Partial Evaluation, Programming Methodology, and Artificial Intelligence

Kenneth M. Kahn

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

Editor's Note: AI workers have claimed for some time that the size and complexity of the program which they attempt to implement forces the field to continuously improve its programming environments. Kahn discusses in this article another such technique, namely "partial evaluation." Partial Evaluators are transformation systems which have knowledge about transformations and the target programming language and transform general functions into ones better tailored to particular situations.

Kahn notes that partial evaluators whilst being another technique for the use with (AI) programming environments, can claim to be a subject for study within the field. Hopefully, partial evaluators will enable more complex AI programs to be written and that the more powerful techniques which thus evolve will enable more powerful partial evaluators to be implemented... — *Derek Sleeman and Jaime Carbonell*

Abstract

This article presents a dual dependency between AI and programming methodologies AI is an important source of ideas and tools for building sophisticated support facilities which make possible certain programming methodologies These advanced programming methodologies in turn can have profound effects upon the methodology of AI research Both of these dependencies are illustrated by the example of a new experimental programming methodology which is based upon partial evaluation Partial evaluation is based upon current AI ideas about reasoning, representation, and control The manner in which AI systems are designed, developed and tested can be significantly improved in the programming is supported by a sufficiently powerful partial evaluator In particular, the process of building levels of interpreters and of intertwining generate and test can be partially automated Finally, speculations about a more direct connection between AI and partial evaluation are presented.

PARTIAL EVALUATION is a relatively new program manipulation technique that is being used to optimize programs, generate compilers from interpreters, generate programs automatically, open-code functions, and efficiently extend languages. 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) where x is known to be (list f) to (cons f y). 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 as (lambda (x) (eq x nil)) and list as (lambda (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. For more details about the operation of partial evaluators see Emanuelson (1980), Beckman *et al* (1976) and Kahn (1982b).

Clearly the availability of a reliable powerful partial evaluator can drastically change the way one programs. First and foremost, one can concentrate on getting the job done and let the partial evaluator take care of the efficiency aspects. This differs significantly from the current situation of programming in a language which has an optimizing compiler because the extent of the optimizations is so much greater. A programmer with a partial evaluator is freer to write simple pure modular programs, test them, and reason about them in that form. It is the partial evaluator that will automatically generate a semi-production version.

To take a simple example, consider the problem of determining in LISP whether there are no elements in common between two lists. One may have available a function intersection which returns a list of elements common to two lists. One is tempted to write (null (intersection x y)). This, however, can lead to much unnecessary computation if the lists have many elements in common. My partial evaluator, "Partial LISP," can transform the call (null intersection x y)) to (null-intersection-1 x y) where null-intersection-1 is a function the system generates based upon LISP definition of intersection. It does no consing and returns nil as soon as an element in common is found.

Briefly, the system begins by creating a new function null-intersection-1 and adds to the LM-PROLOG database a clause stating that any problem similar to the original one should use this function. It begins to create the definition of null-intersection-1 by applying the definition of intersection to x and y. Since nothing is known about them this is the same as opening the definition in the call. Next the system transforms the application of the function null to a conditional (the body of intersection) to a conditional where the function is applied on each conditional branch. The original problem has now been transformed to:

```
(cond ((null x) (null nil))
      ((member (first x) y)
       (null (cons (first x) (intersection (rest x) y))))
      (t (null (intersection (rest x) y))))
```

The first branch can be evaluated to t, the second branch to nil thereby avoiding the recursive call and the consing, and the third branch can be recognized as an instance of the original problem. The definition of null-intersection-1 is completed and shown below:

```
(defun null-intersection-1 (x y)
  (cond ((null x) t)
        ((member (first x) y) nil)
        (t (null-intersection-1 (rest x) y))))
```

A similar example is described in more detail in Kahn (1982b).

