TM-0367

An Algorithm for Partial Evaluation with Constraints

by
H. Fujita

August, 1987

An Algorithm for Partial Evaluation with Constraints

Hiroshi FUJITA

ICOT Research Center

Institute for New Generation Computer Technology,

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

A new algorithm for partial evaluation of Prolog programs is presented. The algorithm is enhanced by considering constraints. It may be a Prolog counterpart of Futamura's *Logic Dependent Partial Evaluation* [3] for functional programs. Using the algorithm, *more than linear* improvement [4] of runtime performance for a program can be obtained, provided with a specific partial input that has potential for certain optimization.

1. Introduction

It has been recognized that partial evaluation has a limit to its capacity to obtain real improvement of performance for a program. Propagation of constants and unfolding calls by their definitions, which are performed by usual partial evaluators, only reduce certain intervals of computation in the whole computation sequence. However, they never make a shortcut in the computation sequence nor do they change the program control to eliminate redundant computations. Researchers in the area have set out to enhance the power of partial evaluation [5] to obtain this kind of optimization.

There are three major problems in partial evaluation that have been researched so far: automating partial evaluation, obtaining a self applicable partial evaluator, and making partial evaluation more practical. The first two have been solved with real implementations [1],[2], albeit unsatisfactorily. However, the last problem still remains untouched and therefore unsolved. To obtain a more powerful partial evaluator and to make it more practical, partial evaluation should be enhanced by additional capabilities.

This report presents a new algorithm of partial evaluation. It not only specializes a program with respect to a given partial input, but also performs some shortcutting as an optimization on the residual program using constraints, if possible. The idea originates in Futamura's *Logic Dependent Partial Evaluation* (or *General Partial Evaluation*) [3], with reformulation to adapt it to Prolog programs instead of functional programs.

2. Outline of the Algorithm

The partial evaluation algorithm comprises two stages: specialization and shortcutting. The algorithm is informally stated in Fig.1 and Fig.2.

When the top-level Spec in Fig.1 terminates, the resultant specialized program clauses are given in S'. A query to the program is to be given by G' instead of G. The idea of the specialization is that for every distinct call pattern (up to variable renaming) of a goal in the original program, S_0, and its descendants generated by unfolding, specialized clauses for the call are created by first copying matching clauses from S_0, then changing the predicate name of the call and of the heads of the copied clauses, so that every goal in S_0 calls only its own clauses not shared by other goals of the same predicate. Case (3) of Spec is a kind of folding operation, but differs slightly from the usual definition of a folding rule as a general method in program transformation techniques.

Now, observing the resultant specialized clauses, it is often noticed that there are a number of redundancies in them. Among others, there is often a predicate introduced in the process for case (4) of Spec, such that it has only a single clause in S'. This predicate

```
$S_0$ := a set of original program clauses,
De: a set of folding predicate definitions := $\phi$,
S': a set of specialized program clauses := $\phi$,

   Spec: input a goal G, output a specialized goal G',

     If G is of a primitive predicate,
     then if G is evaluable,                                        ... (1)
          then evaluate G, and
               G' := true/fail according to the evaluation,
          else G' := G,
     else if G is a conjunction, (A,B),                             ... (2)
     then recursively compute Spec(A) and Spec(B),
          getting A' and B' respectively, and
               G' := and(A',B'),
     else if there exists (H :- B) in De such that                  ... (3)
             B is a variant of G with the renaming, $\sigma$,
     then G' := H$\sigma$,
     else if there exists non-null set {$H_i$ :- $B_i$} in $S_0$ such that   ... (4)
             $H_i$ is unifiable with G with the mgu, $\theta_i$,
     then create a new goal $G_n$ with a new predicate symbol and
          all distinct variables occurring in G as its arguments, and
          G' := $G_n$, and
          for each i,
              compute Spec($B_i\theta_i$) getting $B_i$', and
              if $B_i$'$\neq$fail,
              then add ($G_n\theta_i$ :- $B_i$') to S',
     else G' := fail.
```

