PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

The Compilation of Prolog Programs without the Use of a Prolog Compiler

Kenneth M. Kahn & Mats Carlsson

Uppsala Programming Methodology and Artificial Intelligence Laboratory
Department of Computing Science, Uppsala University
P.O. Box 2059, S-750 02 Uppsala, Sweden

The research reported herein was supported by the National Swedish Board for Technical Development (STU).

Abstract

An efficient Prolog interpreter written in Lisp is presented. The interpreter is then specialized to run different Prolog predicates. These specializations are generated automatically by a *partial evaluator* for Lisp programs called "Partial Lisp". It transforms Lisp programs to other Lisp programs and knows nothing about Prolog. It is argued that the partial evaluation of interpreters can be a substitute for compilation. The results of partial evaluating the Prolog interpreter for simple Prolog predicates are presented. The speed of the specialized interpreters has been found to be about ten times faster than ordinary interpretation. These speeds compare favorably with an optimizing compiler for the same Prolog dialect and computer system. The advantages of using partial evaluation upon an interpreter include a much smaller and easily modifiable implementation. The major difficulty in generating thousands of small specialized interpreters is that it currently takes about two orders of magnitude more time than compilation. Different approaches to reducing partial evaluation time are presented. The possibilities of specializing the interpreter for different uses of the same Prolog predicate are discussed.

1 Motivation

The research presented here is based upon the hope that computer languages can be implemented by simple interpreters. The advantages of implementing systems as interpreters are many. The size and complexity of the implementation is drastically reduced compared to the traditional approach of implementing languages as compilers. Additionally, many modern programming languages are implemented as dual systems containing both an interpreter and compiler. Interpreters are preferred for development and debugging, while compilation is usually necessary for production. The maintenance and modification of dual systems is difficult since the subsystems must be kept compatible. Too many bugs are of the sort "my program worked interpretively but not compiled (or visa versa)".

This paper argues that a system consisting of a very large collection of specialized interpreters is an attractive alternative to the traditional interpreter/compiler systems. Some of these interpreters are very specific, while others are rather general. Ideally, these interpreters should be small and share as much code as possible. As a rule, the more specific an interpreter is, the more efficient it is. The system is arranged so that programs get run by the most specific appropriate interpreter. One of the interpreters is distinguished by being the most general one. It is only this interpreter that is written by a programmer. The specialized interpreters are generated automatically from the general interpreter by a partial evaluator.

Partial evaluation provides a means of implementing a language solely as an interpreter without sacrificing efficiency. In the experiment presented in this paper, a Prolog interpreter, a Lisp partial evaluator, and a Lisp compiler together generate efficient specialized Prolog interpreters that are of about the same size and speed as compiled Prolog programs. A major advantage of partial evaluation is that the partial evaluator remains fixed as one changes the interpreter. The introduction of a new language feature is accomplished by modifying just the interpreter. The partial evaluator can play the role of a compiler for any language in which an interpreter exists written in the language of the partial evaluator. One can view a traditional compiler as a partial evaluator specialized for a particular language. In practice, however, compilers are not as flexible as partial evaluators. Prolog compilers, for example, *specialize* the interpreter to run different Prolog predicates. The only extent to which they can specialize the interpreter for particular uses a predicate is limited to *mode declarations* [Warren 1977].

Lisp was chosen as the implementation language both because it is well-suited for partial evaluation and because of the availability of Lisp Machines [Greenblatt 1974] for doing this research. We regard Lisp as an excellent machine language. Prolog was chosen because it is an interesting high-level language and because existing Prolog compilers can be used for comparison. LM-Prolog was chosen because it is a serious Prolog implementation on Lisp Machines. In principle, one can write a partial evaluator for any language and it can be run upon any interpreter.

2 Partial Evaluation

Partial evaluation is a relatively new program manipulation technique that is being used to optimize programs, generate programs automatically, open-code functions, efficiently extend languages, generate compilers, and compile programs. It is this latter capability that is the focus of this paper.

