Experiences in Transporting Concurrent Prolog and the Bagel Simulator to LM-Prolog

Part I of a Report of a Visit to ICOT November 1983

Kenneth M. Kahn

This is an informal discussion of the experiences Ehud Shapiro and I had in transporting Concurrent Prolog [Shapiro 1983] and the Bagel Simulator [Shapiro 1984] written in Dec10 Prolog to LM-Prolog. This report assumes the reader is familiar with Concurrent Prolog. LM-Prolog [Carlsson 1983] is a Prolog system written for Lisp Machines [Moon 1983]. Since the implementation of LM-Prolog is accessible it was possible to modify the system to better accommodate the needs of Concurrent Prolog. This was a good test of the user-extensible unification of LM-Prolog. Also it was an opportunity to see if various extensions to Dec10 Prolog incorporated into LM-Prolog were of use in running Concurrent Prolog.

The Initial Transport

For the purposes of comparison we first ported Concurrent Prolog to LM-Prolog by using a syntactic translator. There were minor difficulties in running the translated programs. LM-Prolog distinguishes between symbols and terms of no arguments. "true" and "(true)" are different in LM-Prolog and both are written as "true" in Dec10 Prolog. The translator made several mistakes of this sort which were corrected by hand. Another difficulty is that the body of clauses in LM-Prolog is simply a list of predications, while in Dec10 Prolog they are binary terms whose principle functor is "," (i.e. conjunction). This necessitated the translation of a list of goals to "(and g-1 (and g-2 ...))". Read only variables were translated incorrectly by the translator and needed to be fixed by hand. After a few hours of making such changes, we had Concurrent Prolog running. We timed the execution of naive reverse of a list 30 long, and found it took about 65 seconds in breadth-first mode and about 35 in depth-first. (I.e. nearly 200 times slower than compiled LM-Prolog and about 60 times slower than interpreted.)

Direct Modification of Unification

The first experiment that I performed was to modify the part of LM-Prolog which implements the unification of lists with lists. The idea was very simply to special case any term which was read-only (i.e. whose "car" began with "??"). This cut the time for depth-first naive reverse in half.

This modification to LM-Prolog was both inelegant and slowed down the normal operation of LM-Prolog since a few instructions were added to the unification of compound terms. Considering that the experiment took less than one hour it was worthwhile.

Representing read-only variables as ordinary variables with special names

The next idea was to represent read-only variables the same as ordinary variables except that their names are lists of their name rather than symbols. LM-Prolog maintains the names of variables for the purposes of printing and debugging, so the idea was to use names to distinguish the two types of variables.

This sped up Concurrent Prolog even further. (About 12 seconds for naive reverse.) This scheme no longer slowed down the ordinary execution of LM-Prolog as much (though the basic binding primitive needed to test the type of the name of unbound variables). This scheme was incorrect in general however, as I discovered when I tried to run other examples. The difficulty was that in following chains of invisible pointers between value cells, the system could pass by one that was read-only and thereby allow it to be bound ("written"). The scheme was also not very elegant, and probably was a waste of the half day that was invested in it.

Representing Read-only variables as Lisp Machine Flavor instances

The LM-Prolog unifier is user-extensible. The idea is that when a non-variable term is unified with a flavor instance, the instance is sent a "unify" message with the other term as an argument. This facility costs very little if not used. The LM-Prolog system uses it to implement lazy and eager values created by lazy and eager "bag-of" and "set-of" and constraints created by "freezing" predications [Carlsson 1983].

It was very easy to define the flavor for read-only variables. When an instance receives a "unify" message it checks to see if the value cell of the variable it is an access to is bound or not. If it is not it fails. If it is then it calls "unify" on its binding and the argument of the "unify" message. It replies to the message "variable-p" with "t" or "nil" depending upon whether its variable is unbound or not. It replies to the message "ordinary-term" (used in the interface to Lisp) with itself if unbound, otherwise with the value of its variable.

