A Concurrent Prolog for Interoperation with C#

Jonathan Cook

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

Abstract. We have previously modified a tool which translates Prolog to Java, so that it produces C# instead, see [7]. We now add language constructs on the Prolog side to allow concurrent Prolog code to be written. The changes are made with C# interoperation in mind. We add a primitive predicate which allows a structure to be evaluated 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, with accesses automatically protected by a mutex. All of the new features are available to the programmer via new built-in Prolog predicates.

1 Introduction

Modern languages such as Java and C# [9] have built-in concurrency. For languages such as C and C++ various libraries are available for concurrent programming. Older languages such as Fortran and Prolog have often had various forms of concurrency or parallelism added to them. Sometimes they have been the inspiration for concurrent languages which are similar to but incompatible with them. Most Prolog implementations are built on languages which do not have as good support for concurrency as C#. Our tool, P#, is a Prolog implementation which translates LLP/Prolog to C# source code. We have added to P# a form of concurrency designed with interoperation with C# in mind.

We wish to provide a model of multi-threading for Prolog which retains Prolog as a subset. No existing P# Prolog program should be broken, provided it happens not to use predicate names to which we give new meaning. In addition we would like programming multi-threaded operations to 'feel like' programming in Prolog. Finally we seek a model which naturally and efficiently integrates with the C# threading model. This is, we want clean integration on the C# side as well as the Prolog side.

We add to P# several new built-in predicates (there are no changes to the Prolog syntax) which 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 (the C# term for notification).

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

We also allow interaction between threads by providing a global database which all threads are able to read and modify, with automatic locking, while also giving 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.

As a core part of the .NET platform [10] C# is an important language, and its interoperation with older languages is useful. When designing support for interoperation between two languages, it is desirable to add features to obtain close integration between the two languages, without existing code being broken or rendered excessively inefficient. When one of the languages possesses a powerful construct which the other lacks, in this case concurrency, it can be highly beneficial to build an equivalent implementation of the construct in the language without it.

2 Prolog Café and P#

Prolog Café [2–4, 8, 11] is a program which was developed by Mutsunori Banbara and Naoyuki Tamura. The most recent version, 0.5.0 beta, was released in April 2002. P# is based on Prolog Café version 0.4.4. Prolog Café translates a linear logic extension of Prolog (LLP), via a linear logic extension of the WAM (LLPAM) [13] to Java.

For more information on the WAM, a standard compilation strategy for Prolog, see [1, 14, 15]. For more information on the Prolog language, see [6].

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 can be 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 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, `main/0`. Prolog Café can then be told to run this predicate, instead of the interpreter, 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.

We modified Prolog Café so that it compiles LLP to C# rather than Java. This enables us to create C# objects and call their methods from normal Prolog. We hand-translated the run-time system from Java into C#, and modified the translation engine (written in Prolog) to take account of the syntactic differences between Java and C#. We altered the name generation code to produce more natural names in the generated C# code, and gave P# an architecture suitable for use within the .NET Framework, see [7].

3 Making P# Thread-Safe

Prolog Café 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, results in chaos because various static fields are 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, as 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.

3.1 The Runtime System

A search for problematic static data in the runtime system points to three things:

Firstly, there is a problem with the choice point frame stack which encapsulates an array of choice point entries. The object representing an entry contains a static field, which is used to optimise the operation of this stack. This problem is solved by moving the static field into the stack object, where it becomes 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` (representing a Prolog variable). The next value to be allocated is stored in a static field. This field needs only to be protected by a mutex.

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

3.2 Changes to the Translator

Having made these changes, multiple engines are still unable to execute concurrently. The problem now is with the `Predicate` objects. Each Prolog predicate is translated into a C# class which 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 in use, and another instance field for the continuation, that is the `Predicate` to run next. This would be fine, but for the fact that predicates which have more than one clause have static fields which 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 which was statically created. Thus, when two or more threads try to manipulate the engine and continuation fields of this object, they can interfere with each other.

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

The engine field is only ever used in the `exec()` method of the `Predicate`. This method is always called by the engine. Usually a method is called just before such a call to initialise the engine value. Instead we add an argument to the `exec()` method, whereby the engine can pass a reference to itself to each call to `exec()`. This should not be significantly slower, it may even be faster.