A partial evaluator is an interpreter that, with only partial information about a program's inputs, produces a specialized version of the program which exploits the partial information. For example, even a simple partial evaluator for Lisp can partial evaluate the form (append x y) to (cons f y) when x is known to be (list f). This process begins by opening the form with the definition of append.

```
(defun append (front back)
  (cond ((null front) back)
        (t (cons (first front) (append (rest front) back)))))
```

The cond is encountered, causing the form (null front) to be partial evaluated. Since front is bound to x which is known to be (list f) this evaluates to nil. This conclusion is based upon the definition of null and list as

```
(defun null (x) (eq x nil))
(defun list (x) (cons x nil))
```

The system can decide that (eq (cons f nil) nil) must return nil since its arguments are of different types. This reduces the original problem to

```
(cons (first (list f))
      (append (rest (list f)) y))
```

which becomes (cons f y) since (first (cons f nil)) reduces to f and
(append (rest (list f)) y) reduces to (append nil y) which reduces to y.

The basic operation of our partial evaluator, "Partial Lisp", mirrors that of an ordinary Lisp evaluator. Variables partial evaluate to their values if bound, otherwise to themselves. Special forms such as cond have special handlers. Ordinary forms are partial evaluated by applying the function to the evaluated arguments. Lambda application works by binding the variables and then partial evaluating the body. Unlike ordinary lambda application, it then performs lambda abstraction to pull out any expressions which otherwise might be recomputed redundantly. For example,

```
((lambda (x y) (list x x y))
 (f x)
 (g x))
```

partial evaluates to

```
((lambda (x') (list x' x' (g x)))
 (f x))
```

Most of the power of the system comes from the capabilities of the handlers for the basic Lisp primitives cond, car, cdr, cons, eq, typep, rplaca, and rplacd when given only partial information. Conditionals, for example, work by partial evaluating the test. If the test is decidable, then the result is the result of partial evaluating the appropriate branch. If the test is not decidable, then the true branch is partial evaluated with the added information that the test and its consequences are true and the false branch with the information that the test is false.

Partial Lisp is able to deal very effectively with a small set of Lisp primitives. Except for these primitives, all functions are known to the system only by their Lisp definition. The major disadvantage with this design is that commonly used Lisp functions such as equal, atom, and nth could be slow to partial evaluate since the system must reason about the definition of these functions. Atom for example is defined as (lambda (x) (eq (eq (typep x) 'list) nil)). Reasoning in terms of a small set of primitives takes too much time. This problem is one of the most serious obstacles for practical use of partial evaluation.

One solution which was taken by the developers of the Redfun partial evaluator [Beckman *et al.* 1976] is to hand-code *handlers* for the most common Lisp functions. Equal, for example, has the same status as eq in their system. Not only does this require a much larger implementation effort but the problem reappears in handling commonly used application functions. Partial evaluating, say LM-Prolog's unify function, would either be very slow or require a special handler. Writing special handlers for the basic units of an interpreter would defeat most of the advantages of partial evaluation, since it is not very different from writing a compiler.

Another solution to this problem was proposed in [Futamura 1971]. The idea is to apply the partial evaluator to itself, producing specializations for running equal or unify or whatever. This solution puts very stringent requirements upon the partial evaluator. It must be both very powerful and very cleanly implemented in order to specialize itself. We began working on partial evaluation with this goal in mind ([Kahn 1982] and [Kahn 1984]) and have come to appreciate how difficult it is. We believe it continues to be a good area for research.

The approach to this problem taken in the partial evaluation of the LM-Prolog interpreter is based upon a powerful *generalization* mechanism built into Partial Lisp. The idea is that one can "train" the partial evaluator upon typical examples. In training mode the system maintains a record of the partial information that it uses to partial evaluate an example. It then abstracts the original problem and the result based upon what aspects of the problem were used and which ones were ignored. The abstracted problem and solution are then added to a data base. By this means one can build up the equivalent of a handler for the most common uses of a function. For example, partial evaluating the problem (unify nil nil) in training mode generates a clause which says that if both arguments to unify are fully specified and are eq then it partial evaluates to t.

In addition to handlers for a dozen or so Lisp primitives and the ability to generate handlers for the common uses of Lisp functions, Partial Lisp consists of a simple transformation module. The few transformations that the system performs preserve equivalence and do not affect the run-time performance of programs. Their purpose is to facilitate further partial evaluation. For example, a function of a conditional is transformed into a conditional where the function is applied on each branch. This gives the function involved more useful partial information about its arguments than the original conditional.