The difficulty with using this facility to implement Concurrent Prolog's read-only variables is in creating the flavor instances. The other applications of flavor instances create instances in special primitive predicates. Read-only is a syntactic property of variables. The solution was to extended the LM-Prolog parser, so that read-only variables have their own "template type". A template in LM-Prolog is like a structure in a structure-sharing Prolog. The set of types of templates was extended to include read-only. Then the system functions for parsing, unifying terms with templates, and constructing terms from templates were extended. Only the parsing function was thereby slowed down if no read-only variables are used. The others were extended by adding new cases after the existing ones so that no performance degradation was involved.

This third experiment was very successful. It took about half a day to implement and debug. Its performance was equivalent to the variable name scheme, but this one was more general and modular. It also entailed no run time over head when read-only variables are not used (except for assert and compile which need to parse). It had two additional advantages. The syntax of read-only was now more flexible (I choose to follow the Dec10 implementation and use "?" at the end of the name). Also, the existence of a read-only access for a bound variable was made invisible to the entire system including all built-in predicates and Lisp. This is a consequence of the definition of the "ordinary-term" method. Unlike the Dec10 implementation, there is no need to remove "?" when printing or doing arithmetic. Concurrent Prolog primitives such as "wait" were thereby greatly simplified and their performance enhanced. Also LM-Prolog's built-in predicates "variable" and "not-variable" behave correctly for read-only accesses.

Along with this change in the representation of read-only variables another optimization became feasible. When LM-Prolog interprets a predication, it tries to unify the heads of the appropriate clauses with the predication. Only if the unification is successful is the body of the clause copied. This is why the clause is represented in a structure-sharing like "template" structure. This could now be done by Concurrent Prolog. A change to the "guarded-clause" predicate significantly improved the performance of Concurrent Prolog in the case where the unification of the goal with the head of the first clause (or clauses) fails.

Use of Multiple Worlds to Enhance Performance

LM-Prolog finds the definition of a predicate by searching through a list of the current worlds called the universe. A world is a set of predicates. This search is only done the first time a predicate is used in the current universe. One rather expensive aspect of the Dec10 implementation is that in order to implement tracing and the Bagel simulator, the system does run-time checks to see if they are enabled. This happens so frequently that the system could be speed up by about 40% by eliminating the trace predicates. One wants to be able to trace the execution of Concurrent Prolog, however, so the solution to this was to have traced and untraced versions of pieces of the implementation in different worlds. Tracing works fine in the LM-Prolog implementation and yet costs nothing when not used.

After these optimizations and a few others, the implementation of Concurrent Prolog was running naive reverse of a list 30 long in about 4.5 seconds — the same speed as the Dec10 implementation on the 2060. This is especially noteworthy when one considers that Dec10 Prolog on the 2060 typically runs about 8 times faster than LM-Prolog on a 3600 running Symbolics Release 4.5. (LM-Prolog runs about 15% faster in Release 5.0.) I began to break down that 4.5 seconds into components and discovered that about .75 seconds went into unification, about .5 seconds to copying bodies of clauses and .1 seconds to finding clauses in the database. The rest is presumably in reduction and scheduling.

Use of Worlds to Maintain Different Versions of Concurrent Prolog

We have already seen how worlds were used to maintain tracing and non-tracing versions of Concurrent Prolog. It was also used to have two versions of the predicate "schedule1" where one of them lived in the "depth-first-cp" world. By simply adding or removing that world from the current universe we can switch from breadth-first to depth-first processing.

The Implementation of the Bagel Simulator

Shapiro has extended the Concurrent Prolog interpreter to allow one to specify using turtle commands where a process should be executed [Shapiro 1984]. The Bagel upon which such programs are run is a torus of processors which communicate only by sending messages. This Bagel is simulated by displaying the current state of the processes running on each processor. The processors are mapped to a rectangular grid on a display.

The Bagel simulator in LM-Prolog exploits the Lisp Machine's window system and high resolution graphics. The displaying of the processes can be done in any font and at any raster location. Arrows for up, down, left, and right are real arrows as opposed to the approximations used on the VT100 Bagel simulator display.

