ICOT Technical Report: TR-361

TR-361

Competitive Partial Evaluation —Some Remaining Problems of Partial Evaluation—

by
A. Takeuchi & H.Fujita

March, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

COMPETITIVE PARTIAL EVALUATION

— Some Remaining Problems of Partial Evaluation —

Akikazu Takeuchi† *and Hiroshi Fujita*‡

† Central Research Laboratory

Mitsubishi Electric Corp.

8–1–1, Tsukaguchi Honmachi, Amagasaki, Hyogo 661 Japan

‡ ICOT Research Center

Institute for New Generation Computer Technology

1–4–28, Mita, Minato-ku, Tokyo 108 Japan

Abstract: This paper discusses the remaining problems of partial evaluation, comparing competitive partial evaluation to expert programmers who are able to improve programs very effectively. Two case studies of medium size examples are presented to show how to obtain maximal speedup and global control of partial evaluation respectively. These studies are expected to suggest the direction of future research towards everyday-use partial evaluation.

Keywords: Partial Evaluation, Logic Programming, Meta Programming, Optimisation.

1. Introduction

We belong to a growing group of scientists working towards a situation where partial evaluation enables average programmers to exhibit an ability equal to

those of expert programmers. The approach is based on a fine combination of partial evaluation and meta programming.

Meta programming is a strong paradigm for software development, because of its conceptual clarity. It has been said that meta programs are inefficient because of layers of interpretation. However, it is now known that the problem can be solved by partial evaluation of meta programs with respect to object programs[1),2)]. We are in a position where it is natural to expect partial evaluation to be a competitive tool against expert programmers and to replace skilled programming.

However, these statements are somewhat optimistic. In some cases, partial evaluation succeeds in deriving programs as efficient as those written by expert programmers, but usually it needs human interaction, that is, the assistance of experts in partial evaluation. This implies that we need partial evaluation expertise instead of programming expertise. Recently, several researchers have tried to automate partial evaluation[3)]. However, there remain some questions to be solved to make partial evaluation a practical tool.

The first question is on the degree of speedup obtainable by partial evaluation. There are many examples where expert programmers improve programs by some order of magnitude, utilising specificity inherent in a given problem very effectively. Can a partial evaluator, which is concerned mainly with specialisation obtain the same degree of speedup as that obtained by expert programmers? If it cannot, what enhancement for partial evaluation is needed?

The second question is on global control of partial evaluation. A more concrete statement is:

Is the result of partial evaluation guaranteed to be the most efficient or the best program in some sense?

Usually, the result of partial evaluation is given to users in an ad hoc manner. Its quality depends on the strategy embedded in the partial evaluator and the quality of human interaction if it is semi-automatic. A competitive partial evaluator needs to be automated so that any programmer can utilise it easily. It is also necessary for such systems to assure the quality of a resultant program. Otherwise, a programmer is in danger of obtaining many useless programs. To provide such assurance, it is essential to introduce measures which enable the comparison of two programs. These measures should be reflected in the global control strategy of partial evaluation.

These are fundamental problems independent of programming languages and applications. This paper discusses these problems using Prolog. The first problem is discussed in Section 2, where it is assumed that it is difficult to obtain maximal speedup for a given problem by simple partial evaluation. The least enhancement which provides the expected improvement is investigated. Section 3 is devoted to the discussion of the second problem.

2. How to Obtain Maximal Speedup

For a Prolog program, partial information is usually given as partial instantiation of arguments in a procedure call or assertions of facts. By unfolding the instantiated calls or those calling the assertions, the substitutions are propagated up or down through shared variables. Some of the instantiated calls may be to-

tally solved, others may be made residual calls which are suspended to be opened up.

However, the improvement of the efficiency of the program obtained by partial evaluation, which is derived by only substitution propagation and procedure call reduction (unfolding), seems to be limited to some degree in many cases where further improvement is possible by utilising the specificity of known data, facts and the strucure of the subject program. What can we do to achieve maximal speedup in such circumstances by partial evaluation as the core technique? To explore the solution to this problem, a case study is presented in the following subsections.

2.1. Production System – A Case Study

Production system is widely used in state of the art expert system shells. It is an inference system for a set of forward chaining rules using an iterative algorithm. The system holds a rule memory (RM), which stores a set of rules, and a working memory (WM), which stores elements that are added to or removed from it dynamically during an execution. Its execution algorithm is simply an iteration of the following three steps:

(1) Match the current WM with the LHS of the rules in RM.

(2) Select one of the matched rules *(conflict resolution)*.

(3) Fire the RHS of the selected rule.

It is implemented easily in Prolog as in Fig.1, and is called *production engine*.

