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 2050, 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 is 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 vice 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 use 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. LMC-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 partially evaluate the form (append x y) to (cons y) when x is known to be (list t). This process begins by opening the form with the definition of append.
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 nil and list as

\[
\text{null}\ x = (eq\ x\ \text{nil})
\]

\[
\text{list}\ x = (\text{cons}\ x\ \text{nil})
\]

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

\[
(\text{cons} (\text{first} (\text{list} f)) (\text{append} (\text{rest} (\text{list} f)) y))
\]

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 cons 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 re-computed redundantly. For example,

\[
(\text{lambda} (x\ y) (\text{list} x\ x\ y))
\]

\[
(f\ x)
\]

partial evaluates to

\[
(\text{lambda} (x') (\text{list} x\ x' (g\ x)))
\]

Most of the power of the system comes from the capabilities of the handlers for the basic Lisp primitives cond, car, cdr, cons, eq, , , , , , , and , , , , , , which are added when given only partial information. Conditions, 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 nil 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 Redhun 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 [Futs-
mura 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 eval-
uation 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 were not. The abstracted problem and solution are then added to a database. By this means one can build up the equivalent of a handler for the most common uses of a function. For example, partial evaluation 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 the 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 recur, then the system

1. generalizes the problem by replacing forms which it considers uninteresting by variables,
2. makes up a name, \( \phi \), 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 \( \phi \) and note that a recursive call was generated,
4. partial evaluates the problem as generalized in step 1 to \( \xi \),
5. if a recursive call was generated, fills in the body of \( \phi \) with \( \xi \) and returns a call to \( \phi \), otherwise returns \( \xi \).

Partial Lisp has no understanding of iteration primitives such as do 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 do and go.

