Chapter 4

Artificial Intelligence

The limitations of partial evaluation

Frank van Harmelen
Department of Artificial Intelligence
University of Edinburgh
Edinburgh EH1 1HN
Scotland
frankh%uk.ac.ed.aiva@nss.cs.ucl.ac.uk

1 Introduction

There is widespread agreement in the literature on expert system about the desirability of the separate and explicit representation of control knowledge in reasoning architectures. Such separate and explicit representation of control knowledge, preferably in a declarative format, has the following advantages:

- Ease of development and debugging because of increased modularity.
- Multiple usage of same domain knowledge for different tasks (e.g. problem solving, teaching, etc), by combination with different control knowledge.
- Improved possibilities for explanation of the behaviour of the system, because both control knowledge and domain knowledge can be part of the explanation.
- Fully declarative nature of the domain knowledge, since all procedural aspects can be moved into the control knowledge.

On the basis of these arguments, many workers in AI have implemented architectures to support the explicit and separate representation of control knowledge. One family of architectures that has been particularly successful in this respect are the so called *meta-level architectures.* These systems provide two levels, the *object-level*, used for representing domain knowledge, and the *meta-level*, used for representing control knowledge. In particular, when logic is used as the knowledge representation formalism, a meta-level architecture will provide both an object-level and a meta-level *theory*, with the object-level theory refering to domain knowledge, and with the meta-level theory refering to formulae of the object-level theory, and how to use them. An *interpreter* for both levels will be provided to make *logical inferences* with the expressions from the meta- and object-level theories.

Although a large number of logic-based meta-level systems have been successfully reported in the literature, a serious deficiency that prohibits their use as a practical tool in many situations is the overhead which is incurred by the extra layer of interpretation by the meta-level. The advantage of an explicit meta-level theory is of course (among others) that it is possible to write specialised control regimes for particular applications, which can significantly cut down the search space, and consequently significantly reduce the number of logical inferences needed

to compute the desired object-level result. However, this advantage comes at the price of an increased cost per logical inference made at the object-level. After all, a system with a hardwired control strategy can be optimised for that particular strategy, and current object-level logic programming systems routinely achieve speeds of 100K logical inferences per second, whereas meta-level systems that have their control regime explicitly specified in the meta-level theory will have to perform expensive meta-level inference in order to execute the object-level strategy. Thus, the trade-off is between cutting down the number of logical inferences made at the object-level versus an increase in cost per logical inference due to the extra time spent interpreting the meta-level theory.

It is of course impossible to say anything in general about the reduction in the number of object-level logical inferences, since this depends entirely on the effectiveness of the control strategy as formalised at the meta-level, but simple quantitative models (see for instance [Rose83]) show that in realistic situations there will be a diminishing return on the effectiveness of meta-level effort, and that as a result, there will be a point where the meta-level effort (ie. the increase in cost per logical inference) outweighs the savings made at the object-level (ie. the reduction in number of logical inferences). Where this break-even point lies depends of course entirely on the characteristics of a particular system, but experiments described in [O'Ke88], [Lowe88] and [Owen88a] indicate that the increase in costs for a single object-level inference is often a factor in the range 10-100. This factor was found to be stable across a number of different applications, implemented in different systems.

Although many of the papers in the literature dealing with the use of meta-level interpreters for control issues acknowledge the inefficiency that is inherent in the multiple layers of interpretation, very few of them offer any solutions to this problem. An important section of the limited work on reducing meta-level overhead in recent years has been based on the idea of specialising the general purpose formulation of the meta-level control regime with respect to the particular object-level theory that is being used in the system. Most of the work on this idea is based on the use of *partial evaluation* as an optimisation technique.

Work reported in for example [Venk84], [Take86], [Take85], [Safr86], [Levi88] and [Gall86], is all based on this technique. In this paper we will first describe this technique, and its application to meta-level interpreters in logic-based systems. In the second part of this paper we will explore some of the limitations of this technique in the context of meta-level interpreters for logic-based expert systems, which remain largely undiscussed in the literature.

2 A description of partial evaluation

The main goal of partial evaluation is to perform as much of the computation in a program as possible without depending on any of the input values of the program. The theoretical foundation for partial evaluation is Kleene's S-M-N theorem from recursive function theory [Klee52]. This theorem says that given any computable function f of n variables ($f = f(x_1, \ldots, x_n)$), and k ($k \leq n$) values $a_1, \ldots, a_k$ for $x_1, \ldots, x_k$, we can effectively compute a new function f' such that

$$f'(x_{k+1}, \ldots, x_n) = f(a_1, \ldots, a_k, x_{k+1}, \ldots, x_n)$$

The new function f' is a specialisation of f, and is easier to compute than f for those specific input values. A partial evaluation algorithm can be regarded as the implementation of this theorem, and is, in fact, slightly more general in the context of logic programming: it allows not only that a number of input variables are instantiated to constants, but also that these variables can be partially instantiated to terms that contain nested variables. Furthermore, a partial evaluation algorithm allows k in the above theorem (the number of instantiated input variables), to be 0,

that is, no input to f is specified at all. Even in this case a partial evaluation algorithm is often able to produce a definition of f' which is equivalent to f but more efficient, since all the computations performed by f that are independent of the values of the input variables can be precomputed in f'. Thus, a partial evaluation algorithm takes as its input a function (program) definition, together with a partial specification of the input of the program, and produces a new version of the program that is specialised for the particular input values. The new version of the program may then be less general but more efficient than the original version.

A partial evaluation algorithm works by symbolically evaluating the input program while trying to (i) propagate constant values through the program code, (ii) unfold procedure calls, and (iii) branching out conditional parts of the code. If the language used to express the input program is logic, then the symbolic evaluation of the program becomes the construction of the proof tree corresponding to the execution of the program.