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 we create a new local variable instead of using the object field, and initialise this from the engine. Again, this should incur no serious performance penalty.

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

3.3 First Call Optimisation

Finally a problem arises with the implementation of first call optimisation (FCO). FCO is used when the head and first body goal of a clause are the same predicate. It allows the first body goal to call the predicate without consuming an unnecessary choice point. When the `Predicate` class representing the first clause is entered, a reference to this `Predicate` object is stored in the *static* field of the `Predicate` class, `entry_code`. A subclass representing a later clause may then use this value to call the first clause again. It seems that we should move this data into the engine. We cannot, however, place it in a simple variable in the engine to be retrieved when needed. This is because between the storing of the value and its recall, another predicate which also uses FCO might 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 is to use a hash-table in the engine which maps `Predicate` classes (that is, the type of the Predicate object) to Predicate objects (that is the current `entry_code` for that predicate). This seems a slightly heavyweight solution, but is efficient as hash-tables provide fast lookup.

3.4 Summary

Thus, we found that we had to make changes to both the runtime system and to the translator. Having made all of these changes, multiple engines could run different (or the same) Prolog programs concurrently, without interference.

4 Existing Concurrent Prolog Derivatives

We now survey the paper [5], indicating how it influenced the design of our concurrent version of Prolog.

4.1 Aurora

Aurora is an OR-parallel Prolog, well suited to the parallel programming of parallel processors. This means that a predicate which is marked as parallel can have its *clauses* evaluated in parallel.

Aurora provides two types of database modification primitives. With one, the alteration to the database can occur concurrently, asynchronously, anywhere in the proof tree. Use of this requires the programmer to use locking primitives. The other blocks until it is in the leftmost branch of the search tree, adding some degree of determinism.

The aims of Aurora are very different to those of P#. We are 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`, although the user is not required to use explicit locking.

4.2 Flat Concurrent Prolog

Flat Concurrent Prolog, or FCP, is a stream-based parallel logic language. Communication is via shared variables. Each clause is guarded by goals. The clause which 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 an 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 [5]):

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

FCP has, from our point of view, a serious disadvantage. This 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#.

4.3 Parlog

Parlog 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.

4.4 DeltaProlog

DeltaProlog is an extension of Prolog based on CSP. AND-parallelism is achieved via a "fork goal". Communication is via special event goals, which come in an few different flavours depending on whether the goal is backtrackable and whether it is synchronous.

5 Forking New Threads

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 forks a new thread which evaluates the structure.

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

5.1 Implementation

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 which have 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` which can be reached from the functor's arguments is correctly cloned in this way. The entire cloned structure is then built in a new Prolog engine, and executed.

6 A Global Table

Each forked thread is equipped with a private database which 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.

6.1 Implementation

If one thread places a term in the global table, and another retrieves it, then that term passes from one engine to another. Whenever a term does this it must be rebuilt in the new engine, since `VariableTerm`s make use of the engine's trail. As a general solution, we provide a special engine for the global table. When a term is globally asserted it is first built on the global table engine, then the Prolog assertion code is run on the global table engine. We do not wish to create a new thread every time this occurs, so the assertion code is run on the global table *engine*, but on the thread which 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 an internal predicate, `$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, using an internal predicate. 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.

7 Monitors

We provide two further primitives, `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.

Similar predicates are provided for waiting and pulsing.

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. Then, the monitor is 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 on the proof tree forms a critical region.

8 A form of AND-parallelism: `wait_for/1`

The `wait_for` predicate takes as an argument a variable which 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.

The programmer should regard these variables are special output variables, to be instantiated with care, only when it is known for sure what value should be passed to any waiting thread.

8.1 Simple Example

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 perfectly acceptable for a forked thread to wait for the thread which forked it or for forked threads to, in turn, fork more threads. This 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 (integers in the above example) can be any Prolog term. Thus, it would be possible, for example, to pass a predicate call to be evaluated.

8.2 Implementation

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.

wait_for 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 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 each other.

8.3 Backtracking Example

The following example shows that forking integrates to some extent with backtracking.

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

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

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

The guess/1 predicate 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. The variable Z correctly retains its concurrent information on backtracking, as it comes into existence as soon as guess(Z) is called.