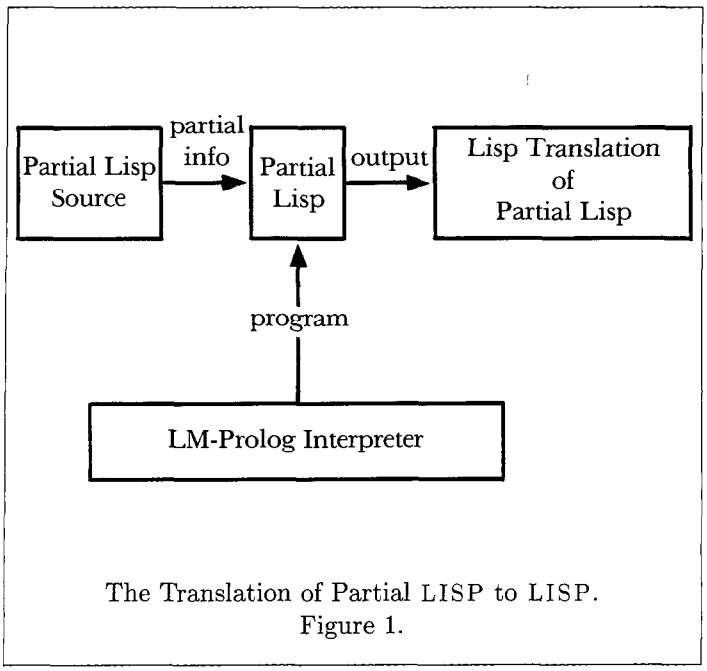
Programming methodology in AI shares much with general programming methodology but differs in significant ways. An AI researcher does not typically understand the problem being programmed very well. An essential aspect of a very common style of doing AI research is to write programs in order to understand something better. Under these conditions one is more concerned with the ease of developing, testing, modifying and evaluating programs than efficiency or correctness.

An AI programmer cannot typically ignore questions of efficiency. Programs often become too slow to test and debug effectively. More fundamentally AI is concerned with models of intelligence which are computationally constrained. A program is not intelligent, regardless of how powerful it is, if it takes it more time than the age of the universe to solve a problem. The complexity of the underlying algorithms of an AI program are significant.

The major difficulty in AI programming is that the combined demands of acceptable efficiency and sophisticated behavior forces AI programmers to write extremely complex programs which are difficult to understand, much less extend or debug. A very important technique for controlling the complexity is to build systems in layers of abstractions. A major difficulty with exploiting the "layers of abstraction" programming technique is that each level is interpreted by the one below and the system becomes to slow. When this happens a programmer typically either mixes layers (e.g. by escaping to LISP) resulting in a faster but more complex and less general program or writes a compiler which takes descriptions from one level down to the next lower level (e.g. the pattern matching compiler in CLISP). Writing a compiler, however is a large job in itself and the requirement that it be kept compatible with the interpreter slows down further developments.

This is a good example of where partial evaluation can help. A powerful partial evaluator can flatten the "layers of abstraction" and optimize the results so that the overhead of interpretation is removed. (Futamura (1971) seems to be the first to have realized this) One can view an interpreter as a program which takes two inputs: a user program and the inputs to that program. A partial evaluator can create specializations of the interpreter for different user programs (with their inputs unknown). These specializations are effectively compilations of the user prorams into the language in which the interpreter is written.