A special case of partial evaluation is when none of the values for the input variables $x_1, \ldots, x_k$ are given (in other words, $k = 0$). In this case, the partial evaluation algorithm cannot do as much optimisation of the input program, and as a result the new program will not be as efficient. However, the new program is no longer only a specialisation of the original program, but indeed equivalent to it. Thus, in this way partial evaluation can be used as a way of reformulating the input program in an equivalent but more efficient way.

As a simple example of partial evaluation, consider the following function in Lisp:

```
(defun assoc (key alist)
       (cond ((null alist) nil)
             ((eq key (caar alist)) (car alist))
             (t (assoc key (cdr alist)))))
```

This function accesses the standard Lisp assoc-list data-structure. If we specify a partial input, such as

```
alist = '((key1 . val1)(key2 . val2))
```

then we can partially evaluate `assoc`, using the following call to the partial evaluator:[1]

```
(peval '(assoc key '((key1 . val1)(key2 . val2))))
```

to return the derived program `assoc'`:

```
(defun assoc' (key)
       (cond ((eq key 'key1) '(key1 . val1))
             (t (cond ((eq key 'key2) '(key2 . val2))
                      (t nil)))))
```

One problem with partial evaluation in general is that the partial evaluator has to handle uninstantiated variables. This is because the input of the source program is only partially specified and some of the variables in the source program will not have a value at partial evaluation time. In most programming languages it is hard to deal with uninstantiated variables, and the partial evaluator has to be very careful about what it evaluates, and what not.

This is exactly the reason why logic programming is especially suited for partial evaluation. Unification is a fundamental computational operation in logic programming, and handling uninstantiated variables in unification is no problem at all. In fact, uninstantiated variables arising

[1] This call to the partial evaluator only specifies half of its input: the partially specified input to the source program. The other half of the input to the partial evaluator (the actual definition of the source program) is assumed to be globally available in the execution environment of this call. This argument could be made explicit in the obvious way.

```
theory(t1,
      [o1(a) & h1(X) => c1(X, a),
       o2(a) & h2(X) => c1(X, a),
       o3(X, Y) => h2(Y),
       o3(b, c),
       o2(a)
      ]).
```

Figure 1: An object-level theory in Prolog

from partially specified input[2] can be treated like any other term. For instance, in the example above, care had to be taken not to further evaluate the eq's and cond's, since the variable key was uninstantiated at partial evaluation time. However, in Prolog, the program assoc:

```
assoc(_, [], []).
assoc(Key, [[Key, Value]|_], Value).
assoc(Key, [_|Alist], Value) :-
        assoc(Key, Alist, Value).
```

plus a call to the partial evaluator:

```
:- peval(assoc(Key, [[key1, val1], [key2, val2]], Val)).
```

partially evaluates into:

```
assoc'(key1, [[key1, val1], [key2, val2]], val1).
assoc'(key2, [[key1, val1], [key2, val2]], val2).
assoc'(_, [[key1, val1], [key2, val2]], []).
```

without having to worry about evaluation at all, since even with an uninstantiated variable Key, the procedure evaluates to the equivalent specialised code, which now contains no further calls to be executed at run time.

This technique of specialising a program with respect to its (partial) input to derive a more efficient version, can also be applied in the special case when the source program is itself the definition of an interpreter (i.e. a meta-level program). This will then produce a version of the meta-level interpreter which is specialised for the particular object-level theory program that was given as input specification.

Example in Prolog: Consider an object-level theory, called `t1`, as in figure 1 and a meta-level interpreter that specifies how to use these clauses, as in figure 2.

This meta-level interpreter assumes that the predicate `object_level_interpreter` generates all possible formulas derivable from the input formula using the available object-level inference rules. A full instantiation of the input to the meta-level interpreter consists of the object-level theory, plus the top goal that should be proved. So, if the above meta-level program is partially evaluated with the arguments:

[2] In the context of logic programming G' is a partial specification of G if G' is subsumed by G. We also say that G' is an *instantiation* of G (notation: $G' \leq_{inst} G$): there exists a substitution θ for variables in G such that θ applied to G gives G': $G' = \theta G$. G' is a *strict instantiation* of G (notation: $G' <_{inst} G$) if there is a non-empty substitution θ for variables in G such that $G' = \theta G$.

```
[1] proof(Goal, Theory) :-
        lookup(Goal, Theory).
[2] proof(Goal, Theory) :-
        object_level_inference(Goal, Theory, New_Goals),
        proof(New_Goals, Theory).

[3] proof([],_).
[4] proof([Goal|Goals], Theory) :-
        proof(Goal, Theory),
        proof(Goals, Theory).
```

Figure 2: A meta-level interpreter in Prolog

```
[1]  proof(o1(a) & h1(X) => c1(X,a), t1).
[2]  proof(o2(a) & h2(X) => c1(X,a), t1).
[3]  proof(o3(X,Y) => h2(Y), t1).
[4]  proof(o3(b,c), t1).
[5]  proof(o2(a), t1).
[6]  proof(h2(c), t1).
[7]  proof(c1(c,a), t1).
[8]  proof(X & Y, t1) :-
         proof(X, t1),
         proof(Y, t1).
[9]  proof([X|Y], t1) :-
         proof(X, t1),
         proof(Y, t1).
[10] proof([], t1).
```

Figure 3: The programs after partial evaluation

```
Goal = c1(X, a)
Theory = t1
```

with the following call to the partial evaluator:

```
:- peval(proof(c1(X, a), t1)).
```

then, assuming that the inference rules Modus Ponens and And Introduction are in the object-level inference rules, the derived version of the meta-level program becomes:

```
proof(c1(c, a), t1).
```

However, we do not want to specify the top goal of the query, since we cannot predict which goal we will want to prove. So, we underspecify the input to the source program (the meta-level interpreter): we leave the query uninstantiated, and only specify the object-level theory that we want to use. In the example above we would call the partial evaluator with

```
:- peval(proof(Goal, t1)).
```

giving us the remarkably transformed source program as shown in figure 3. This program contains the direct results for all the successful proofs that could be performed by the meta-level interpreter, namely:

- clauses [1]-[5] contain all the results derived via clause [1] of the meta-level interpreter (using `lookup`),
- clauses [6]-[7] contain all the results derivable via application of Modus Ponens (via clause [2] of the meta-level interpreter),
- clause [8] contains a precomputed scheme for applying And Introduction (based on clauses [2]-[4] of the meta-interpreter),
- clauses [9] and [10] repeat the code from the meta-interpreter for iterating over conjunctive goals. This code will never be used by the partially evaluated version, since the iteration over conjunctive goals has already been precomputed in clause [8]. The fact that this superfluous code still appears in the partially evaluated version is due to the difference in status of clauses [1]-[2] and [3]-[4] of the meta-level interpreter: clauses [1]-[2] are meant to be called by the user of the meta-level interpreter, whereas clauses [3]-[4] are only meant to be called by the code itself. If this information had been conveyed to the partial evaluator (for instance by introducing a new predicate name, clauses [9]-[10] would not have occurred in the partially evaluated code.

The partially evaluated code from figure 3 is of course much more efficient than the original code from figures 1 and 2: Any of the facts mentioned in clauses [1]-[5] of figure 3 can now be proved in 1 logical inference[3] instead of taking anywhere between 4 and 8 logical inferences (depending on their place in the object-level theory). Similarly, the facts from clauses [6]-[7] can now be proved in 1 logical inference, instead of 18 and 41 logical inferences respectively. The speedup for conjunctive goals is 3 logical inferences per conjunction. These speedups may be quite small as absolute figures, but taken as a proportion of the small amount of inference done by this toy example it amounts to about an order of magnitude speedup.

3 Problems of partial evaluation

Although the above example indicates the power of partial evaluation, there are some serious problems associated with partial evaluation as a tool for reducing meta-level overhead. Experiments with partial evaluation of small Prolog programs that were executed by a simple meta-level interpreter indicated two main problems. The first of these is related to the definition of the object-level program (which is one of the arguments of the meta-level procedure that is instantiated at partial evaluation time), and the second problem is to do with the amount of information that is available to the partial evaluator.

3.1 Changing object-level programs at run time

In partial evaluation, programs are specialised with respect to their (partial) input. This gives a derived program that is specialised with respect to its input, and obviously this specialised program cannot be used to do computations on different inputs.

[3] One logical inference corresponds roughly to one procedure call.

```
proof(X,t1) :-
        lookup(X,t1).
proof(X,t1) :-
        proof([Y=>X,Y],t1).
proof(X&Y,t1) :-
        proof([X,Y],t1).
proof([X|Y],t1) :-
        proof(X,t1),
        proof(Y,t1).
proof([],t1).
```

Figure 4: The programs after restricted partial evaluation

In our specific case, the only part of the input to the source program (the meta-level interpreter) that is specified is the object-level theory. However, this object-level theory is likely to change while the meta-level interpreter is running. We are likely to want to add to the object-level theory during the proof of a particular query, thereby invalidating the optimised version of the meta-level program.

Furukawa and Takeuchi [Take86] describe a solution for the special case when the object-level theory grows monotonically. This involves constructing a version of a partial evaluator which is specialised for the meta-level interpreter plus the current version of the object-level theory, by applying the partial evaluator to itself with the meta-level interpreter and object-level theory as input. When a clause is added to the object-level theory, the specialised version of the partial evaluator can be used to construct both a partially evaluated version of the meta-level interpreter for the increased object-level theory, as well as a new version of the specialised partial evaluator, which is in turn to be used when the next clause is added to the object-level theory. Since this process can be performed incrementally, the overhead of repeated partial evaluation is greatly reduced. However, although this incremental approach might be useful during the development stages of a system, it is doubtful whether the price of repeatedly constructing a new specialised version of both meta-level interpreter and partial evaluator at run time does not cost more than it gains, especially when the object-level theory changes frequently. In the context of the problem of changing object-level theories, Sterling and Beer [Ster86] talk about "open programs", which are programs whose definition is not complete, for instance because input data for the expert system needs to be provided at run time. They do not provide a solution for this problem, since they

> "assume that a goal which fails during partial evaluation time will also fail at run time, that is, we assume that a system to be partially evaluated is closed."

Another solution to the problem of run time changes to the object-level theory would be not to include the object-level theory in the specialisation process. This would leave us with only the object-level rules of inference as input for specialisation of the meta-level interpreter. In terms of the example meta-level interpreter from figure 2 this would mean that we do not supply the object-level theory from figure 1 but that we do supply the definition of the predicate `object_level_interpreter`. This restricted version of the partial evaluation process would not generate a very efficient program like the one in figure 3 but the code in figure 4. Although this code is not as optimal as the code in figure 3 (since a lot of computation is still to be

done at run time), it is still more efficient than the original version from figure 2. The new program will not be invalidated by run time changes, since the object-level theory was not used in the specialisation process (the predicate `lookup` will still be executed at run time, and was not precomputed by the partial evaluator, as it was in figure 3. The rules of inference (which were used in the specialisation process) are not very likely to be the subject of run time changes.

3.2 Lack of static information

An important distinction can be made between so called *static* and *dynamic* information. Static information is information which is part of, or can be derived from, the program code, whereas dynamic information is dependent on the run time environment of the program. For the purposes of partial evaluation we include the values of input variables supplied at partial evaluation time in our definition of static information. For example, in the following `or` statement

```
(or (eq x 'a)(eql y 2))
```

under the partial input specification

```
y = 3
```

both arguments to the call to `eql` are statically available (and therefore so is the result of the call to `eql`), whereas only one argument to the call to `eq` is statically available, since the value of x can only be dynamically determined.

[Beet87] distinguishes three different types of information that can influence the search strategy:

1. Information that is independent of the current problem and the current state of the problem solving process.

2. Information that is dependent on the current problem, but independent of the current state of the problem solving process.