The remaining part of Partial Lisp is a recursion handler. When a program might possibly recurse, then the system

1. generalizes the problem by replacing forms which it considers uninteresting by variables,
2. makes up a name, ϕ, for a function,
3. adds a clause to the database which states that if a problem similar to the current one occurs replace it by a call to ϕ and note that a recursive call was generated,
4. partial evaluates the problem as generalized in step 1 to ξ,
5. if a recursive call was generated, fills in the body of ϕ with ξ and returns a call to ϕ, otherwise returns ξ.

Partial Lisp has no understanding of iteration primitives such as prog and do. This greatly simplifies the system. Iteration is assumed to be programmed as tail recursion. A separate module is applied after partial evaluation to convert tail recursion into a more efficient program using prog and go.

3 The LM-Prolog Interpreter

Initially, the plan for this project was to write a very powerful partial evaluator for Pure Lisp and to write a Prolog interpreter in Pure Lisp. The advantages of this approach are many. The interpreter is very simple. Side-effects, especially non-local ones, can cause partial evaluators to lose what partial information they have, preventing them from making further optimizations. An interpreter in pure Lisp which was considered can be found in [Kahn 1983/1984].

This approach was abandoned for several reasons. The primary difficulty is that such an interpreter spends most of its time searching for the current bindings of variables. The one we considered used alists and spent most of its time inside of assoc. An interpreter using alists for variable bindings continues to do so even after being partial evaluated.

Instead of starting with a toy interpreter, a production quality interpreter was used. We chose the LM-Prolog interpreter we had developed for other purposes. This was considered to be a more serious test of the practicality of the partial evaluation of interpreters.

LM-Prolog represents its run-time terms as Lisp s-expressions. Variables are represented by a special ZetaLisp pointer called a *locative*. Variables are bound to variables via ZetaLisp's *invisible pointers*. Invisible pointers are also used within terms so that dereferencing is done automatically by ZetaLisp's first (= car) and rest1 (= cdr).

LM-Prolog is based upon *lazy structure copying*. Clauses at define time are transformed into a *template* structure. The four types of templates are "ground term", "first occurrence of a variable", "repeated occurrence of a variable", and "non-ground compound term". The templates for variables contain indices into a scratch pad vector.

This template representation can be viewed as a very simple, yet effective, compilation of unification. It is particularly amenable to microcoded implementations using dispatch hardware. Templates can be viewed as a crude abstract instruction set which is interpreted by the function (or instruction) unify-term-with-template which is presented below. This interpretation process can also be compiled by partial evaluation. Our representation is analogous to the unification instructions of D.H.D. Warren's "New Engine" [Warren 1983], although in his scheme the head compiles to a sequence of instructions whereas in ours it compiles to an s-expression and a single function call (or machine instruction).

The control structure of LM-Prolog is based upon *success continuations*. A success continuation is passed downward and invoked upon the successful execution of a goal. This control structure tends to make heavy use of Lisp's control stack. Two important optimizations which alleviate this problem are performed when either the goal is known to be deterministic or it is the last one in the body of a clause.

The basic cycle of the interpreter is as follows:

1. Find the clauses for the current goal.
2. One by one try each clause, by unifying the goal (which is a term) and the head of the clause (which is a template structure). If successful, then prove the conjunction of the goals created by *constructing* the body of the clause (i.e. converting the template structure into a term thereby renaming the variables in the clause which is necessary to avoid variable conflicts between simultaneous uses of the same clause).
3. A conjunction is proved by proving the first goal with a continuation which proves the rest of the goals with the original continuation.

This scheme constructs a copy of the body of a clause only if the goal successfully unifies with the head. Additionally only those parts of the head that contain variables and are unified with variables are constructed. Since terms are represented by Lisp s-expressions the system can use Lisp's primitives for creating, manipulating, and printing s-expressions. An important property of the scheme is the fact that the environment is local to step 2 of the interpreter cycle, and so only a global scratch pad vector is needed.

Other properties of the scheme are discussed in [Warren 1983] where it is labeled "Goal Stacking".

More details about the implementation of LM-Prolog can be found in [Kahn & Carlsson 1984], [Carlsson & Kahn 1983], and [Carlsson forthcoming].