A more significant experiment done on the LM-Prolog Bagel implementation was the introduction and use of real numbers for the position and heading of processes. The position is rounded down to the nearest integer when determining what processor it is running on. This is a more flexible and general scheme than the one forced upon the Dec10 implementation by the lack of real numbers. For example, a very large array relaxation problem where the problem's dimensions are 10 times larger than the Bagel can be programmed where the turtle commands are "forward .1" rather than "forward 1". In this way several processes are time shared upon the same processor. Using the "wrap around" facility of the Bagel would be inappropriate here since it would unnecessarily increase the amount of communication required. Also the "real turtle" version of the Bagel does not need the twist ed wrap-around that the integer version needs. For example, in order to create an infinite linear pipe exploiting all the processors of the Bagel, one can either begin the program headed at some small angle or in addition to going forward also going to the left a small amount. Again, using worlds one can have both versions and switch between them easily.

Prolog in Concurrent Prolog in LM-Prolog on the Bagel

One question I explored is how to implement Prolog in Concurrent Prolog. The difficulty is to achieve the full non-determinism of Prolog in Concurrent Prolog which lacks "don't-know" non-determinism. A more practical and general, but more complex, solution to this problem can be found in [Hirakawa 1984]. The program I developed is:

```
(define-cp pr ;;top-level queries come here
  ((pr ?query)
   (prove (and ?query (and (prolog (print ?query)) (fail))))))
```

```
(define-cp prove ;;equivalent to Prolog's "call"
  ((prove (true)))
  ((prove (and (true) ?b) (prove ?b)))
  ((prove (and ?a ?b)) (commit (system ?a) (sequential-and ?a ?b)))
  ((prove (and ?a ?b))
   (commit (clauses ?a ?clauses) (try-each ?clauses ?a ?b))))

(define-cp try-each
  ((try-each ((?goal ?ignore-guard ?body) . ?) ?goal ?cont)
   ;;if its a Prolog clause it shouldn't have a commit
   (commit (prove (and ?body ?cont)) (true)))
  ((try-each (? . ?more-clauses) ?goal ?cont)
   (commit (try-each ?more-clauses ?goal ?cont) (true))))
```

The next question which I began to explore is how to allow the Prolog programs to use the Bagel. Slightly simplified, the following two clauses can be added to "prove":

```
((prove (and (@ ?a ?turtle-program) ?b))
 (commit (system ?a) (@ (sequential-and ?a ?b) ?turtle-program)))

((prove (and (@ ?a ?turtle-prolog) ?b))
 (commit (clauses ?a ?clauses)
         (@ (try-each ?clauses ?a
                      (@ ?b (run-backwards ?turtle-program)))
          ?turtle-program)))
```

Where "run-backwards" is a turtle command that causes it to interpret its turtle program "backwards", i.e. last clause first and all arguments become the negative of their original value. This is used to return the rest of the conjunction to its original location. How to do this more elegantly and efficiently is an interesting topic for future thought.

Topics for Future Research

The integration of read-only variables with other parts of LM-Prolog could be worthwhile. In particular, the idea of using them for communication between LM-Prolog processes created by "eager-bag-of" is very promising. Read-only variables may provide an alternative to "freeze" [Colmerauer 1982].

The implementation of Concurrent Prolog does busy waiting. If instead, "unify" suspended when unbound read-only variables are unified then efficiency can be improved significantly for suspended processes. Also the Concurrent Prolog primitive "otherwise" could then be implemented correctly. Perhaps "freeze" would be ideal for implementing this change.

Adding a general ability to define arbitrary appearances to processes. Perhaps this is a good way to do graphics and animation?

Applying the techniques of partial evaluation to the implementation of Concurrent Prolog. This could be done upon the LM-Prolog implementation of Concurrent Prolog using a Prolog partial evaluator. A prerequisite to this is to re-write the interpreter without "cut". This would enable one to compile Concurrent Prolog programs into ordinary Prolog. Another idea would be to re-write the interpreter in Lisp and then run a Lisp partial evaluator on it. This would enable one to compile Concurrent Prolog programs into Lisp. See [Kahn 1982] for a more detailed discussion of this.