3. Information that is dependent on both the current problem and the current state of the problem solving process.

Since both the second and the third type of control information will only be dynamically available, search strategies that use such information can not be optimised (or only to a limited extent) by using partial evaluation. Only search strategies that are independent of both the input problem and the current state of the problem solving process can make full use of partial evaluation. However, such search strategies are very weak and general, and do not typically play a very important role in expert systems applications. It is notable that all the examples given in the literature on partial evaluation show programs that employ only the first type of information.

In order to analyse the problem of dynamic information in more detail, we can distinguish three techniques that are used by a partial evaluator to produce more efficient code:

1. branching out conditional parts of the code,

2. propagating data structures,

3. opening up intermediate procedure calls (unfolding).

We will argue that each of these techniques is crucially dependent on a large proportion of the information being statically available. If most of the information is only dynamically available, partial evaluation will generate quite poor results.

The first technique deals with *conditional branches in the code.* If the condition for such a branch cannot be evaluated at partial evaluation time, because its value is dependent on dynamic information, the partial evaluator either has to stop its evaluations at this point, or it has to generate code for both branches of the conditional, and leave it to the run time evaluation to determine which of these branches should be taken. Neither of these strategies is very successful: if a partial evaluator has to stop optimising the source code at the first dynamically determined conditional it encounters in the code, the resulting code may be not very different at all from the original code, and hence will not be any more efficient. The other strategy (generating code for all possible branches) is also usually not very attractive, given the high branching rates of most programs. This will result in very bulky output code (possibly exponential in the size of the original code), most of which will not be executed at run time. In the context of Prolog, this will mean a large number of clauses that will have to be tried at run time, even though most of them will fail in most cases.

The second technique (*propagating data structures*) tries to pass on data structures through the code in the program. This passing of data structures can be done both *forward* (for input values), and *backward* (for output values)[4]. Obviously, the forward passing of data structures only works for those parts of the data that have been provided statically as partial input. The backward passing of data structures is typically dependent on the values of the input, and is therefore also blocked if most information is only available dynamically. A special problem occurs with the so called built-in predicates that are provided by the Prolog interpreter. These predicates often depend on the full instantiation of a number of their arguments (e.g. `is`), and can therefore not be executed by the partial evaluator if the argument-values are not available. Other built-in predicates cause side effects that must occur at run time (e.g. `write`), and such predicates must also be suspended by the partial evaluator.

The third technique (*unfolding*) tries to insert code for procedure calls as 'in line code', rather than explicitly calling the procedures at run time. This technique runs into trouble as soon as the source program contains recursive calls. Although a particular recursive program may in practice always terminate at run-time, this is not necessarily the case at partial-evaluation time, due to the lack of static information. In the case of a logic program this means that the proof tree may contain infinite paths for some uninstantiated goals, and the partial evaluation would be non terminating. Since these infinite computations only arise from the lack of information at partial-evaluation time, and do not occur at run time, we will use the phrase *pseudo-infinite* computation. Two different types of pseudo-infinite computation can be distinguished. The first type, *pseudo-infinitely deep* computation, is caused by programs whose recursive clauses always apply (at partial evaluation time), but whose base clauses never apply, due to the lack of static information. This gives rise to a proof tree with infinitely long branches. The second type, *pseudo-infinitely wide* computation, is caused by programs whose recursive clauses always apply, but whose base clauses also apply sometimes. This gives rise to a proof tree with infinitely many finite branches. A mixture of both types of pseudo-infinite computation is of course also possible. Pseudo-infinitely deep computation corresponds to a program that needs an infinite amount of time to compute its first output, and pseudo-infinitely wide computation corresponds to a program that computes an infinite number of outputs (on backtracking, in the case of Prolog).

[4]It is exactly this passing of data structures which makes logic programs so suited for partial evaluation, since both the forward and the backward passing is done automatically by the unification mechanism that is provided by the standard interpreter for the language.

A good example of both problems is the predicate `num-elem` given below, which selects numeric elements from a list:

```
num-elem(X, [X|_]) :- number(X).
num-elem(X, [_|L]) :- num-elem(X,L).
```

If this predicate is partially evaluated with no input specified, then the base case will never apply, and an infinitely deep computation will result. If this predicate is partially evaluated with the first argument bound to a specific number, but the second argument still unbound, then the base case will always apply, but so will the recursive clause, resulting in an infinitely wide computation. (Notice that when this predicate is partially evaluated with the second argument bound, then none of these problems occur).

In certain cases, the occurrence of breadth-infinity does not need to lead to problems during partial evaluation, in particular if it is known at partial evaluation time how many outputs are required of the source program. If it is known that at most n different outputs are needed from the source program, then the partial evaluator can unfold the proof tree of the source program until the base clauses have applied n times. A realistic example of this is where a predicate P is immediately followed by a cut in a Prolog program:

$$\mathtt{Q_1}, \ldots, \mathtt{Q_i}, \mathtt{P}, !, \mathtt{Q_{i+1}}, \ldots, \mathtt{Q_k}$$

or more generally

$$\mathtt{Q_1}, \ldots, \mathtt{Q_i}, \mathtt{P}, \mathtt{Q_{i+1}}, \ldots, \mathtt{Q_j}, !, \mathtt{Q_{j+1}}, \ldots, \mathtt{Q_k}$$

where all the conjuncts $\mathtt{Q_{i+1}}, \ldots, \mathtt{Q_j}$ are known to be deterministic (that is: given an input they compute exactly one output). In such a case $n = 1$, i.e. only 1 output will ever be required from P. This sort of analysis does of course presuppose that the partial evaluation algorithm has knowledge about properties of cut and of determinateness of predicates in the source program.