`exec` is the main loop of the engine, which moves an initial state of *WM*, `WM0`, to its final state, `WMn`, iterating `one_cycle`. `one_cycle` performs a single iteration of the three steps described above. `match` collects all the rules matched with *WM* using the `setof`* primitive, obtaining the *conflict set*, `CS`. `match1` picks up a rule from *RM* and passes it to `match2`, which in turn checks all the patterns on the LHS of the rule using `match3`. `match3` performs case analysis for a pattern on the LHS, which is annotated with + if it is expected to exist in *WM*, or - if it is expected not to exist in *WM*. `match4` succeeds if pattern `X` matches an element in *WM*, otherwise it fails. The detail of `select` is omitted; however, it is assumed that its complexity does not depend on the size of *RM*. `fire` executes all the commands on the RHS by iterating `fire1`. `fire1` despatches commands according to the annotation on element `X`, + when `X` will be added to `WM`, - when `X` will be removed from `WM`. The details of `add` and `remove` as well as `match4` are omitted, for they are dependent on the data structure of `WM`, which is irrelevant to the following discussion.

An execution of the naive implementation of the engine is very inefficient. The inefficiency is caused mainly by `match` procedure, which has two sources of redundancy. One is that it scans the whole *RM*, checking all patterns on the LHS of rules for each iteration (`match1`), without taking acount the fact that there may be common patterns. The other is that it scans the whole *WM*, checking all its elements against the LHS of rules for each iteration (`match4`), without taking acount the fact that there may be some elements whose successful match

* `setof(A,P,S)` computes all solutions for `A` satisfying condition `P` with the resultant list of solutions `S`, if at least one solution exists. Otherwise, it fails.

is unchanged since the previous iteration. The order of computation in the `match` procedure is $O(P \times W)$, where P is the sum of the number of all the LHS patterns in the rule set, RM, ie.:

$$P = \sum_{RM} | \, LHS \, |$$

and W is the number of elements in WM. Usually, P is about linear with respect to the number of rules, $R = \ | \, RM \, |$, and the average number of LHS patterns in a rule is given as $\overline{p} = \overline{| \, LHS \, |} = P/R$, which is very likely to be bounded by some constant, λ.

2.2. Inferential Ability

When a rule set is given, RM is fixed during an execution of the production engine. Therefore, the engine can be specialised with respect to the rule set by partial evaluation. For instance, suppose that the following rules are given.

```
rule([+expr(X,Y+Z),+expr(Z,0)] => [-expr(X,Y+Z),+expr(X,Y)]).
rule([+expr(X,Y*Z),+expr(Z,0)] => [-expr(X,Y*Z),+expr(X,0)]).
```

First, the `match1` procedure is specialised as:

```
match1(([+expr(X,Y+Z),+expr(Z,0)] => [-expr(X,Y+Z),+expr(X,Y)]),
                                   WM) :-
   match4(expr(X,Y+Z),             WM),
                match4(expr(Z,0), WM).

match1(([+expr(X,Y*Z),+expr(Z,0)] => [-expr(X,Y*Z),+expr(X,0)]),
                                   WM) :-
   match4(expr(X,Y*Z),             WM),
                match4(expr(Z,0), WM).
```

Since `WM` is unknown until execution, `match4` is not evaluated any more.

Experiences in partial evaluation shows the difficulty in fully utilising information that is partially known, because, there often are many obstacles which stop the information flow, and make partial evaluation ineffective. For instance, suppose that $(p(X,Y), q(Y,Z))$ is being partially evaluated and X is partially instantiated at this point. Then, p will possibly be partially evaluated without instantiating Y at all, and there may be no chance of q being partially evaluated. However, more can be done. If a domain or type of Y is infered from the given X, q may be partially evaluated further. This is possible by introducing certain inferential capabilities into partial evaluation.

Now, in our case, although `select` cannot propagate the information from `match` to `fire` through the `CS` and `RHS`, the enhanced partial evaluator can infer the possible form of `RHS` which will be output from `select`, ie. any RHS of rules in RM. Thus, `fire` is specialised as:

```
fire(  [-expr(X,Y+Z),+expr(X,Y)],WM0,    WM1]) :-
  remove(expr(X,Y+Z),            WM0,WMi),
                 add(expr(X,Y),     WMi,WM1).

fire(  [-expr(X,Y+Z),+expr(X,0)],WM0,    WM1]) :-
  remove(expr(X,Y*Z),            WM0,WMi),
                 add(expr(X,0),     WMi,WM1).
```

The specialised `match1` and `fire` procedure will have as many clauses as the given rules ($R = \mid RM \mid$) respectively. This will be a reasonable number even if R is ten thousand or more.

Thus, `match1` and `fire` are specialised with respect to the rules in a reasonable size, and overhead of interpretation of each rule is removed by enhanced

partial evaluation. However, P, the number of LHS patterns tested in the `match` procedure, is not reduced at all, and the speedup thus achieved is only $c \times P$, where c is the constant which is determined by units of computations including head unification and opening up of a procedure call. Accordingly, the complexity has been improved now, from $O(\tau \times P \times W)$ to $O((\tau - c) \times P \times W)$, where τ is the unit time to process one pattern on the LHS, and $\tau > c$. That is, the complexity is still linear with respect to the number of patterns, P, which in turn is linear wrt. the number of rules, $R(= P/\overline{p})$.