8.4 Square Root Example

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", provided that the programmer does not forget the extra brackets.

9 Obtaining Multiple Solutions

It may be that we wish to use a concurrent variable more than once, indeed if we cannot, then some problems will require unnatural solutions.

If a concurrent variable having been bound, is later unbound by backtracking, and then bound again to the same or a different value, then that new binding is asserted in the global table.

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 implementation of `wait_for` contains sufficient locking for this to work correctly.

`wait_for` 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 `backtrackable_lock` predicate discussed above is used to ensure that a lock is held between obtaining each solution. This lock is released by `wait_for` when it actually begins to wait for the next solution because `wait_for` calls the C# `Wait()` method.

The following code consists of a producer which produces the integers from 0 to 10 and a consumer which doubles each one and outputs the 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 ).

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 ).
```

The `ConcurrentVariable` object is notified of the termination of any forked thread which has a copy of its variable. When it is detected that all of those threads which are still running are waiting for the same 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. 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.

It is possible for there to be multiple producers or even multiple consumers. In the case of there being more than one consumer, each produced term will be consumed exactly once by exactly one of the consumers.

10 Case Study: Disconnected Shared Database

We now give an example which illustrates the usefulness of our concurrency support in Prolog. Suppose that we have a central database which stores a generic set of facts. Several users are able to connect to this database, and 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. In particular, while a particular 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 which 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 starting, it 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. The user is able to connect to and disconnect from the server by selecting appropriate menu items. When disconnected, facts may be asserted to 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 is asked to ask the user which one of the set of conflicting facts should be used. When the user has specified this fact, all of the other facts in the conflicting set are retracted. Finally, the new database is broadcast to all the currently connected agents.

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

Recall that a `wait_for` call detects when all the remaining threads are waiting for a certain variable. We exploit this by passing a variable to all forked threads and then waiting for that variable on the main thread. This ensures that the main thread waits until all conflicts have been resolved before it broadcasts the new database to the agents.

The code to maintain the database and to detect and resolve conflicts is all 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.

11 Interoperation with C#

The locks dealt with by `lock` and `unlock` are C# locks (on C# objects). Similarly `fork` initialises 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 which 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 Prolog before the Prolog terminates.

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, in which case the C# will be passed a `VariableTerm` object. Thus, a concurrent variable can be passed from the Prolog side to the C# side. This time 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.

12 Performance

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. This is, in our opinion, an acceptable penalty given the greatly enhanced possibilities for interoperation with C#, some of which we have demonstrated in this paper. Thus, Prolog programs which do not involve concurrency experience a relatively small overhead from these changes.

In a concurrent program, the most important overhead is due to extra work which must be done when a unification which involves a concurrent variable is performed. 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 to be expected that when a unification occurs which will cause a value to be passed to another thread, some overhead will be involved.

13 Conclusion

We have implemented a concurrent version of Prolog which 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 leads 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 this are the way in which a conjunction of goals can be passed to a fork, and the fact that a message may consist of any Prolog term. The higher order and type-less nature of Prolog afforded a freedom which allowed many of the features mentioned in this paper to be easily implemented partly in Prolog.

Acknowledgements

I would like to acknowledge Mutsunori Banbara and Naoyuki Tamura, the developers 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 financial 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. 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. Ciancarini, P., (1992) Parallel Programming with Logic Languages: A Survey, In Journal Computer Languages 17(4) 213–239.
6. Clocksin, W. F., Mellish, C. S., (1994) Programming in Prolog 4th Edition. Springer.
7. Cook, J. J., (2002) P#: Using Prolog within the .NET Framework. Division of Informatics, University of Edinburgh Technical Report. Available from `http://www.dcs.ed.ac.uk/home/jjc`.
8. 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.
9. Liberty, J. (2001) Programming C#. O'Reilly.
10. The Microsoft Developer .NET home page. `http://msdn.microsoft.com/net`
11. Prolog Café home page: `http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/index-jp.html`
12. SICStus Prolog home page: `http://www.sics.se/sicstus/`
13. 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.
14. Warren, D. H. D, (1983) An Abstract Prolog Instruction Set. Technical note 309, SRI International, Menlo Park, CA, October 1983.
15. 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.