A few minor changes were made to the LM-Prolog interpreter for the convenience of the partial evaluator. The partial evaluator cannot generalize results that make use of special or global variables. LM-Prolog's global scratch pad vector is replaced by a vector which is created on each predicate call. The construction of these vectors are partial evaluated away so there no run-time cost. The production version of LM-Prolog uses a global trail of changes to implement backtracking. This was left for this experiment since there are few optimizations that can be performed upon trailing.

In some versions of LM-Prolog, the functions unify, unify-term-with-template, construct, reference and dereference are implemented in microcode. In such cases, the partial evaluator can be told to leave calls to these functions alone. The interpreter which we present below, unlike the production version, for simplicity's sake does not support some of LM-Prolog's more exotic features such as the optional occur check, circularity handling, and lazy values. Both interpreters exploit ZetaLisp's *area* mechanism for storage allocation. The partial evaluator can deal with this, however, in this paper we have edited away references to areas.

As an interface to compiled predicates, each predicate has a Lisp function associated with it. These functions, called *provers*, return nil upon failure and invoke the continuation if successful.

Compiled predicates are translated to provers. Provers of interpreted predicates typically just call try-each, which tries each clause of the definition: (Predicates containing Prolog's cut are handled by slightly more complicated provers.)

```
(defun p-prover (continuation &rest arguments)
 (try-each clauses-for-p continuation arguments
           *trail* ;;the current trail pointer
           (make-list p-number-of-arguments)))
```

Clauses are represented as ZetaLisp flavor instances [Moon et al. 1983] which contain a cons of a template for the head and a template for the body. This is for the convenience of the database procedures and slows down the interpreter just a few percent. The *special form* deffun is like ZetaLisp's defun except that it transforms tail recursion into iteration.

```
(deffun try-each (clauses continuation arguments mark vector)
  ;;this tries each clause by unifying its head with
  ;;ARGUMENTS. VECTOR is the local environment.
  ;;MARK is the trail marker.
  (cond ((null clauses) nil) ;;no clauses left so fail
        ((let ((templates (send (first clauses) ':templates)))
           (and (unify-term-with-template
                  arguments (first templates) vector)
                ;;unification of head and goal succeeded
                ;;so construct the body and executed it
                (prove-conjunction
                 (construct (rest1 templates) vector)
                 continuation))))
        (t ;;something failed, so undo any bindings made
           (untrail mark)
           (try-each
            (rest1 clauses)
            continuation arguments mark vector))))
```

The production version of try-each compiles open and is not invoked at all if there are no clauses. It calls prove-conjunction tail-recursively in the last clause. Prove-conjunction proves a conjunction and, if successful, invokes a continuation.

```
(deffun prove-conjunction (predications continuation)
  ;;if the conjunction of PREDICATIONS can be proved,
  ;;invoke CONTINUATION
  (cond ((null predications) (invoke continuation))
        (t (let* ((predication (first predications))
                  (definition
                    (current-definition (first predication))))
             (cond ((definition-deterministic definition)
                    ;;goal is deterministic so prove the
                    ;;first goal and try the remainder
                    (cond ((apply (definition-prover definition)
                                  (cons (continuation (true))
                                        (rest1 predication)))
                           (prove-conjunction
                            (rest1 predications)
                            continuation))))
                   (t
                    ;;goal is nondet. so prove the first goal
                    ;;with continuation to prove the remainder
                    (apply (definition-prover definition)
                           (cons
                            (continuation
                             (prove-conjunction
                              (rest1 predications)
                              continuation))
                            (rest1 predication)))))))))
```

The production version of prove-conjunction is not invoked at all if the conjunction is empty. It also avoids constructing a continuation for the last goal and calls try-each directly for simple interpreted predicates, instead of calling their provers. This enables prove-conjunction to tail-recurse even for some nondeterministic predicates.