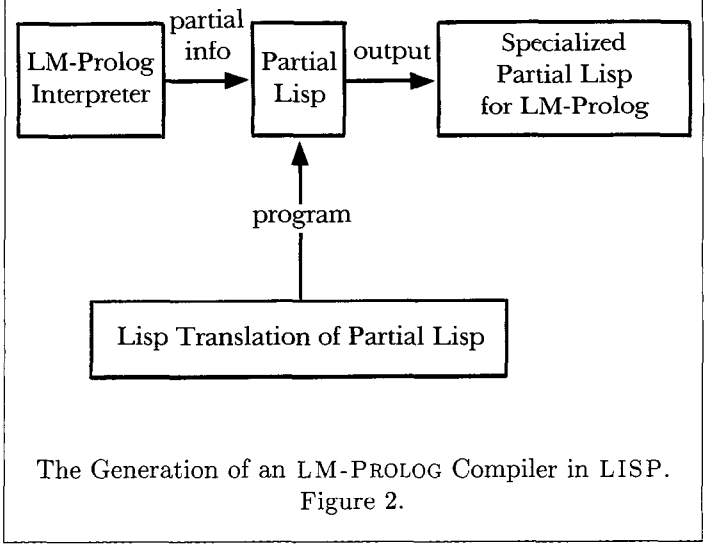
I am currently engaged in an experiment to test this idea upon an interpreter for an AI programming language. A PROLOG interpreter (Clocksin & Mellish 1981) has been written in LISP Machine LISP (Moon *et al* 1983) called LM-PROLOG (Kahn & Carlsson forthcoming). A partial evaluator



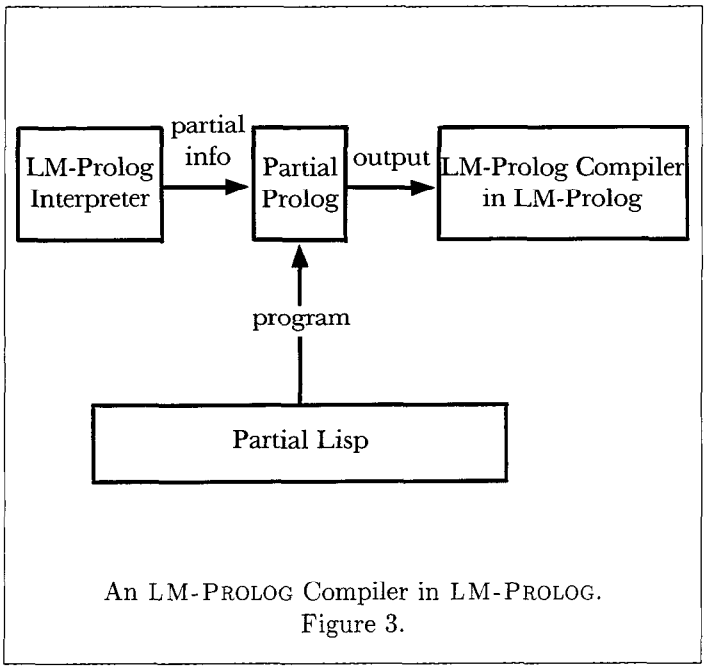
The Translation of Partial LISP to LISP.
Figure 1.

called "Partial LISP" is being written in LM-PROLOG. Partial LISP is being applied first to the primitives of the LM-PROLOG interpreter to bring their implementation down to the LISP level. Then the LM-PROLOG predicates which constitute Partial LISP will be reduced to LISP by partial evaluating the LM-PROLOG interpreter with respect to the partial evaluator.

The result of this process can be applied to itself by partial evaluating Partial LISP with respect to the LM-PROLOG interpeter producing an LM-PROLOG compiler. In other words, we can specialize the partial evaluator where its input is the LM-PROLOG interpreter. The result is a LISP program which translates LM-PROLOG programs to LISP,



The Generation of an LM-PROLOG Compiler in LISP.
Figure 2.



An LM-PROLOG Compiler in LM-PROLOG.
Figure 3.

i.e. an LM-PROLOG compiler. Independently, Carlsson has written a compiler for LM-PROLOG and it will be interesting to compare the code generated by it and the automatically generated compiler (Kahn & Carlsson forthcoming).

Another experiment I am engaged in is the development and application of a partial evaluator for LM-PROLOG called "Partial PROLOG". This partial evaluator (perhaps "partial interpreter" would be a better name) is written in LM-PROLOG and specializes LM-PROLOG programs. Partial PROLOG resembles the PROLOG partial evaluator described in Komorowski (1981) with additional capabilities of automatically handling recursive and "built-in" predicates. I am considering applying Partial PROLOG to the implementation of Partial LISP where the partial input is the LM-PROLOG interpreter. It will be interesting to compare the compiler generated this way with the one generated as illustrated in Figure 2.

