probably based on that of SICStus Prolog. This 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.

12 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.dcs.ed.ac.uk/home/jjc/`

Acknowledgements

I would like to acknowledge Mutsunori Banbara and Naoyuki Tamura, the authors of Prolog Café, the tool on which ours is based.

I would also like to acknowledge the kind advice and assistance of Stephen Gilmore; and the support of the EPSRC.

References

1. Aït-Kaci, H., (1999) Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (out of print) Available from `http://www.isg.sfu.ca/~hak`
2. Albahrari, B. A Comparative Overview of C#. Available from `http://genamics.com/developer/csharp_comparative.htm`
3. Banbara, M., Tamura, N., (1999) Translating a Linear Logic Programming Language into Java. In Proceedings of ICLP'99 Workshop, 1999.
4. 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.
5. 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.
6. Clocksin, W. F., Mellish, C. S., (1994) Programming in Prolog 4th Edition. Springer.
7. Codognet, P., Diaz, D., wamcc: Compiling Prolog to C. In PLILP'95.
8. Gosling, J., Joy, B., Steele, G., Brancha, G., (2000) The Java Language Specification, Second Edition. Addison Wesley.
9. The HAL home page: `http://www.csse.monash.edu.au/~mbanda/hal/`

10. 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.
11. jprolog home page. http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/
12. Liberty, J. (2001) Programming C#. O'Reilly.
13. Lindholm, T. and Yellin, F. (1999) The Java Virtual Machine Specification, Second Edition. Addison Wesley Longman Inc.
14. The Mercury home page: http://www.cs.mu.oz.au/research/mercury/
15. MINERVA home page: http://www.ifcomputer.com/MINERVA/
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. Pusch, C., (1996) Verification of Compiler Correctness for the WAM. In Proceedings of the 29th International Conference, TPHOLs'96, August 1996, Turku, Finland. LNCS 1125 pp.347–361.
19. SICStus Prolog home page: http://www.sics.se/sicstus/
20. Stroustrup, B., (2000) The C++ Programming Language, Special Edition. Addison Wesley.
21. 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.
22. 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.
23. Warren, D. H. D, (1983) An Abstract Prolog Instruction Set. Technical note 309, SRI International, Menlo Park, CA, October 1983.
24. Warren, D. H. D, (1988) Implementation of Prolog. Lecture notes, Tutorial No. 3, 5th International Conference and Symposium on Logic Programming, Seattle, WA, August 1988.