Unification between terms and templates essentially dispatches on the template code:

```
(deffun unify-term-with-template (term template vector)
  ;;unifies TERM with a coded representation of a term
  ;;in the environment of VECTOR
  (let ((type (first template)))
    (cond ((eq type 0) ;;template is a ground term
           (unify term (rest1 template)))
          ((eq type 1) ;;first occurrence of variable
           (cond ((eq -1 (rest1 template))) ;;void variable
                 (t ;;permanent var—store term in vector
                  (setf (nth (rest1 template) vector) term)
                  t)))
          ((eq type 2) ;;repeated occurrence of variable
           ;;retrieve value and unify
           (unify term
                  (dereference (nth (rest1 template) vector))))
          ((eq type 3) ;;complex template
           (cond ((consp term) ;;unify corresponding parts
                  (and (unify-term-with-template
                         (first term) (second template) vector)
                       (unify-term-with-template
                         (rest1 term) (rest2 template) vector)))
                 ((value-cell-p term)
                  ;;construct and unify
                  (unify term (construct template vector))))))))
```

Unification of terms is straightforward:

```
(deffun unify (x y);;unifier of two terms (s-expressions)
  (cond ((eq x y))
        ((and (consp x) (consp y))
         ;;both are conses, so unify corresponding parts
         (and (unify (first x) (first y))
              (unify (rest1 x) (rest1 y))))
        ((value-cell-p x) (bind-cell x y));;bind X (a variable)
        ((value-cell-p y) (bind-cell y x));;bind Y (a variable)
        ((equal x y)))) ;;may be EQUAL strings or numbers
```

The production version of unify optionally handles cyclic terms or optionally performs an occur check.

Construction of terms essentially dispatches on the template code. Ground terms are truly structure shared.

```
(deffun construct (template vector)
  ;;convert TEMPLATE into an ordinary term
  ;;in the environment of VECTOR
  (let ((type (first template)))
    (cond ((eq type 0) (rest1 template)) ;;ground term
          ((eq type 1) ;;first occurrence of a variable
           (cond ((eq -1 (rest1 template));;only occurrence
                  (cell))
                 (t ;;permanent variable
                  (let ((cell (cell)))
                    (setf (nth (rest1 template) vector) cell)
                    cell))))
          ((eq type 2)
           ;;subsequent occurrence, so retrieve its value
           (dereference (nth (rest1 template) vector)))
          ((eq type 3)
           ;;construct compound term from parts
           (prolog-cons (construct (second template) vector)
                        (construct (rest2 template) vector))))))
```

Macro definitions.

```
(defsubst value-cell-p (x) (eq (typep x) 'locative))
(defsubst prolog-cons (x y)
  (cons (reference x) (reference y)))
(defsubst bind-cell (cell value)
  (set-contents cell (reference value))
  (trail cell))
```

```
(defsubst invoke (x)
 ;;Continuations in LM-Prolog are currently represented by
 ;;a list which is invoked as follows.
 ;;The macro CONTINUATION produces such lists.
 (apply (first x) (rest1 x)))
```

The partial evaluator is not given the definitions of reference, dereference, cell and set-contents. These deal with locatives and invisible pointers that are beyond the scope of the partial evaluator's current capabilities. Also the definitions of trail and untrail are not given to Partial Lisp because of their use of arrays. Most of these functions are given definitions in the partial evaluator as if they were primitive Lisp functions. These definitions take up a page or two and would not have been necessary if Partial Lisp could handle these data types better. The special handlers for trail and untrail exist only to eliminate those calls to untrail that have nothing to untrail.

4 The "Compilation" Of =

Let us consider the compilation of the predicate = defined by the LM-Prolog clause ((= ?x ?x)) (in Dec-10 Prolog that would be X=X.). The following prover is generated:

```
(deffun =-prover (continuation &rest arguments)
  (try-each clauses-for-=
            continuation
            arguments ;;the arguments to a call to =
            *trail*
            (make-list 1))) ;;there is just one variable in =
```

If we declare that = has exactly two arguments x and y (this could easily be done automatically) then the prover is transformed to

```
(deffun =-prover (continuation x y)
  (try-each clauses-for-=
            continuation
            (list (dereference x) (dereference y))
            *trail*
            (make-list 1)))
```

This transformation is justified by the semantics of ZetaLisp's &rest arguments. It is not performed in the production version of the interpreter since the reconstruction of the list of arguments would be wasteful.

Let us inspect how the partial evaluator specialized the interpreter for running =. The following is a selection from the trace of partial evaluating the body of =-prover.