Other experiments that are being considered are to partial evaluate languages written in LM-PROLOG (*e.g.* LM-PROLOG versions of Intermission (Kahn 1982a) or Uniform (Kahn 1981)). And perhaps programs written on top of languages could be reducible.

This process of using partial evaluation to flatten layers of interpretation is in progress and difficult to illustrate upon realistic examples. To given an impression of the process, consider the partial evaluation of calls to a toy pattern matcher. Emanuelson (1980) describes the application of a partial evaluator to a "full-fledged" pattern matcher. The toy interpreter is defined as follows.

```
(defun match (pattern subject)
  (cond ((null pattern) (null subject))
        ((eq (first pattern) '-)
             ;;any length segment
         (or (match (rest1 pattern) subject)
             (match pattern (rest1 subject))))
        ((null subject) nil)
        ((eq (first pattern) '&)
             ;;matches any corresponding element
         (match (rest1 pattern) (rest1 subject)))
        ((equal (first subject) (first pattern))
         (match (rest1 pattern) (rest1 subject)))))
```

The call (match pattern xx) where pattern is known to be '(& a b) evaluates to:

```
(cond ((eq xx 'nil) 'nil)
      ((eq (rest1 xx) 'nil) 'nil)
      ((eq (first (rest1 xx)) 'a)
       (cond ((eq (rest1 (rest1 xx)) 'nil) 'nil)
             ((eq (first (rest1 (rest1 xx))) 'b)
              (null (rest1 (rest1 (rest1 xx))))))))
```

Notice that there are no longer any calls to match and the code produced is less compact but faster.

Another example of how partial evaluation can influence AI programming is with respect to the "generate and test" paradigm. Many problems can profitably be broken up into generating possibilities and testing to see whether any of them satisfy some constraints. Frequently, a much more efficient method exists in which the generation and testing are intimately intertwined. For example, consider the problem of sorting a list The problem can be broken up into a generator of permutations of a list and a predicate which decides if a list is ordered. These sub-problems are both conceptually and computationally easier to deal with than the original problem Unfortunately, the corresponding program is absurdly slow. Darlington (1976) applied a program transfomation system which has much in common with partial evaluation to this problem and generated efficient sort programs which intertwine the permutation generation and ordering test. A powerful partial evaluator should be able to do this automatically.

A general phenomenon in AI and computer science is the trade-off between weak but general solutions and strong but specialized ones. A partial evaluator's job is to take general programs and generate more efficient but less general versions. Trying to apply partial evaluation to building AI systems leads one to consider whether AI programming in the future might correspond to programming general but weak methods, providing (or letting the system discover) descriptions of the most common special cases and letting the system generate programs to effectively cover those cases.

AI Techniques in the Implementation of Partial Evaluators

A partial evaluator is very much like an ordinary evaluator (e.g LISP's "eval") except that instead of computing the result of some computation it reasons about what the result must be. When everything is known that is needed for an ordinary evaluator to compute, then a partial evluator computes the same results as an ordinary evaluator (though typically the partial evaluator is much slower). The purpose of partial evaluation is to reason about the result of executing a program even if the information necessary to run the program is incomplete or partial. (Hence the name partial evaluation.) One can view a partial evaluator as a generalized interpreter.

A powerful partial evaluator needs to use various AI techniques such as knowledge bases, self knowledge, dependency maintenance, inference, modeling of change, etc. Dealing with a conditional, for example, requires sophisticated inferences about the consequences of the predicate being true or false and using them in dealing with the different branches of the conditional. Dealing with the recursion and iteration demands some degree of self knowledge and some dependency maintenance to know whether a sub-problem is a proper instance of a super-problem and thus can become a recursive call. Dealing correctly with programs with side-effects is a special case of the frame problem. Situational calculus, circumscription, non-monotonic or dynamic logic or the like is needed to model the changing state of an impure program

Here is an opportunity to explore a rather incestuous relation between partial evaluation and AI. AI programming is used to make a partial evaluator more and more powerful, while the partial evaluator is used in critical ways to ease the task of the AI programming. As the partial evaluator improves, it can be used to make further improvements to itself making it easier to program more improvements, and so on.