In the more general case, where such information about n is not known, or for pseudo-infinitely deep programs, it is necessary for a partial evaluation program to select a finite subtree from the infinite proof tree, in order to guarantee termination of the partial evaluation algorithm. Let π be a source program, θ an input substitution to π, $P(\pi, \theta)$ a partial evaluation procedure, and let $\pi(\theta) \downarrow$ mean that π terminates on input θ, then we would at least require P to terminate whenever π would terminate on θ (or on some instantiation of θ). Formally:

$$\forall\pi\forall\theta : (\exists\theta' \leq_{inst} \theta : \pi(\theta') \downarrow) \rightarrow P(\pi, \theta) \downarrow .$$

This can of course always be achieved by trivial means, such as not unfolding recursive predicates at all, or only unfolding them once (as in [Venk84]), or in general only unfolding them to a fixed maximum depth. However, a more sophisticated solution would be to incorporate a stop criterion in the partial evaluation procedure that will tell us whether a branch of the proof tree for $\pi(\theta)$ is infinite. Thus, we need a stop criterion S such that:

$$\forall\pi\forall\theta : (\forall\theta' \leq_{inst} \theta : \pi(\theta') \uparrow) \leftrightarrow S(\pi, \theta). \tag{1}$$

As soon as S becomes true on a branch for $\pi(\theta)$, the partial evaluation procedure should stop. The problem with such a criterion S is that it amounts to solving the halting problem for Prolog, and that therefore it is undecidable. The halting problem for a given language L is to find a predicate H_L that will decided whether an arbitrary program P written in L will halt on an arbitrary input I or not:[5]

$$\forall P\forall I : P(I) \uparrow \leftrightarrow H_L(P, I) \tag{2}$$

[5]The reader should be aware of a possible confusion: the stop criterion S from 1 is true not when $\pi(\theta)$ will stop, but when $\pi(\theta)$ will *not* stop, indicating that the partial evaluator *should be stopped.* In analogy, 2 has been formulated using $P(I) \uparrow$ instead of the usual $P(I) \downarrow$.

One of the fundamental theorems of the theory of computation states that this problem is undecidable for any sufficiently powerful language L. Prolog is certainly sufficiently powerful, since it is Turing complete [Tarn77]. Since having S from 1 would also give us H_{Prolog} from (2), S must also be undecidable. This means that the best we can hope for regarding a stop criterion for partial evaluation is one that is either too strong or too weak. A stop criterion which is too strong will satisfy the $\rightarrow$ direction of 1, but there will be some π_0 and θ_0 such that

$$S(\pi_0, \theta_0) \wedge \exists \theta' \leq_{inst} \theta_0 : \pi(\theta') \downarrow,$$

in other words, S will tell us that π_0 will not terminate on θ_0 (or any instantiation of it), while in fact it would. This would result in stopping the unfolding of the partial evaluation algorithm prematurely, thereby producing suboptimal results. Conversely, a stop criterion which is too weak will satisfy the $\leftarrow$ direction of 1, but there will be some π_0 and θ_0 such that

$$\neg S(\pi_0, \theta_0) \wedge \forall \theta' \leq_{inst} \theta_0 : \pi(\theta') \uparrow,$$

in other words, S will tell us that π_0 will terminate on θ_0 (or some instantiation of it), while in fact it would not. This would result in a non-terminating partial evaluation.

In the practical use of a stop criterion, a partial evaluator would keep a stack of goals that are unfolded during the expansion of the input program. If we call the original goal $G = G_0$, and we describe the stack of unfolded goals by $G_i (0 > i > j)$, then a number of useful termination criteria are:

1. unification: $\exists i < j, \exists \theta : \theta G_j = \theta G_i$. We write $G_j =_{unif} G_i$.
2. instantiation: $\exists i < j, \exists \theta : G_j = \theta G_i$, i.e. $G_j \leq_{inst} G_i$.
3. strict instantiation: $\exists i < j, \exists \theta (non\text{-}empty) : G_j = \theta G_i$, i.e $G_j <_{inst} G_i$.
4. alphabetic variancy: $G_j \leq_{inst} G_i$ and $G_i \leq_{inst} G_j$, i.e. $G_j =_{inst} G_i$ (G_i and G_j are identical up to renaming of variables).

It is not necessary to consider another variation, namely $(G_j >_{inst} G_i)$ where G_i is a strict instantiation of G_j, since any chain of ever more general subgoals (a chain $G_0, \ldots, G_n$ where $G_i <_{inst} G_j$ if $i < j$) will always have a most general goal as its limit, and therefore such a computation must always terminate. However, we can have two different variations of (1), namely unification without occurs-check (1a) and unification with occurs-check (1b). These stop criteria relate to each other as follows:

$$(3 \vee 4) \leftrightarrow 2$$
$$2 \rightarrow 1b \rightarrow 1a$$

Simple examples can show that none of these criteria performs satisfactorily. In particular, none of these criteria deals satisfactorily with pseudo-infinitely wide computations. Consider for example a predicate like:

```
p1(X,Y).
p1(X,Y) :- p1(s(X),Y)
```

which generates on backtracking all terms $s^n(0)$ after the call

```
:- p1(0,Y).
```

None of the above stop criteria is able to prevent a partial evaluator from looping while trying to partially evaluate p1 with the first argument instantiated. The point here is not that we would expect a great optimisation from the partial evaluation (it is not clear what such optimisation could possibly be), but rather that the presence of a predicate like p1 in any code makes the partial evaluation non-terminating. An example that illustrates the difference between some of the termination criteria is partially evaluating the above predicate p1 with no input specified. In this case stop criterion 3 is too weak, and lets the partial evaluator loop infinitely, while criteria 4 (and by implication 2, 1a and 1b) properly halt the partial evaluator and reproduce the original code. However, the roles of 3 and 4 swap over on the predicate:

```
p2(0).
p2(s(X)) :- p2(X).
```

which will succeed on any input of the form $s^n(0)$. Partially evaluating p2 with no input specified will be properly stopped by 3 (and by implication also by 2, 1a and 1b), but not by 4 which will loop forever.