Figure 1 [STAGE 1] *Specialization*

is the most trivial one which should be eliminated, since it only adds an overhead of call-exit handling that produces the opposite result to the principle of partial evaluation. Looking more closely into program S', another redundancy may be found. For instance, there may be a clause in S' that cannot be called by any goal in S'. This fact can be revealed by considering some residual goals in the specialized clauses as constraints for other goals, and by checking the consistency of the constraints during expansion of the goals by their defining clauses. STAGE 2 in Fig.2 deals with a post process for the partial evaluation algorithm.

```
While there exists (H :- B) in De such that
    one of the following transformations are possible,
do the transformation:

 (1) *eliminating a trivial predicate*
    if there exists only a single clause (B :- C) in S' for B,
    then remove (B :- C) from S', and
        for each ($H_i$ :- $B_i$) in S' such that
            $B_i$ is unifiable with B with mgu, $\tau$,
        do remove ($H_i$ :- $B_i$) from S', and
            add ($H_i$ :- C)$\tau$ to S', and
            remove (H :- B) from De, and
            add (H :- C) to De,

 (2) *reduction based on constraint*
    if there exists common constraint C for every call of B,
    then for each ($B_i$ :  $D_i$) in S' such that
            $B_i$ is unifiable with B with mgu, $\rho$,
        do check whether $\rho$ and $D_i\rho$ is consistent with
                the constraint C, and
            if it is not consistent,
            then remove ($B_i$ :- $D_i$) from S',

where
  *constraint*:  For a goal $B_i$ in a clause
        H :- $B_1$, ..., $B_{i-1}$, $B_i$, $B_{i+1}$, ..., $B_n$,
    ($B_1$, ..., $B_{i-1}$, $B_{i+1}$, ..., $B_n$) is called its constraint.

  *consistent*:
    A unifier $\rho$ is said to be consistent with a constraint C,
    if C$\rho$ is satisfiable under the program S'
    (and under the evaluation system of primitive predicates).
    A goal G is said to be consistent with a constraint C,
    if the conjunction, (C,G), is satisfiable under the program S'
    (and under the evaluation system of primitive predicates).
```

Figure 2 [STAGE 2] *Shortcutting*

In fact, the second rule is the heart of the algorithm, although it is simplified as far as possible. For the algorithm to be more concrete, it must be further elaborated. The more sophisticated the definition of constraint and consistency checking provided, the more powerful the partial evaluation that can be obtained. A simpler implementation of the rule may be given by restricting the general definitions of constraint and consistency.

In the sequel, constraints are restricted to equalities or non-equalities. The consistency check is applied only to equalities or non-equalities. For example,

```
    p(X,Y) :- X\=a, q(X,Y).

    q(a,Y) :- r(Y).
    q(X,Y) :- X\=b, s(Y).
    q(X,Y) :- X\=a, t(Y).
```

The constraint for a goal, q(X,Y), in the first clause is X\=a. The mgu, {a/X}, unifying q(X,Y) with q(a,Y), the head of the first clause for q, is inconsistent with the constraint, X\=a. Hence, the clause is never used for solving p(X,Y). The goal, X\=b, in the body of the second clause for q, after unifying q(X,Y) with its head, is consistent with the constraint, X\=a. The goal, X\=a, in the body of the third clause for q is identical to (and, of course, consistent with) the constraint, X\=a, hence, it can be eliminated from the body as far as the clause is used for solving p(X,Y).

3. An Example

Consider the example program for string matching below.

```
match(P,T) :- match1(P,T,P,T).

match1([A|Ps],[A|Ts],P,T) :- match1(Ps,Ts,P,T).
match1([A|Ps],[B|Ts],P,[_|T]) :- A\==B, match1(P,T,P,T).
match1([],_,_,_).
```

The first argument of match, P, is a pattern string which is matched against the second argument, T, a target string. This is a Prolog version of the example quoted from [1]. The procedure, match1, checks matching of pattern P with target T at the top element from each of them, A and B. If they are identical, the process moves to the next element along both the pattern and the target. Continuing the successful matching, it eventually reaches a point where all elements in the pattern have been checked, when the whole match has succeeded. Otherwise, matching fails at some point going down to the pattern and the target, then the top element of the target is discarded and the whole matching is restarted from the beginning of the pattern and the tail of the target. Hence, even if partial match has succeeded to some depth into the pattern and the target, the fact is discarded and never utilized as useful information for later processes of matching.

Now, suppose that the pattern is fixed to the list [a,a,a,b]. Then, the algorithm outputs the following trace listing.

[STAGE 1] *Specialization*

```
 | ?- stage1(match([a,a,a,b],T),A).

 Def g1(A) :- match([a,a,a,b],A).
 Def g2(A) :- match1([a,a,a,b],A,[a,a,a,b],A).
 Def g3(A) :- match1([a,a,b],A,[a,a,a,b],[a|A]).
 Def g4(A) :- match1([a,b],A,[a,a,a,b],[a,a|A]).
 Def g5(A) :- match1([b],A,[a,a,a,b],[a,a,a|A]).
 Def g6(A) :- match1([],A,[a,a,a,b],[a,a,a,b|A]).

 g6(A).
 g5([b|A]) :- g6(A).

 Def g7(A,B) :- match1([a,a,a,b],[a,a,A|B],[a,a,a,b],[a,a,A|B]).
 Def g8(A,B) :- match1([a,a,b],[a,A|B],[a,a,a,b],[a,a,A|B]).
 Def g9(A) :- match1([a,b],[A|B],[a,a,a,b],[a,a,A|B]).

 g9(a,A) :- g5(A).
```

```
Def g10(A) :- match1([a,a,a,b],[a,A|B],[a,a,a,b],[a,A|B]).
Def g11(A) :- match1([a,a,b],[A|B],[a,a,a,b],[a,A|B]).

g11(a,A) :- g4(A).

Def g12(A) :- match1([a,a,a,b],[A|B],[a,a,a,b],[A|B]).

g12(a,A) :- g3(A).
g12(A,B) :- a\==A, g2(B).
g11(A,B) :- a\==A, g12(A,B).
g10(A,B) :- g11(A,B).
g9(A,B) :- a\==A, g10(A,B).
g8(A,B) :- g9(A,B).
g7(A,B) :- g8(A,B).
g5([A|B]) :- b\==A, g7(A,B).
g4([a|A]) :- g5(A).
g4([A|B]) :- a\==A, g10(A,B).
g3([a|A]) :- g4(A).
g3([A|B]) :- a\==A, g12(A,B).
g2([a|A]) :- g3(A).
g2([A|B]) :- a\==A, g2(B).
g1(A) :- g2(A).
```