2.3. Eliminating Redundant Calls by Factoring

A possible source of special speedup in the production engine is the redundancy of re-execution of common pattern matching among different rules. In our example, `match4(expr(Z,0),WM)` will be executed twice in a cycle, once for the `plus_0` rule, and once for the `times_0` rule. This common call can be factored out as in the following code:

```
match1(Rule,WM) :- match4(expr(Z,0),WM), match1_1(Rule,Z,WM).

match1_1(([+expr(X,Y+Z), +expr(Z,0)] => [-expr(X,Y+Z),+expr(X,Y)]),
                                Z,WM) :-
     match4(expr(X,Y+Z),          WM).

match1_1(([+expr(X,Y*Z), +expr(Z,0)] => [-expr(X,Y*Z),+expr(X,0)]),
                                Z,WM) :-
     match4(expr(X,Y*Z),          WM).
```

Note that an extra argument, `Z`, is added to make the split patterns of a LHS in a rule consistent with respect to the shared variable, `Z`.

Now, suppose that goal P is just being partially evaluated in a clause body of $(H :- Q, P, \ldots)$, where Q has been suspended to be expanded and made residual. Then, Q can be utilised as a constraint for P. When P is expanded by some clause such as $(P :- Q, R, \ldots)$, Q is solved and eliminated from the expanded code as the trivial consequence of the constraint. This somewhat context sensitive extension of partial evaluation is denoted as:

$$peval(P, Q, R)$$

which reads that R is the result of partial evaluation of P in context Q. This is viewed as a kind of *logic dependent partial computation* introduced by 4), or *partial evaluation with constraints*[5].

This suggests the following transformation. First, a new definition

$$(P_{new} :- Q, P)$$

is added for P, all of whose defining clauses share Q in their bodies, and all occurrences of P in the program are replaced by P_{new}. Second, P is partially evaluated with the constraint Q. Then, all occurrences of Q in the clause bodies are eliminated from the residual clauses for P. This transformation can be formalised and justified as the *goal replacement rule* defined in 6).

Using this method in our example, procedure calls for common pattern matching are factored out from the LHS of the different rules. If there are many common patterns, and they are factored out by the method in an appropriate order, a good resultant code will be obtained in a form something like a well

formed decision tree. This further improves the performance of the naive implementation of the production engine. The complexity of `match`, and hence that of `exec`, becomes $O((\tau - c) \times P' \times W)$ where P' is the number of patterns to be tested in a cycle after factoring and $P' \leq P$. In the worst case, $P' = P$, that is, there is no common pattern to be factored out from the different LHS of rules in RM. At the other extreme, although it is not realistic, where all of the rules have a common LHS, the complexity would become $O((\tau - c) \times \overline{p} \times W)$, ie. it would become independent of the number of rules, R.

2.4. Reorganisation of Iterative Structure

So far, some enhancement of partial evaluation has made it possible to obtain further speedup derivable from the specificity of the given problem. However, it still fails to obtain order of magnitude speedup.

Now, looking into the semantics of the program, it should be noted that the `fire` process is viewed as a generator and the `match` process as a tester. Therefore, it is expected that, by placing `match` immediately after `fire`, some shortcut from `fire` to `match` may be made, which may contribute to further speedup. This will be realised by the tentative expansion of `exec` as:

```
exec(WM0,WMn) :- one_cycle(WM0,WM1),
                 one_cycle(WM1,WM2),
                 exec(WM2,WMn).
```

and, further expansion of the two `one_cycle` calls as:

```
exec(WM0,WMn) :-  match(CS1,   WM0),
                  select(CS1,R1,WM0),
```

```
fire(    R1,WM0,WM1),
   match(CS2,   WM1),
  select(CS2,R2,WM1),
    fire(    R2,WM1,WM2),
               exec(WM2,WMn).
```

At this point, by unfolding the first `fire` call with its specialised clauses already obtained, the specialised `exec` procedure is obtained. Note that, in a body for each of the specialised `exec` clauses, the effect of the first `fire` can be propagated to the second `match` call through `WM1`. If it is known that the firing of `R1` inhibits the next firing of a rule, `R2`, in `CS2`, say, because `R1` removes an element which is required to match a pattern in `R2`, then such a code sequence corresponding to the firing of `R1` followed by `R2` can be discarded. Moreover, it is often the case that the number of commands on the RHS of a rule which causes changes in *WM* is bounded, and the number of changes in matching conditions during consecutive two cycles is also bounded. Thus, there may be chances to reduce the computation in the `match` which follows the specialised `fire`.

To make the main procedure, `exec`, utilise this fact, the well known unfold/fold program transformation[6] can be applied as shown in Fig.2.

In Fig.2, `fire_match` is a compound procedure including a shortcut from `fire` to `match`. The new procedure, `exec1`, replacing `exec`, is the reorganised main loop of the production engine, and iterates `fire_match` instead of `match` and `fire`.