For a more realistic example, we can turn to the example that was used in the previous section to describe the use of partial evaluation for meta-level interpreters, shown in figures 1, 2 and 3. The partially evaluated code in figure 3 was computed from the code in figures 1 and 2 using stop criterion 3 (strict instantiation). Had we used the stronger stop criterion 2 (non-strict instantiation), the partial evaluator would have produced the same code in as figure 3 but with clauses [6]-[7] replaced by the following clause:

```
[6a] proof(X, t1) :-
          proof([Y => X, Y], t1).
```

This new clause represents a precomputed version of Modus Ponens, but the partial evaluator has stopped short of actually applying this rule, as in in figure 3, due to the stronger stop criterion. As a result, this new code is not as efficient as the code from figure 3. An advantage of the stronger stop criterion is that it takes significantly less time to execute the partial evaluator (the difference between the execution times of the partial evaluator with stop criteria 2 and 3 is more than a factor 100).

A different situation occurred while computing the code for figure 4. Fewer bindings for variables were known for that partial evaluation, since the object-level theory was not included in the input specification. As a result, the stronger stop criterion 2 had to be used to produce the result in figure 4. Any weaker stop criterion would result in either a non-terminating partial evaluation, or in code with many spurious branches.

3.3 Summary of problems

Summarising, we can say that although in principle a powerful technique, partial evaluation is rather restricted in its use for optimising meta-level interpreters for two reasons.

- Firstly, if the object-level theory is going to be changed at run time, at least part of the object-level theory cannot be included as input to the partial evaluation algorithm, thereby negatively affecting the optimisations achieved by partial evaluation.
- Secondly, if most of the information in a program is only dynamically available (i.e. at run time), partial evaluation suffers from the following disadvantages:
  - If the source code contains conditional expressions, then a partial evaluator will either have to stop the optimisation process at that point, or produce very bulky code.

- Data structures cannot be propagated throughout the code.
- If the source code contains recursive procedures, then, unless specific termination criteria are programmed for particular predicates, a partial evaluator will either produce suboptimal code, or termination of the partial evaluator is no longer guaranteed.

It has to be stressed that although the above discussion quotes examples in Prolog, the problems are not due to this choice, and are fundamental to the concept of partial evaluation.

4 Heuristic guidance to partial evaluation

Although the problems discussed above seriously limit the applicability of partial evaluation as a tool for reducing meta-level overhead, a number of heuristics solution can be found to alleviate the problems to a certain extent. The heuristics discussed below are all based on the idea that we will try to build a specific partial evaluator for a particular application, rather than a general, application independent one. More precisely, we can maintain the general framework for a partial evaluator as discussed above, but identify specific places in the algorithm where a user can tune the algorithm to suit a particular application. In particular, we will point out a number of places in the partial evaluation algorithm where we can insert specific knowledge about the behaviour of components of the program to be evaluated. In our case these components will be meta-level predicates, ie. predicates occurring in the meta-level interpreter.

An obvious candidate for removing over-generality is the stop criterion used to determine when to stop unfolding recursive predicates. This criterion can never be correct in the general case (since such a criterion would solve the halting problem for Prolog programs, and hence for Turing machines). As a result, any general criterion is either going to be too weak (i.e. not halting on some infinite recursion), or too strong (i.e. halting too early on some finite recursion). However, if we know the intended meaning of particular parts of a program we can construct specialised termination criteria that are just right for these particular procedures. As example, consider the definition of `member/2`:

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

We know that this predicate is guaranteed to terminate as long as the length of the second argument is decreasing. A good specialised stop criterion for this predicate would therefore be:

$$\exists i,j : i > j,\ member_i(X, L_i) \& member_j(X, L_j) \wedge \parallel L_i \parallel \geq \parallel L_j \parallel,$$

where $member_i$ represents the i-th call to `member`. This stop criterion would successfully unfold all calls to `member/2` where the second argument is instantiated, but will not loop on those calls where the second argument is uninstantiated (due to a lack of static information). This corresponds to the notion that `member/2` will be used to test membership of a given list, and not to generate all possible lists containing a certain element. This criterion will even work for partially instantiated second arguments. A call to `member/2` like

```
:- member(X, [1,2|L]).
```

will properly partially evaluate to:

```
member(1, [1,2|L]).
member(2, [1,2|L]).
member(X, [1,2,X|_]).
member(X, [1,2,_|T]) :- member(X, T).
```

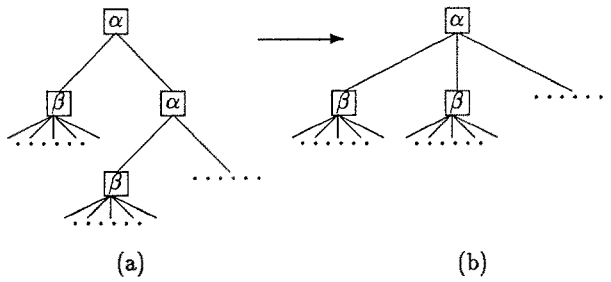

Figure 5: Heuristically limited partial evaluation

The partial evaluation has generated all possible results based on the available static information, while stopping short of looping on the uninstantiated part of the input.

The above halting criterion is based on the intended meaning and use of the predicate `member/2`, and cannot be generally used, since it would again be either too strong or too weak for certain predicates. Consider for instance the predicate `nlist/2` which generates a list of length n:

```
nlist(0, []).
nlist(s(X), [_|T]) :- nlist(X, T).
```

(We use terms $s^n(0)$ for representing the number n to avoid problems with the built in arithmetical predicates. More about this below). This predicate should not be stopped when its second argument is increasing in length, as with `member/2`, but rather when its first argument is increasing in depth. Such metrics should be devised where possible for predicates used in a meta-level interpreter, and the stop criterion should be specialised for these cases.

A second heuristic that we can inject in the partial evaluation algorithm is a special treatment for certain predicates which are known to be easy to compute at run time, but possibly hard or impossible to compute with only static information. This idea is based on the notion of an *operational predicate* as introduced in the explanation-based generalisation algorithm [Mitc86] which turns out to be closely related to the partial evaluation algorithm [Harm88]. The partial evaluation algorithm should stop when encountering such an operational predicate (which is declared as such beforehand), no matter what amount of precomputation could potentially be done using the definition of such a predicate. For instance, it is possible that the definition of such a predicate has a very high branching rate, leading to an explosion of the size of the code generated by partial evaluation, while only one of the many branches would be chosen and computed at low cost at run time, pruning all the other branches. In such a case it is better not to generate the highly branched search space explicitly at partial evaluation time, but to leave it for run time computation.