3 The LM-Prolog Interpreter

Initially, the plan for this project was to write a very pow-
erful partial evaluator for Pure Lisp and to write a Prolog-inter-

```
preter 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 assac. 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 locat. Variables are bound to variables via ZetaLisp’s invisible pointers. Invisible pointers are also used within terms so that deforcercing is done automatically by ZetaLisp’s first (= car) and rest (= cdr).

LM-Prolog is based upon lazy structure copying. Changes at define time are transformed into a template structure. The four types of templates are “ground terms”, “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 microroed 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 comes 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 cons-embedding the body of the clause (i.e. converting the template structure into a term thereby reusing 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 & Carlson 1984], [Carlson & Kahn 1985], and [Carlson forthcoming].

A few minor changes were made to the LM-Prolog interpreter for the convenience of the partial evaluator. The partial evaluator cannot globalize 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 is partial evaluated away so there no run-time cost. The production version of LM-Prolog uses a global trial of changes to implement backtracking. This was left for this experiment since there are few optimizations that can be performed upon compiling.

In some versions of LM-Prolog the functions unify, unif-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 out are handled by slightly more complicated provers.)

\[
\text{[defun p-prover [continuation &rest arguments]}
\text{[try-each clauses-for-p continuation arguments]}
\text{[trail, ;the current trail pointer]}
\text{[make-list p-number-of-arguments])}
\]

Clauses are represented as ZetaLisp flavor instances [Mooi et al. 1985] which contains 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 defun is like ZetaLisp’s defun except that it transfers tail recursion into iteration.
(defun try-each (clauses continuation arguments mark vector)
  ;; tries each clause by unifying its head with
  ;;ARGUMENTS. VECTOR is the local environment.
  ;;MARK is the trail marker.
  (cond (null clauses) nil
        (true clauses (nil (let (templates (cond (first clauses) 'templates))
                      (and (unify-term-with-template
                               arguments (first templates) vector)
                           ;; unification of head and goal succeeded
                           ;; so construct the body and evaluate it
                           (prove-conjunction
                            (construct (rest templates) vector)
                            continuation))))
        (t ;something failed, so undo any bindings made
           (untrail mark)
           (try-each
            (rest 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.

(defn prove-conjunction (predications continuation)
  ;; if the conjunction of PREDICATIONS can be proved,
  ;; invoke CONTINUATION
  (cond (null predications) (invoke continuation)
        (true (let (predication (first predications))
                (definition
                 (current-definition (first predications))))
                (cond ((definition-deterministic-definition
                         ;; goal is deterministic so prove the
                         ;; first goal and try the remainder
                         (:first goal and try the remainder
                          (cond ((apply (definition-proof-definition)
                                   (cons (continuation true)
                                          (rest1 predications))
                                 (prove-conjunction
                                  (rest predicates)
                                  continuation))))
                       (t ;goal is not set, so prove the first goal
                         with continuation to prove the remainder
                         (apply (definition-proof-definition)
                                (cons
con
(prove-conjunction
(rest predicates)
continuation))
(rest1 predications))))))))))

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 sim-
ple interpreted predicates, instead of calling their proven. This
enables prove-conjunction to tail-recursive even for some non-
deterministic predicates.

Unification between terms and templates essentially dis-
patches on the template code:

(defun unify-term-with-template (term template vector)
  ;; unifies TERM with a solid representation of a term
  ;; in the environment of VECTOR
  (let ((type (first template)))
    (cond ((eq type 0) ;template is a ground term
             (unify term (rest template)))
          ((eq type 1) ;first occurrence of variable
             (cond ((eq -1 (rest template))) ;void variable
                    (t ;permanent var—store term in vector
                     (setf (rest (rest template) vector) term)
                     t))
          ((eq type 2) ;repeated occurrence of variable
             (retun term
             (dereference (nth (rest1 template) vector)))))
          ((eq type 3) ;complex template
             (cond (equal 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:

(defun unify (t y) ;unifier of two terms (e-expressions)
  (cond ((eq t y)
             (t))
        ((and (cons x) (cons y))
          ;; both are conses, so unify corresponding parts
          (and (unify (first x) (first y))
               (unify (rest x) (rest y))))
        ((value-cell-p x) (bind-cell x y))
        ((value-cell-p y) (bind-cell y x))
        ((equal x y)) ;may be EQUAL strings or numbers
          t)

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

Construction of terms essentially dispatches on the tem-
plate code. Ground terms are truly structure shared.

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

Macro definition.

(defsubst value-cell (x) (eq (type x) 'locative))
(defsubst prolog-cons (x y)
  (cons (reference x) (reference y)))
(defsubst bind-cell (cell)
  (set-contents cell (reference value)))
(trail cell)
The partial evaluator is not given the definitions of reference, dereference, call and set-contents. These deal with locative 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 ((= 7 x (x)) (in Dec-10 Prolog that would be X = X)). The following prover is generated:

(defun ==-prover (continuation &rest arguments)
  (try-each class 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

(defun ==-prover (continuation x y)
  (try-each class for=)
    continuation
    (list (derefrence x) (derefrence y))
    "trail"
    (make-list 1))

This transformation is justified by the semantics of Zetalisp's direct 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 specialised 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.

(uniﬁ-y-term-with-template
 (list (derefrence x) (derefrence y))
 't (3 (1 0) 3 (2 0) 0) ;template for (x x)
 (list nil))

-partial evaluates to →
  (cond ((uniﬁ (derefrence y) (derefrence x)) t))

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

(uniﬁ-y-term-with-template
  (derefrence x))

"(t 0) ;first occurrence of x
 (list nil))

-partial evaluates to →
  t ;in addition & changes (list nil) to (list (derefrence x))

(uniﬁ-y-term-with-template
  (derefrence y)
  't (2 0) ;second occurrence of y
  (list (derefrence x))

-partial evaluates to →
  (uniﬁ (derefrence y) (derefrence x))

(uniﬁ-y-term-with-template
  nil '(t) ;template for nil
  (list (derefrence x))

-partial evaluates to → (uniﬁ nil nil)

-partial evaluates to → t

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

(construct '() (list (derefrence x)))

-partial evaluates to → nil

(prove-conjunction nil continuation)

-partial evaluates to → (invoke continuation)

4. Partial evaluate the non-existing remaining clauses.

(try-each nil
  continuation
  (list (derefrence x) (derefrence y))
  mark
  (list (derefrence x))

-partial evaluates to → nil

After partial evaluation the prover looks like

(defun ==-prover (continuation x y)
  ;i.e. if X and Y unify then invoke the CONTINUATION
  (cond ((uniﬁ (derefrence y) (derefrence x))
    (invoke continuation))))

The resulting version of the prover is equivalent to the prover produced by LM-Prolog's compiler. With microcode support, the predicate (= 1 1) takes 76 microseconds with = compiled versus 584 interpreted. Without microcode support, it takes 179 microseconds compiled and 1305 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 . x)) (member x (x . ) (member x y))

In Dec-10 Prolog it would look like:

member(x[X...])
member(x[X...]) -> member(x,Y).

The following is the interpreter's prover:
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 dist) 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 sometime mode declarations can significantly improve the runtime 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 predicate (member (key *value* *value* *value* *value*)) where either the system concludes that neither is a symbol, that both are symbols, or that *value* is a list. Part of Lisp currently cannot understand that something in a list of const 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 so follows:

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 programs produced and their effectiveness.

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 (microencoded unly, unstr, cell, etc.). All times are in milliseconds.

<table>
<thead>
<tr>
<th>Microencoded</th>
<th>Without Microcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partial Evaluated</td>
<td>73</td>
</tr>
<tr>
<td>Compiled</td>
<td>103</td>
</tr>
<tr>
<td>Interpreted</td>
<td>523</td>
</tr>
</tbody>
</table>

6 Relation To Previous Work

This research was originally inspired by the work of Emanuelsson [Emanuelsson 1986]. 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 evaluations 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.

[Kemowski 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 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 adapting the system.

Further work on specializing particular uses of Prolog predicates needs to be done. Examples like the specialization of member for ground atoms 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 PartialLisp 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 Per Emaneulson for providing the original direction for this research.

References


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