After further optimisation, the complexity of `fire_match` will become linear with respect to the number of changes for matchable patterns after the firing of the last rule, which is bounded by ρ, ie.:

$$\Delta(P' \times W) \propto \overline{\mid RHS \mid} < \rho$$

If the complexity of each iteration of `exec1` is dominated by that of `fire_match`, instead of `select`, its order becomes $O(\rho)$, which is constant with respect to R, the number of rules. Thus, *super linear speedup*[7] is achieved.

2.5. More Than the RETE Algorithm

A RETE-like algorithm[8] is commonly used to improve efficiency of the production systems, OPS5 and its descendants. With this algorithm, a set of production rules is transformed into a network which can be viewed as a kind of compiled code of the rule set (Fig.3). *WM* is distributed in the network as separate local memories. In its execution, a stream of tokens, each of which is affixed by a flag indicating whether it is to be added to or removed from *WM*, is input to the network, and a set of firable rules (a conflict set) is output from the network. Then, one rule is selected from the conflict set, and fired in the next iteration (Fig.4). The RETE transformer and the RETE network driver is defined once and for all; however, a RETE network must be reconstructed each time the rule set is changed.

What if partial evaluation (Fig.5, Fig.6) is used in place of the RETE Algorithm?

If the full possible speedup obtainable from the specificity of the problem is achieved and a fully optimised production engine with respect to the given rules is obtained by enhanced partial evaluation, it will be a rival of the RETE Algorithm.

Another advantage is gained if partial evaluation is adopted. That is, a partial evaluator will be partially evaluated with respect to a production engine,

obtaining a kind of compiler that inputs a set of production rules and outputs the specialised production engine (Fig.7). Furthermore, incremental compilation[1] may be applied effectively (Fig.8), since the knowledge aquisition process is incremental in nature. This is difficult using the RETE algorithm as it is.

3. Global Control of Partial Evaluation

3.1. Spectrum of Differently Specialised Programs

In general, partial evaluation can be regarded as a sequence of atomic transformations such as the following:

(1) Unfolding a call by a body of its definition

(2) Introduction of a specialised predicate (p_a) for a call with partial input ($p(..a..)$)

$p_a(..) : -p(..a..).$

and replacing the call by a new call

$..., p(..a..), ... \text{ —> } ..., p_a(..), ...$

(3) Specialisation of the body of a newly introduced predicate definition (this can be represented as a combination of (1) and (2))

There are several control issues. The most famous is prevention of loops. Other important issues are the order of calls to be partially evaluated and selection of residual calls. Existing partial evaluators have some strategies about these issues. So far, we have developed three partial evaluators for Prolog programs, which adopt the following control strategies.

Peval[1)]

Loop detection is automatic. If a loop is found, the call is kept untouched. A user must notify Peval of calls to be residual. The notification can be done before partial evaluation or incrementally at run-time.

Automatic PE[2)]

The system consists of two stages, the analysis stage and the transformation stage. At the analysis stage, it extracts several items of information by abstract interpretation. These items include cross references and safe recursive calls (safety means that a call is guaranteed to terminate). At the transformation stage, all non-recursive calls and all safe recursive calls are unfolded.

Self-applicable PE[9)]

The program is intended to be self-applicable. No loop detection mechanism is provided. A user must notify the system of calls to be residual.

Among the control strategies, the specification of residual calls most influences the final form.

Let us look at an example. Suppose we have the following program.

```
ancestor(X,Y)      :-   parent(X,Y).
ancestor(X,Y)      :-   parent(X,Z), ancestor(Z,Y).
```

The data given is the following set of clauses.

```
parent(a1, a11).        parent(a1, a12).
parent(a11,a111).       parent(a11,a112).
parent(a12,a121).       parent(a12,a122).
        ...
```

One of the possible specifications gives instructions to expand all calls. The following program is obtained as a result.

```
ancestor(a1, a11).      ancestor(a1, a12).
ancestor(a11,a111).     ancestor(a11,a112).
ancestor(a1, a111).     ancestor(a1, a112).
ancestor(a12,a121).     ancestor(a12,a122).
ancestor(a1, a121).     ancestor(a1, a122).
        ...
```

The result is a set of unit clauses. It is much larger than the original one. If the number of the parent relation is N, then the result has $O(NlogN)$ clauses.

Another possibility is unfolding of calls to the parent relation only. The following program is obtained.

```
ancestor(a1, a11).      ancestor(a1, a12).
ancestor(a11,a111).     ancestor(a11,a112).
ancestor(a12,a121).     ancestor(a12,a122).
        ...
ancestor(a1, Y)    :-   ancestor(a11, Y).
ancestor(a1, Y)    :-   ancestor(a12, Y).
ancestor(a11,Y)    :-   ancestor(a111,Y).
ancestor(a11,Y)    :-   ancestor(a112,Y).
ancestor(a12,Y)    :-   ancestor(a121,Y).
ancestor(a12,Y)    :-   ancestor(a122,Y).
        ...
```

Furthermore, depending on the number of times recursive calls in the body of the ancestor are unfolded, a series of differently specialised programs may be obtained.