An example of this specialised treatment of the partial evaluation algorithm for certain predicates is the standard logic programming technique where a predicate, when applied to a list, unpacks the list into its elements, and then takes a specific action for each of the elements in the list. If these specific actions have a very high branching rate, a good strategy for the partial evaluator is to precompute the process of unfolding the list into its elements (a deterministic operation), but to stop short of partially evaluating the actions taken for the individual elements. These actions will have to be performed at run time, when extra available dynamic information will possibly cut down the branching rate. Graphically, this optimisation process can be depicted

as in figure 5. Figure 5a shows the search tree of original code before partial evaluation, with the α nodes doing the unfolding of the list, and the β nodes doing the highly non-deterministic actions for the individual nodes. If we designate β to be an operational predicate, the limited partial evaluation described above will produce code that has a search space as in figure 5b showing that the limited partial evaluation still optimises the search space, but does not get bogged down in the explosive parts of it.

Another heuristic to optimise partial evaluation is *mixed computation.* This involves declaring certain meta-level predicates to be executable at partial evaluation time. If the partial evaluator comes across such a predicate during unfolding, it does not unfold that predicate using its ordinary unfolding strategy, but rather it calls the hardwired interpreter that would normally execute the meta-level code (i.e. in our examples the Prolog interpreter) to execute the particular predicate. The resulting variable bindings are then taken into account during the rest of the unfolding process, but the predicate itself can be removed from the code.

A final heuristic to be embodied in the partial evaluator concerns the evaluable predicates, as typically built into a Prolog system. These predicates can be divided into three types, and for each of the types a different partial evaluation strategy should be used.

- The first type of evaluable predicates are those that perform *side-effects* (e.g. input-output). These predicates can never be performed at partial evaluation time, and must always be postponed until run time.

- The second type of predicates are those that can be partially evaluated if certain conditions hold. These conditions are specific for each particular predicate. For instance, the predicate `var/1` (which tests if its argument is a variable), can be partially evaluated (to `false`, pruning branches from the code) if its argument is not a variable. The reason for this is of course that if the argument is not a variable at partial evaluation time, it will never become a variable at run time, since variables only get more instantiated, not less. On the other hand, if the predicate `var/1` succeeds at partial evaluation time, it must remain in the code, since its argument might or might not have become instantiated at run time. Another example is the predicate `==/2` (testing if its arguments are the same Prolog object). This predicate can be evaluated (to `true`, that is: removed), if it succeeds at partial evaluation time. The argument here is that if it succeeds at partial evaluation time, it will also succeed at run time (since objects that are the same can never become different again), but when the arguments are different at partial evaluation time the predicate should remain in the code, since the objects might or might not have become the same at run time[6]. A third example of this category is the predicate `functor/3` (which computes functor and arity of a Prolog term). This predicate can be partially evaluated if either the first or both the second and third arguments are (at least partially) instantiated. Further criteria could be provided for a number of other built in Prolog predicates.

- The final class of evaluable predicates are those which can be either fully evaluated (if they are fully instantiated at partial evaluation time), or translated into a number of simplified constraints. A good example of this class of predicates are the arithmetic predicates. Obviously, a goal like `X is 5+4` can be fully computed at partial evaluation time, as well as goals like `9 is X+4` and `X>10, X<5` (although a somewhat more sophisticated algorithm is required). In general a goal like `X is Y` ⟨op⟩ `Z`, where ⟨op⟩ is any of the functions `+`, `-` and `*` can be fully computed at partial evaluation time if at least 2 out of the 3 arguments are

[6]This criterion for `==/2` is quite different from that used in the partial evaluator described in [Prie87], where it is incorrectly treated the same as `=/2` (unification), which can always be performed at partial evaluation time, unlike `==/2`.

instantiated. Some types of calls cannot be fully computed at partial evaluation time, but can be transformed into simplified conditions, for instance `X<10, X<11` can be reduced to `X <10`, and the integer division `4 is X/3` can be transformed into `X>11, X<15`. The partial evaluator incorporated in the PROLEARN system [Prie87] implements these heuristics.

4.1 Summary of heuristics

The partial evaluator can be tailored to specific applications by

- incorporating specialised stop criteria for certain recursive predicates.
- using knowledge about operational predicates to stop the partial evaluation process
- using knowledge about the branching factor of certain predicates to stop the partial evaluation process
- allowing execution during partial evaluation, the so called mixed computation
- using specialised knowledge to deal with evaluable predicates, dividing them in three ways:
  - predicates that can never be partially evaluated (predicates with side effects).
  - predicates that can only be evaluated if certain conditions hold
  - predicates that can be transformed into simpler constraints.

5 Related work in the literature

Much related work has been done in the last few years on partial evaluation and its application to meta-programming (e.g. [Bjor87] and [Lloy88]). Some of these papers analyse and discuss the limitations of partial evaluation in a similar way as we have done in this paper, and we will discuss two of these papers in particular.

A well known problem with the use of negation in logic programming is that it is only sound when applied to fully instantiated goals. If applied to partially uninstantiated subgoals, the negation is said to "flounder" [Lloy84], and produces unsound results. This problem is particularly urgent during partial evaluation. A partial evaluation algorithm cannot unfold a negated subgoal if it is not fully instantiated. However, due to the lack of dynamic information at partial evaluation time, many negated subgoals will not be fully instantiated, thus hampering the performance of partial evaluation. As a result, the partial evaluation algorithm given in [Lloy87] is restricted to either evaluate negated subgoals completely (if they are fully instantiated), or not at all otherwise. To solve this problem, [Chan88] gives two techniques for dealing with negated subgoals. This solution is based on two separate techniques for eliminating negation from a program, or at least splitting up into smaller pieces, so that the partial evaluation algorithm can optimise larger parts of the source program.