1. Partial evaluate the head of the first clause.

```
(unify-term-with-template
 (list (dereference x) (dereference y))
 '(3 (1 . 0) 3 (2 . 0) 0) ;;template for (?x ?x)
 (list nil))

-partial evaluates to →
  (cond ((unify (dereference y) (dereference x)) t))
```

The above is a consequence of the following three partial evaluations:

```
(unify-term-with-template
 (dereference x)
 '(1 . 0) ;;first occurrence of ?x
 (list nil))
-partial evaluates to →
  t ;;in addition it changes (list nil) to (list (dereference x))

(unify-term-with-template
 (dereference y)
 '(2 . 0) ;;subsequent occurrence of ?x
 (list (dereference x)))
-partial evaluates to →
  (unify (dereference y) (dereference x))

(unify-term-with-template
 nil '(0) ;;template for nil
 (list (dereference x)))
-partial evaluates to → (unify nil nil)
-partial evaluates to → t
```

2. Partial evaluate the (empty) body of the first clause.

```
(construct '(0) (list (dereference x)))
-partial evaluates to → nil

(prove-conjunction nil continuation)
-partial evaluates to → (invoke continuation)
```

3. Partial evaluate trying the (non-existing) remaining clauses.

```
(try-each nil
          continuation
          (list (dereference x) (dereference y))
          mark
          (list (dereference x)))
-partial evaluates to → nil
```

After partial evaluation the prover looks like

```
(deffun =-prover (continuation x y)
  ;;i.e. if X and Y unify then invoke the CONTINUATION
  (cond ((unify (dereference y) (dereference x))
         (invoke continuation))))
```

The resulting version of the prover is equivalent to the prover produced by LM-Prolog's compiler. With microcode support, the predication (= 1 1) takes 76 microseconds with = compiled versus 584 interpreted. Without microcode support, it takes 179 microseconds compiled and 1395 interpreted. All timings given are for a Cadr Lisp Machine.

5 The "Compilation" Of member

The following is the LM-Prolog definition of the predicate member:

```
(define-predicate member (:options (:argument-list element list))
  ((member ?x (?x . ?)))
  ((member ?x (? . ?y)) (member ?x ?y)))
```

In Dec-10 Prolog it would look like:

```
member(X,[X,_]).
member(X,[_,Y]) :- member(X,Y).
```

The following is the interpreter's prover:

```
(deffun member-prover (continuation element list)
  (try-each clauses-for-member
            continuation
            (list (dereference element) (dereference list))
            *trail*
            (make-list 2)))
```

The partial evaluator transforms the above into the following:

```
(deffun member-prover (continuation element list)
  (let ((mark *trail*)) ;;save away the trail marker
    (let ((dlist (dereference list))
          (delement (dereference element)))
     (cond ((consp dlist) ;;the second argument is a cons
            (cond ((and (unify (first dlist) delement)
                        (invoke continuation)))
                  (t
                   ;;backtrack point, restore env.
                   (untrail mark)
                   ;;tail-recurse
                   (member-prover
                    continuation delement (rest1 dlist)))))
           ((value-cell-p dlist) ;;the second argument is unbound
            (cond ((progn (set-contents dlist
                                        (prolog-cons delement
                                                     (cell)))
                          (trail dlist)
                          (invoke continuation)))
                  (t
                   ;;backtrack point, restore env.
                   (untrail mark)
                   (let ((cell_16 (cell)))
                     (set-contents
                      dlist (prolog-cons (cell) cell_16))
                     (trail dlist)
                     ;;tail-recurse
                     (member-prover
                      continuation delement cell_16)))))))))
```

One important optimization is in the recursive call after either unification or the continuation failed. The "compilation" of the next clause was aided significantly by noting that the second argument has already been determined to be a cons. The unification of the goal with the head of the second clause is thereby compiled away completely. We are not aware of any compiler that performs such an optimization.

The interpreter specialized to run member is larger than member compiled by LM-Prolog. The body of the second clause is duplicated. This can be avoided by giving the appropriate advice to the partial evaluator, however, this would interfere with various optimizations performed. An important area for further research is how to handle this trade-off between the size of the programs produced and their effectivity.

In the following table, the timings are given for deciding that a constant is a member of a list of 1000 constants where it is the last element. The timings are repeated for those running with and those running without special microcode support (microcoded unify, untrail, cell, etc.). All times are in milliseconds.