In this way, given a program, there is a spectrum of differently specialised programs, each of which corresponds to different specification of residual calls.

Peval and Self-applicable PE allow a user to specify residual calls. In these systems, given such specifications, partial evaluation is rather straightforward transformation. This is the same even in automatic PE, although such specifications are generated automatically based on an embedded criterion at the analysis stage.

However, specification of residual calls is not a trivial task and its foundation is not clear. In our experience with semi-auto partial evaluators (Peval and self-applicable PE), trial and error are required to obtain good results. But how do we know that one result is better than the other? To answer the question, first it is necessary to formalise a measure used in comparing the two programs.

3.2 Comparison of Two Specialised Programs

There can be several measures, and two programs might be compared multi-dimensionally. The following measures seem to be used by expert programmers when evaluating a program.

(1) Efficiency estimated by, for example, the number of calls, number of redundant computations, and possibility of optimisation by compiler, especially clause indexing and tail recursion optimisation, is good in the case of Prolog.

(2) Code size is a critical factor because partial evaluation may cause its combinatorial explosion.

(3) Number of distinct predicates.

(4) Possibility of application of special optimisation techniques such as those mentioned in Section 2.

Let us look at an example. We use a bottom-up parser[1] of context free grammar (CFG). An original program is a pair of two programs. One is a bottom-up interpreter of CFG rules (a meta program).

```
goal((P,Q),S0,S) :- goal(P,S0,S1), goal(Q,S1,S).
goal(C,S,S1) :- dict(F,S,S2),link(F,C),derive(F,S2,C,S1).

derive(F,S,F,S).
derive(F,S2,C,S1) :- rule1((Lemma <= (F,Rest))),link(Lemma,C),
                     goal(Rest,S2,S3),derive(Lemma,S3,C,S1).
derive(F,S2,C,S1) :- rule2((Lemma <= F)),link(Lemma,C),
                     derive(Lemma,S2,C,S1).

link(C,C).
link(F,C) :- rule1((Lemma <= (F,_))),link(Lemma,C).
link(F,C) :- rule2((Lemma <= F)),link(Lemma,C).

dict(F,[X|S],S) :- rule((F <= [X])).

rule1((A <= (B,C)))  :-  rule((A <= (B,C))).
rule2((A <= B))      :-  rule((A <= B)),\+(B=(_,_)),\+(B=[_]).
```

The other is a set of CFG rules (an object program).

```
rule((s   <= (np,vp))).  rule((np  <= (det,n))).
rule((vp  <= vi)).       rule((vp  <= (vt,np))).
rule((n   <= [boy])).    rule((n   <= [girl])).
rule((vi  <= [walks])).  rule((vt  <= [likes])).
rule((det <= [a])).      rule((det <= [the])).
```

Several possible specialisations are presented in this subsection. The first result is the same as that presented in the previous paper[1]. It is obtained by inhibiting unfolding of calls to `dict`, `link`, `goal`, `derive`. The program has essentially the same structure as that of codes generated by the BUP translator,

which was written by expert programmers and directly translates CFG rules into a Prolog program[10].

```
Result A:

dict(det,[a|A],A).        link(A,  A).
dict(det,[the|A],A).      link(det,np).
dict(n,  [boy|A],A).      link(det,s).
dict(n,  [girl|A],A).     link(np, s).
dict(vi, [walks|A],A).    link(vi, vp).
dict(vt, [likes|A],A).    link(vt, vp).

derive(A,  B,A,B).
derive(det,A,B,C) :- link(np,B),goal(n, A,D),derive(np,D,B,C).
derive(np, A,B,C) :- link(s, B),goal(vp,A,D),derive(s, D,B,C).
derive(vt, A,B,C) :- link(vp,B),goal(np,A,D),derive(vp,D,B,C).
derive(vi, A,B,C) :- link(vp,B),             derive(vp,A,B,C).
goal((A,B),C,D)   :- goal(A,C,E),goal(B,E,D).
goal(A,B,C)       :- dict(D,B,E),link(D,A),  derive(D,E,A,C).
```

In the case of automatic PE, the following result is obtained by unfolding all calls except `goal` according to the embedded strategy.

```
Result B:

goal(s, [a|B],C)     :-  goal(n,B,D), goal(vp,D,C).
goal(s, [the|B],C)   :-  goal(n,B,D), goal(vp,D,C).
goal(np,[a|B],C)     :-  goal(n,B,C).
goal(np,[the|B],C)   :-  goal(n,B,C).
goal(n, [boy|B],B).
goal(n, [girl|B],B).
goal(vp,[walks|B],B).
goal(vp,[likes|B],C) :-  goal(np,B,C).
```