A second paper that explores the limitations of partial evaluation is [Owen88b]. Owen applied partial evaluation to a number of meta-interpreters that were developed for a particular application, and compared the results of this with hand-coded optimisations of the same set of interpreters. This careful analysis revealed many problems with the practical use of partial evaluation, some of which have also been discussed in this paper:

- Partial evaluation should not always suspend built-in meta-logical predicates (see section 4).

- Partial evaluation causes significant fruitless branches in the object-level program (see section 3.2).
- Partial evaluation systems always suspend the execution of Prolog's cut, even when this is not necessary.

In order to deal with these problems, Owen proposes a number of enhancements to the partial evaluation algorithm. The most significant of these is what he calls a *folding transformation*, which folds a sequence of conjuncts into a new, uniquely named procedure. This is the opposite of the unfolding operation described in section 2. The main goal of this extra operation is to control the branching rate of the code produced by partial evaluation. However, the introduction of this new operation makes a partial evaluation algorithm non-deterministic (the algorithm will have to choose between different possible operations at each step), whereas this was not the case before. This introduces a search component in the partial evaluation procedure that was not present without the folding operation. Further extensions that Owen proposes to the partial evaluation algorithm are the merging of clauses with identical heads, or with heads that only differ in positions which contain local variables, and rules that allow the treatment of cuts in certain under certain conditions at partial evaluation time. Unfortunately, even with these an many other special purpose extensions to his partial evaluation algorithm, Owen found the results of partial evaluation on his meta-level interpreters suboptimal, and it would require open ended theorem proving and consistency checking to achieve the same results as his hand-code optimisations.

References

[Beet87] M. Beetz. *Specifying Meta-Level Architectures for Rule-Based Systems.* Technical Report SEKI No. SR-87-06 (Diploma Thesis), Universität Kaiserslautern, Fachbereich Informatik, Kaiserlautern, 1987.

[Bjor87] D. Bjorner, A.P. Ershov, and N.D. Jones, editors. *Workshop on Partial Evaluation and Mixed Computation*, Avernaes, Denmark, October 1987.

[Chan88] D. Chan and M. Wallace. A treatment of negation during partial evaluation. In J. Lloyd, editor, *Proceedings of the Meta'88 Workshop on meta-programming in logic programming*, pages 227–240, Bristol, June 1988.

[Gall86] J. Gallagher. Transforming logic programs by specialising interpreters. In *Proceedings of the Seventh European Conference on Artificial Intelligence, ECAI '86*, pages 109–122, Brighton, July 1986.

[Harm88] F. van Harmelen and A. Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence Journal*, 30(3):401–412, October 1988.

[Klee52] S. Kleene. *Introduction to Metamathematics.* Van Nostrand, New York, 1952.

[Levi88] G. Levi. Object level reflection of inference rules by partial evaluation. In P. Maes and D. Nardi, editors, *Meta-level architectures and reflection*, North Holland Publishers, 1988.

[Lloy84] J. Lloyd. *Foundations of Logic Programming. Symbolic Computation Series*, Springer Verlag, 1984.

[Lloy87] J.W. Lloyd and J.C. Shepherdson. *Partial evaluation in logic programming.* Technical Report CS-87-09, Department of Computer Science, University of Bristol, 1987.

[Lloy88] J. Lloyd, editor. *Proceedings of the Meta'88 Workshop on meta-programming in logic programming*, Bristol, June 1988.

[Lowe88] H. Lowe. *Empirical Evaluation of Meta-Level Interpreters.* Master's thesis, Department of Artificial Intelligence, University of Edinburgh, 1988.

[Mitc86] T.M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli. Explanation-based generalization: a unifying view. *Machine Learning*, 1(1):47–80, 1986.

[O'Ke88] R. O'Keefe. Practical prolog for real programmers. August 1988. Tutorial No. 8 of the Fifth Internation Conference and Symposium on Logic Programming, Seattle.

[Owen88a] S. Owen. *The development of explicit interpreters and transformers to control control reasoning about protein topology.* Technical Memo HPL-ISC-TM-88-015, Hewlett-Packard Laboratories Bristol Research Centre, 1988.

[Owen88b] S. Owen. Issues in the partial evaluation of meta-interpreters. In J. Lloyd, editor, *Proceedings of the Meta'88 Workshop on meta-programming in logic programming*, pages 241–254, Bristol, June 1988.

[Prie87] A.E. Priedites and J. Mostow. PROLEARN: towards a Prolog interpreter that learns. In *Proceedings of AAAI-87*, American Association for Artificial Intelligence, Seattle, Washington, July 1987.

[Rose83] J.S. Rosenschein and V. Singh. *The Utility of Meta-level Effort.* Technical Report No. HPP-83-20, Stanford Heuristic Programming Project, March 1983.

[Safr86] S. Safra and E. Shapiro. *Meta Interpreters for Real.* Technical Report No. CS86-11, Department of Computer Science, The Weizmann Institute of Science, May 1986.

[Ster86] L. Sterling and R.D. Beer. Incremental flavor-mixing of meta-interpreters for expert system construction. In *Proceedings of the 3rd Symposium on Logic Programming*, pages 20–27, Salt Lake City, Utah, September 1986.

[Take85] T. Takewaki, A. Takeuchi, S. Kunifuji, and K. Furukawa. *Application of Partial Evaluation to the Algebraic Manipulation System and its Evaluation.* Technical Report TR-148, Tokyo, ICOT Research Centre, December 1985.

[Take86] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta programming. In *Proceedings of IFIPS '86*, Dublin, 1986.

[Tarn77] S.Å. Tarnlund. Horn clause computability. *BIT*, 17:215–226, 1977.

[Venk84] R. Venken. A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query-optimisation. In *Proceedings of the Sixth European Conference on Artificial Intelligence, ECAI '84*, pages 91–100, Pisa, September 1984.