	Microcoded	Without Microcode
Partial Evaluated	72	270
Compiled	103	352
Interpreted	523	2250

Dec-10 Prolog's compiler and LM-Prolog's compiler accept mode declarations indicating whether an argument to a predicate is always unbound (indicated by a name beginning with "-"), always bound (indicated by a name beginning with "+"), or anything else. It was trivial to add this ability to the partial evaluation of the LM-Prolog interpreter. All that was needed was a simple procedure for converting a mode declaration to Lisp predicates. Suppose we declare that member's argument list is (element +list). The partial evaluator runs as before, except it now knows that (not (value-cell-p +list)) is true. This declaration is sufficient for determining that the entire cond clause beginning with (value-cell-p dlist) can be eliminated. In this case the mode declaration simply makes the compiled code more compact. As we shall see in the discussion of concatenate sometimes mode declarations can significantly improve the run-time performance of a specialized interpreter.

This handling of mode declarations as a list of Lisp predicates leads naturally to a very general ability to describe how a predicate is to be used. Here we see one of the real strengths of partial evaluation over compilation. Consider the predication (member (?key . ?value) ?alist) where either the system concludes or the user declares that ?key is a symbol, that ?value is unbound, and that ?alist is a ground list of conses. Partial Lisp currently cannot understand that something is a list of conses or is ground. The system is being extended in a general fashion to handle such partial information. Given such a declaration and the knowledge that the call is deterministic the system should be able to generate a specialization of member-prover as follows:

```
(deffun member-prover_33 (element list)
  (let ((dlist (dereference list))
        (delement (dereference element)))
    (cond ((consp dlist)
           (cond ((eq (first (first dlist)) (first delement))
                  ;;the key is the same as the first of the pair
                  (set-contents (rest1 delement)
                                (rest1 (first dlist)))
                  ;;bind the value and trail it
                  (trail (rest1 element)))
                 (t (member-prover_33 delement
                                      (rest1 dlist))))))))
```

This is quite close to ZetaLisp's definition of assq.

This manner of handling mode declarations and their generalizations as lists of Lisp predicates has the additional advantage that it is trivial to, when desired, compile in run-time checks that the declaration is correct. For example, to check that the declaration (element +list) is correct the prover can be transformed to

```
(cond ((not (value-cell-p +list)) (try-each ...))
      (t <signal error>))
```

6 Relation To Previous Work

This research was originally inspired by the work of Emanuelson [Emanuelson 1980]. He applied a partial evaluator to compile calls to a pattern matcher where the pattern was known and the target was unknown. We view the research presented in this paper as a natural next step in the partial evaluation of interpreters.

[Futamura 1971] argues that partial evaluation could become a means of compilation. Futamura presents the basic idea of viewing an interpreter as a function which can be specialized. He also proposes that the partial evaluation process itself can be applied to itself to produce a specialization of the partial evaluator which can only specialize a particular interpreter. He argues that such a specialized partial evaluator can be viewed as an ordinary compiler.

[Turchin 1982] and [Turchin 1984] present the concept of a "supercompiler" which closely resembles a partial evaluator. The supercompiler is also used to specialize interpreters, though only toy interpreters for simple languages are presented.

[Komorowski 1981] presents a simple partial evaluator for Prolog. Given a powerful partial evaluator for Prolog and a good Prolog compiler, Prolog can replace the role of Lisp in this paper. Languages could be implemented by writing interpreters in Prolog. [Gallagher 1984] has explored this idea as a means of controlling logic programs. The idea is to write in Prolog interpreters for Prolog-like languages with control annotations. Programs exploiting these control annotations are then compiled via partial evaluation into ordinary Prolog programs. These programs are typically much faster and more difficult to read or write.

As noted in the discussion of the compilation of member, it should in principle be possible to partial evaluate Prolog predicates at the level of their Lisp implementation. It would be interesting to compare a partial evaluator for Prolog with Partial Lisp applied to the LM-Prolog interpreter.

7 Conclusions, Problems, And Future Work

Much remains to be done to complete this research. *Built-in predicates* for the LM-Prolog interpreter need to be written. (Currently the LM-Prolog compiler takes care of them.) The partial evaluator should be tested upon control primitives and the interface to Lisp. In general, more testing needs to be performed. Also, to demonstrate the generality of this approach, Partial Lisp should be applied to another language; perhaps a message passing system would be suitable. At partial evaluation time one can often discover which method will be applied to a message and thereby avoid a run-time search.