Addendum

Upon return to Sweden, I showed Mats Carlsson the extensions I had made to LM-Prolog to support read only variable references. He then modified the micro-code support for LM-Prolog accordingly. We then proceeded to benchmark Concurrent Prolog on the Cadr Lisp Machine and we surprised that it took about 7.5 seconds to compute naive reverse of a list 30 long. We then used LM-Prolog's interface to the Lisp Machine's metering facility and discovered that more than half the time was going into a routine for converting LM-Prolog terms into Lisp s-expressions. In general, this is not necessary since the representations are compatible. We quickly tracked down the source and added an extra argument to the call to the Lisp interface specifying that LM-Prolog terms should be passed to Lisp unchanged. Naive reverse then ran in about 3 seconds. While at ICOT, I had access to their Symbolics 3600 which currently lacks a functioning metering facility so I was not able to discover this performance bug.

Acknowledgments

I wish to thank all the people at ICOT for a fascinating visit and a chance to explore the issues discussed in this paper. I also wish to thank Ehud Shapiro for his help in transferring his programs to LM-Prolog.

References

[Carlsson 1983] Carlsson, M., Kahn, K.
"LM-Prolog User Manual", UPMAIL Technical Report No. 24,
Uppsala University, Sweden, Nov. 1983

[Colmerauer 1982] Colmerauer, A.
"PROLOG II Manuel de Reference et Modele Theorique",
Proc. Prolog Programming Environments Workshop,
Linkoping Sweden, March 1982

[Hirakawa 1984] Hirakawa, H., Chikayama, T., Furukawa, K.
"Eager and Lazy Enumerations in Concurrent Prolog",
submitted for publication

[Kahn 1982] Kahn, K.
"A Partial Evaluator of Lisp written in Prolog",
First International Logic Programming Conference,
Marseille, France, Sept. 1982

[Moon 1983] Moon, D., Stallman, R. M., Weinreb, D.
"Lisp Machine Manual", MIT AI Laboratory, January 1983

[Shapiro 1983] Shapiro, E.
A Subset of Concurrent Prolog and Its Interpreter
ICOT Technical Report, TR-003, ICOT, Tokyo, 1983

[Shapiro 1984] Shapiro, E.
"The Bagel: A Systolic Concurrent Prolog Machine",
to appear as an ICOT Technical Report, 1984

Kenneth M. Kahn

UPMAIL, Uppsala Program Methodology and Artificial Intelligence Laboratory
Computing Science Department, Uppsala University, P.O Box 2059
S-750 02 Uppsala, Sweden
Tel: Domestic: (018) 15 54 00 ex. 1840, International: +46-18-11 19 25

Professional Objectives:

To do research and teaching in computer science. My interests are primarily in artificial intelligence, language design, education, and computer graphics.

Educational Background:

Massachusetts Institute of Technology, Sept. 1973 to Feb. 1979
Ph.D in Electrical Engineering and Computer Science, Jan. 1979
Thesis: "Creation of Computer Animation from Story Descriptions"
M.S. in Electrical Engineering, Aug. 1975
Thesis: "Mechanization of Temporal Knowledge"

University of Pennsylvania, September 1969 to June 1971 and September 1972 to June 1973
B.A. in Economics *magnum cum laude* with distinction, June 1973
Thesis: "Bankruptcy, Information Costs and Equilibrium"

University of Stockholm, September 1971 to June 1972

Research Experience:

From 1973 to 1975 I implemented two versions of a time specialist, a computer program capable of accepting a wide range of temporal statements, checking their consistency, and making inferences to answer questions. Following that I implemented in Logo several small systems used by elementary school children for programming animation and natural language. Until 1980, I was involved in the design, implementation, and use of an actor-based computer language called "Director" for use in programming knowledge-oriented computer animation. Simoultaneously I was working on my thesis project involving the creation a system capable of making simple computer animated films in response to vague incomplete story descriptions. In 1981 I designed and implemented a language based upon extended unification called "Uniform". The language is an attempt to combine the important features of Lisp, actor languages, and Prolog into a simple coherent framework. I am the co-author of LM-Prolog, a commercially available state-of-the-art Prolog system on Lisp Machines. I am also working on a project to automatically generate a compiler from LM-Prolog to Lisp, by partial evaluating the LM-Prolog interpreter written in Lisp. The partial evaluator is an LM-Prolog program that I have developed that generates efficient specializations of Lisp programs. I am also building a partial evaluator for LM-Prolog.