This all sounds very promising, but what are the limitations to partial evaluation? Too little research on partial evaluation has been done to answer this question adequately, however too limitations seem intrinsic. One very important optimizing transformation which does not fit within the partial evaluation paradigm is change of representation If the original programs which the partial evaluator optimizes computes with, say, a-lists then the partial evaluator will not consider re-writing it using hash tables. Perhaps a program transformation system could be built which contains both a partial evaluator and a representation shifter.

The other major limitation of partial evaluation is that, unlike program synthesis systems, it has no specification of the problem. It has only a (typically inefficient) program to do the task and this usually over-specifies the task. For example, a LISP program which finds the intersection of two lists returns a list of the elements in a particular order which may be of no relevance to its callers. The current technology behind partial evaluators will perform optimiza-

tions on such a program but will not consider optimizations which might produce a permutation of the output of the original intersection program Goad (1980) has explored a variation of partial evaluation which begins with a program, its specification, and their connections His system can perform optimizations which change the behavior of programs so long as they continue to satisfy the specifications.

Is Partial Evaluation AI?

We have seen how partial evaluation can be the critical tool in a new programming methodology for AI and how it can be a domain for applied AI, but is it AI? Is it a cognitive process like learning from examples, default reasoning, or heuristic search? Or is it only a useful tool in the same class as hash tables or LISP compilers? Abstractly partial evaluation is a process which takes general procedural knowledge and produces effective procedures for special cases Such a process is common in people.

Partial evaluation can be viewed as a kind of learning from examples which is the reverse of the AI paradigm. Instead of going to a general notion from examples, partial evaluation goes from the general notion and examples to more specialized knowledge. Epistemologically partial evaluation does no learning, but from the point of view of generating effective expert knowledge it does. People seem to rarely reason from general principles but instead use specialized knowledge.

Exploring partial evaluation from this AI point of view is an exciting avenue of research What began as a programming tool, has become a domain for applied AI, and is becoming a powerful programming methodology for AI, might become a part of AI.

References

Beckman, L , Haraldsson, A , Oskarsson O , & Sanderwall, E (1976) A partial evaluator and its use as a programming tool, *Artificial Intelligence, 7*, 4, 319-357

Clocksin, W & Mellish, C. (1981) *Programming in Prolog.* Springer-Verlag, Berlin, Heidelberg, New York

Darlington, J (1976) *A synthesis of several sort programs* Research report No 23a, Department of Artificial Intelligence, University of Edinburgh

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

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

Goad, C (1980) *Computational Uses of the Manipulation of Formal Proofs* Doctoral thesis, Stanford University , August

Kahn, K (1981) Uniform - A Language based upon Unification which unifies (much of) LISP, Prolog, and Act1 *IJCAI-7,* August

Kahn, K. (1982a) Intermission - Actors in Prolog In S-A Tarnlund & K Clark (Eds) *Logic Programming* New York, NY: Academic Press.

Kahn, K (1982b) A partial evaluator of LISP written in Prolog *Proceedings of the First Logic Programming Conference,* Marscille France, September

Kahn, K and Carlsson, M. (forthcoming) How to Implement Prolog on a LISP Machine In J Campbell (Ed) *Issues in Prolog Implementations* West Sussex, Great Britain: Ellis Horwood Ltd

Komorowski, H J (1981) A Specification of an Abstract Prolog Machine and its application to Partial Evaluation *Linkoping Studies in Science and Technology Dissertations,* No 69, Software Systems Research Center, Linkoping University, Sweden

Moon, D , Stallman R , & Weinreb D (1983) *LISP Machine Manual.* MIT AI Laboratory