If everything is unfolded, then the following extreme program is obtained.

```
Result C:

goal(s,[a,boy,walks],[]).
goal(s,[the,boy,walks],[]).
goal(s,[a,girl,walks],[]).
goal(s,[the,girl,walks],[]).
goal(s,[a,boy,likes,a,boy],[]).
goal(s,[the,boy,likes,a,boy],[]).
goal(s,[a,girl,likes,a,boy],[]).
goal(s,[the,girl,likes,a,boy],[]).
goal(s,[a,boy,likes,the,boy],[]).
goal(s,[the,boy,likes,the,boy],[]).
goal(s,[a,girl,likes,the,boy],[]).
goal(s,[the,girl,likes,the,boy],[]).
goal(s,[a,boy,likes,a,girl],[]).
goal(s,[the,boy,likes,a,girl],[]).
goal(s,[a,girl,likes,a,girl],[]).
goal(s,[the,girl,likes,a,girl],[]).
goal(s,[a,boy,likes,the,girl],[]).
goal(s,[the,boy,likes,the,girl],[]).
goal(s,[a,girl,likes,the,girl],[]).
goal(s,[the,girl,likes,the,girl],[]).
```

This is a set of all legal sentences.

Another possibility is presented below, where `goal` and `dict` are kept untouched.

```
Result D:

dict(det,[a|A],A).
dict(det,[the|A],A).
dict(n,  [boy|A],A).
dict(n,  [girl|A],A).
dict(vi, [walks|A],A).
```

```
dict(vt, [likes|A],A).

goal(np,B,C) :-    dict(det,B,E), goal(n, E,C).
goal(s, B,C) :-    dict(det,B,E), goal(n, E,D), goal(vp,D,C).
goal(vp,B,C) :-    dict(vi, B,C).
goal(vp,B,C) :-    dict(vt, B,E), goal(np,E,C).
```

Result A can be characterised by the fact that the code size of derive is the same as that of the given CFG rules. It is a good property because the code size of the result is always bound by that of the given data. The code size of result C is equal to the number of legal sentences. This partial evaluation is good for toy rules, but is not realistic. B and D extract legal chains of rules and discard illegal ones. B enumerates legal chains until it reaches terminal categories such as `n`, `det`, `vi` and `vt`. C expands terminal categories further into individual words. Thus, D has more codes than A and B has more codes than D. D is more efficient than A, because legal chains of rules are already constructed in D while A must generate them at run-time. D may benefit from the optimisation technique of collecting common predicate calls mentioned in Section 2. For instance, suppose we have two clauses:

```
goal(np, B,C) :- dict(det,B,D), goal(n,D,C).
goal(np, B,C) :- dict(det,B,D), goal(n,D,E), goal(rel_s,E,C).
```

Then these can be transformed into:

```
goal(np, B,C) :- dict(det,B,D), goal(n,D,E), goal(np1,E,C).
goal(np1,B,B).
goal(np1,B,C) :- goal(rel_s,B,C).
```

Automated partial evaluation, which competes against expert programmers, must take these multi-dimensional comparisons into consideration. However,

their evaluations change when the measures change. Results also change when data (CFG rules) changes. *Thus, there is no fixed strategy of partial evaluation which is always guaranteed to generate the best result.* Therefore, what is required in competitive partial evaluation is the flexibility of being able to respond appropriately to given data and measures.

3.3 State-Space Analogy

Our approach to realise competitive partial evaluation is stepwise specialisation where, at each stage, a result of specialisation is evaluated by the given measures and analyzed to find out whether there is room for additional specialisation which improves the current result.

The framework is analogous to the state-space search. Here, a state and an arc connecting two states correspond to an intermediate result and an atomic transformation. There can be several arcs from one state, which correspond to more than one possibility of transformations. A state is evaluated multi-dimensionally by the difference between its intermediate result and an original program. One dimension corresponds to a measure. For each atomic transformation and each dimension, a primitive difference is defined. For instance, unfolding a call is defined as decreasing the number of calls by one and increasing code size by, at most, the number of clauses defining the call. The difference between two programs with respect to one dimension is calculated by integrating the primitive difference of atomic transformations, the sequence of which transforms one to the other.

Partial evaluation can be viewed as a search for an optimal state with respect to the given evaluation function, starting from the original program. For state-space search, several search strategies are known. The one we are considering is the best first search. Its skeleton is as follows:

Input: P_0: an original program. S: A set of programs. Termination condition specified in terms of the evaluation function.

Output: Specialised program satisfying the termination condition.

(0) Let S be $\{P_0\}$.

(1) Select the best program, P, from S based on the evaluation function. (If all candidates have been tried, then backtrack to the last choice point (1).)

(2) If P satisfies the termination condition, then it is returned. Otherwise, go to (3).

(3) Enumerate possible atomic transformations, $T_1, ..., T_n$, to P and apply them. Let R be a set of resultant programs which are not worse than P.

(4) If R is empty, then backtrack to (1). Otherwise, replace P in S by R and go to (1).

As an example, the state-space view of the BUP example discussed previously is shown in Fig.9.

In Fig.9, a directed arc indicates the possibility of transformation. State E corresponds to the program optimised by clustering the same computations. $[X, Y]$ labelling an arc means that the number of distinct predicates and the number of clauses increase by X and Y, respectively. If we adopt these rather static measures, result B will be obtained as the best result.