More fundamental are the issues involved in controlling the amount of time needed to partial evaluate a predicate and the size of the resulting "compilation" (i.e. the specialized interpreter). Either the system needs to be able to make time versus code size trade-offs wisely or the user needs a good way of advising the system.

Further work on specializing particular uses of Prolog predicates needs to be done. Examples like the specialization of member for ground alists needs to be worked upon. Also the generation of interpreters that can run a class of Prolog predicates is worth exploring. Perhaps an interpreter that can deal only with predicates defined by unit clauses would be worthwhile, or an interpreter that runs only ground goals, or one that can only deal with deterministic goals.

The results to date of partial evaluating the Prolog interpreter are very encouraging. Predicates such as member and concatenate can be sped up by about ten times, the code produced is sufficiently compact, and time needed to perform a specialization is about one or two minutes. More careful engineering of Partial Lisp can probably win a factor of two or three. Newer generation Lisp machines are purportedly two or three times faster. Also, since Partial Lisp is written in LM-Prolog it can be used upon itself to speed itself up. A processing time of 5 to 10 seconds per predicate is still large but considering that this compilation is done without the use of a compiler this seems almost magical.

Acknowledgments

We wish to thank the members of UPMAIL for their helpful comments and advice. We also wish to thank Pär Emanuelson for providing the original direction for this research.

References

Beckman, L., Haraldsson, A., Oskarsson Ö., Sandewall E. "A Partial Evaluator and its use as a Programming Tool", *Artificial Intelligence Journal*, Vol. 7, No. 4, 1976, pp. 319-357.

Carlsson, M. and Kahn, K., "LM-Prolog User Manual", UPMAIL Technical Report 24, Uppsala University, November 1983.

Carlsson, M. "LM-Prolog — the Language and its Implementation", Ph. L. thesis, UPMAIL Uppsala University, forthcoming.

Emanuelson, P., "Performance enhancement in a well-structured pattern matcher through partial evaluation", Linköping Studies in Science and Technology Dissertations, No. 55, Software Systems Research Center, Linköping University, 1980.

Futamura, Y., "Partial Evaluation of Computation Process — an Approach to a Compiler-Compiler", *Systems Computers Controls*, Vol. 2, No. 5, August 1971, pp. 721-728.

Gallagher, J. "Transforming Logic Programs by Specialising Interpreters", Draft of a technical report, Department of Computer Science, Trinity College, University of Dublin, Ireland, April 1984.

Greenblatt R., "The Lisp Machine", MIT AI Lab Working Paper 79, November 1974.

Kahn, K., "A Partial Evaluator of Lisp written in Prolog", *Proceedings of the First Logic Programming Conference*, Marseille, France, September 1982.

Kahn, K. and Carlsson, M., "How to Implement Prolog on a Lisp Machine", in Campbell, J. ed. *Implementations of Prolog*, Ellis Horwood Ltd., Chichester, Great Britain, 1984.

Kahn, K. "A Pure Prolog written in Pure Lisp", Logic Programming Newsletter No. 5, Lisboa, Portugal, Winter 1983/1984.

Kahn, K. "Partial Evaluation, Programming Methodology, and Artificial Intelligence", *The AI Magazine*, Vol. 5, No. 1, Spring 1984.

Komorowski, H.J. "A Specification of an Abstract Prolog Machine and its application to Partial Evaluation", Linköping

Studies in Science and Technology Dissertations, No. 69, Software Systems Research Center, Linköping University, Sweden, 1981.

Moon, D., Stallman R., Weinreb D., "Lisp Machine Manual", MIT AI Laboratory, 1983.

Turchin, V., Nirenberg, R., and Turchin, D. "Experiments with a Supercompiler", ACM Symposium on Lisp and Functional Programming, 1982.

Turchin, V. "The Concept of a Supercompiler", submitted for publication 1984.

Warren, D., "Implementing Prolog — compiling predicate logic programs", Department of Artificial Intelligence, University of Edinburgh, D.A.I. Research Report Nos. 39 and 40, May 1977.

Warren, D. "An Abstract Prolog Instruction Set", SRI International Technical Report 309, October 1983.