4. Conclusion

This paper presented two problems in competitive partial evaluation, maximal speedup and global control of partial evaluation.

To solve the first problem, we showed the enhancement possible for partial evaluation obtaining further speedup utilising the specificity of a given problem. Three techniques were presented, optimising a simple production engine as an example. The first is the inferential ability to derive information that can be used to specialise more parts of a program, where it is not very obvious that the naive PE can derive and use such information. The second is the factoring of common calls of procedures, thereby eliminating a number of redundant computations. The third is the folding transformation which merges the generator (`fire`) and the tester (`match`), and reduces the search space greatly at partial evaluation time. The partial evaluator thus enhanced can improve the naive implementation of a production engine close to that given by the RETE algorithm. Moreover, partial evaluation is better than the RETE algorithm in its generality and in the possibility of extraadvantages in compilation and incremental compilation.

The second problem seems to be implicitly recognised by all scientists in this field. However, the foundation of evaluating the results of partial evaluation is not clear, and hence no scientific result has been obtained yet. In fact, there is a large spectrum of possible results of partial evaluation and there is a big gap between partial evaluation and expert programmers with respect to the ability of evaluation of programs. Our approach is to introduce a global framework which can view and control all possible partial evaluations. We consider that within this framework we can guarantee the quality of results of partial evaluation.

Acknowledgements

We would like to express our gratitude to Dr. Kazuhiro Fuchi, director of ICOT, for giving us the opportunity of doing this research. We also wish to thank Dr. Koichi Furukawa, deputy director of ICOT, for his encouragement, and the referees for their helpful suggestions and comments.

References

1) Takeuchi,A., Furukawa,K., Partial Evaluation of Prolog Programs and Its Application to Meta Programming, in: Kugler,H.-J. (ed.), *Information Processing 86* (North-Holland, Amsterdam, 1986) 415-420

2) Safra,S., Shapiro,E., Meta Interpreters for Real, in: Kugler,H.-J. (ed.), *Information Processing 86* (North-Holland, Amsterdam, 1986) 415-420

3) Fujita,H., On Automating Partial Evaluation of Prolog Programs, Technical Memorandum TM-250, ICOT (1987) (in Japanese)

4) Futamura,Y., Logic Dependent Partial Computation, in *The Third Meeting on Program Transformation and Synthesis,* Japan Society for Software Science and Technology (1987) (in Japanese)

5) Fujita,H., An Algorithm for Partial Evaluation with Constraints, Technical Memorandum TM-367, ICOT (1987) (in Japanese)

6) Tamaki,H., Sato,T., Unfold/Fold Transformation of Logic Programs, in: *Proc. of The Second International Logic Programming Conference, Uppsala* (1984) 127-138

7) Jones,N.D., Challenging Problems, *private communication* (1987)

8) Forgy,C., RETE: A Fast Algoritm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence* 19 (1982) 17-37

9) Fujita,H., Furukawa,K., A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation, Technical Report TR-250, ICOT (1987)

10) Matsumoto,Y., Tanaka,H., Hirakawa,H., Miyoshi,H., Yasukawa,H., BUP: A Bottom-Up Parser Embedded in Prolog, *New Gener. Comput.* 1 (1983) 145-158

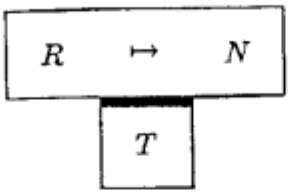
Figures

```
        exec(WM0,WMn) :-     one_cycle(WM0,WM1), exec(WM1,WMn).
        exec(WM, WM).

    one_cycle(WM0,WM1) :-  match(CS,    WM0),
                           select(CS,RHS,WM0),
                             fire(   RHS,WM0,WM1).

         match(CS,WM)  :- setof(Rule,match1(Rule,WM),CS).

match1((LHS=>RHS),WM)  :- rule(LHS=>RHS), match2(LHS,WM).

   match2([X|LHS],WM)  :-   match3(X,WM), match2(LHS,WM).
   match2([],_).

        match3(+X,WM)  :-   match4(X,WM).
        match3(-X,WM)  :- \+match4(X,WM).

         match4(X,WM)  :- { X matches with an element in WM }

    select(CS,RHS,WM)  :- { select RHS of a rule in CS
                            taking acount WM          }

 fire([X|RHS],WM0,WMn) :-  fire1(X,WM0,WM1), fire(RHS,WM1,WMn).
 fire([],     WM, WM).

     fire1(+X,WM0,WM1) :-    add(X,WM0,WM1).
     fire1(-X,WM0,WM1) :- remove(X,WM0,WM1).

        add(X,WM0,WM1) :- {    add X  to  WM0 obtaining WM1 }
     remove(X,WM0,WM1) :- { remove X from WM0 obtaining WM1 }
```

Figure 1 Production Engine

```
exec(W0,W2) :- one_cycle(W0,W1), exec(W1,W2).
               one_cycle(W0,W1) :-
               match(C,W0), select(C,R,W0), fire(R,W0,W1).
|
| unfold ..... one_cycle(W0,W1)
V
exec(W0,W2) :- match(C,W0), select(C,R,W0), fire(R,W0,W1),
                                                exec(W1,W2).
|
| define ..... exec1(R,W0,W2) :-            fire(R,W0,W1),
|                                                exec(W1,W2).
| fold ....... exec  by  exec1
V
exec(W0,W2) :- match(C,W0), select(C,R,W0), exec1(R,W0,W2).

exec1(R,W0,W2) :- fire(R,W0,W1), exec(W1,W2).
|
| unfold ..................... exec(W1,W2).
V
exec1(R,W0,W2) :- fire(R,W0,W1),
   match(C,W1), select(C,R2,W1), fire(R2,W1,Wi), exec(Wi,W2).
|
| fold by ... exec1(R2,W1,W2) :- fire(R2,W1,Wi), exec(Wi,W2).
|
V
exec1(R,W0,W2) :- fire(R,W0,W1),
   match(C,W1), select(C,R2,W1), exec1(R2,W1,W2).
|
| define ... fire_match(R,C,W0,W1) :- fire(R,W0,W1), match(C,W1).
|
| fold ..... exec1  by  fire_match
V
exec1(R,W0,W2) :- fire_match(R,C,W0,W1), select(C,R2,W1),
                                            exec1(R2,W1,W2).
```

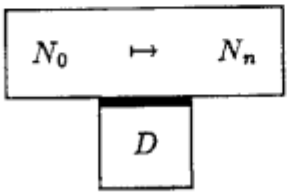
Figure 2 Reorganising Iterative Structure



R: production rules
N: RETE network
T: RETE transformer

$T : R \mapsto N$

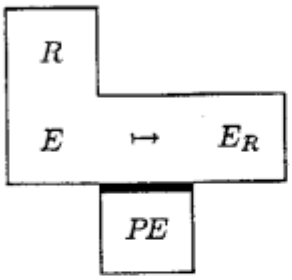
Figure 3 The RETE transformation



N_0: RETE network in ininitial state
N_n: RETE network in final state
D: RETE network driver

$D : N_0 \mapsto N_n$

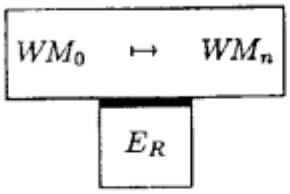
Figure 4 Driving the RETE network



R: production rules
E: production engine
E_R: specialised production engine wrt. R
PE: partial evaluator

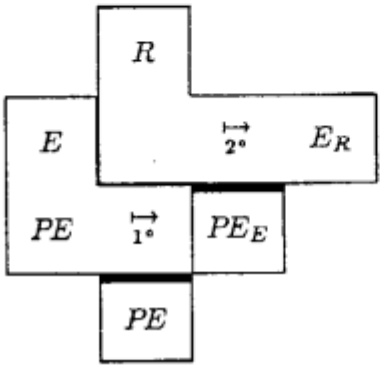
$PE : E \times R \mapsto E_R$

Figure 5 Partial evaluation



$E_R : WM_0 \mapsto WM_n$

Figure 6 Execution of the specialised production engine



$2°\ PE_E : R \mapsto E_R$

$1°\ PE : PE \times E \mapsto PE_E$

Figure 7 Compiler generation and compilation

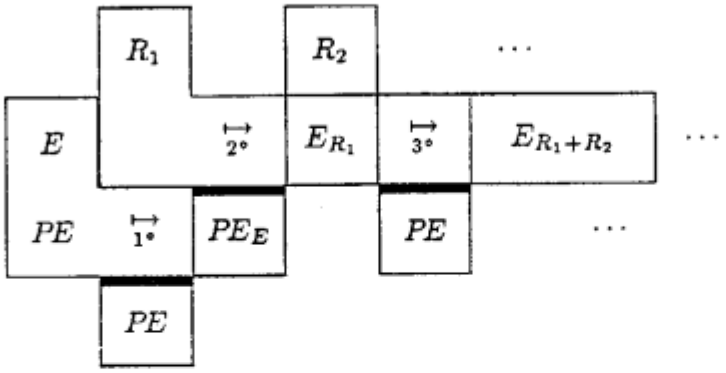


Figure 8 Incremental compilation

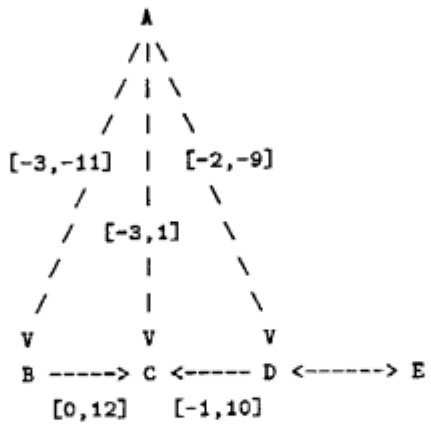


Figure 9 State-space view of BUP example