STAGE 1 terminates successfully. There are twelve definitions of new goals, `g1` to `g12`, each of which corresponds to a distinct goal pattern that appears in subsequent computations under the top level query, `match([a,a,a,b],T)`. The program clauses are generated for each newly introduced predicate, g_i, by unfolding the body of its definition, then folding by another definition. Observe that each of `g1`, `g6`, `g7`, `g8`, and `g10` has only a clause that is trivial. Hence, STAGE 2 follows.

[STAGE 2] *Shortcutting*

```
-- eliminating g1(A)
-- eliminating g6(A)
```

```
g5([b|A]).
-- eliminating g7(A,B)
g5([A|B]) :- b\==A, g8(A,B).
-- eliminating g8(A,B)
g5([A|B]) :- b\==A, g9(A,B).
-- eliminating g10(A,B)
g4([A|B]) :- a\==A, g11(A,B).
g9(A,B) :- a\==A, g11(A,B).
-- evaluating constraints for g11(A,B)
g11(A,B) :- g12(A,B).
-- eliminating g11(A,B)
g4([A|B]) :- a\==A, g12(A,B).
g9(A,B) :- a\==A, g12(A,B).
-- evaluating constraints for g12(A,B)
g12(A,B) :- g2(B).
-- eliminating g12(A,B)
g3([A|B]) :- a\==A, g2(B).
g4([A|B]) :- a\==A, g2(B).
g9(A,B) :- a\==A, g2(B).
```

The result is:

```
g2([a|A]) :- g3(A).
g2([A|B]) :- a\==A, g2(B).

g3([a|A]) :- g4(A).
g3([A|B]) :- a\==A, g2(B).

g4([a|A]) :- g5(A).
g4([A|B]) :- a\==A, g2(B).

g5([b|A]).
g5([A|B]) :- b\==A, g9(A,B).

g9(a,A) :- g5(A).
g9(A,B) :- a\==A, g2(B).
```

The trivial clauses for g1, g6, g7, g8, and g10 are eliminated. Then, all the clauses that call g11 have a\=A as their constraint. a\=A is inconsistent with the first clause for g11, hence, the clause is discarded. The body in the second clause for g11 is reduced to only a call to g12, because the goal, a\=A, is identical to the constraint, and can be eliminated. Now that the clause for g11 becomes trivial, it is eliminated. Similarly, the clauses for g12 are eliminated after constraint evaluation. To run the program, a query like g2(T) should be given in place of match([a,a,a,b],T).

Actually, the partially evaluated program for match([a,a,a,b],T) corresponds to the decision tree shown in Fig.3. Each of the numbered nodes corresponds to g_i in the definitions introduced by STAGE 1. In particular, the node marked with an asterisk corresponds to the predicate that remains in the final residual program clauses. The number in brackets denotes the call that is never called. The capital, A (B), indicates that it is known at that point that the element is not a (b). Question marks in or under strings indicate the position at which matching is checked. The tree is, in fact, not a tree but a cyclic graph, which reflects the recursive execution for the program.

The difference between the behavior of the original program and that of the partially evaluated program is illustrated in Fig.4. The partially evaluated program realizes the same shortcutting as the KMP algorithm does.

In general, if the pattern has a repeated prefix as in $a^N b$, and the target has $a^N c$ as a prefix, then the original program repeats check $a = a$, $N - 1$ times, which has been already satisfied before unmatch $b \neq c$, shifting elements in the target one by one in each step. The partially evaluated program skips immediately to the element just after c in the target. Thus, if it takes time τ to match an element, time $N\tau$ is saved in the partially evaluated program. This might support the claim that the partially evaluated

```
                          1 aaab
                          | ....
                          |
                        * 2 aaab
                        /   ?...
                      /          \
                  * 3 aaab         aaab _______ aaab
                  /   a?..         A...        A?...
                /          \                        \
            * 4 aaab         aaab _______ aaab        2
            /   aa?.         aA..        aA...
          /          \                    ?     \
     * 5 aaab          aaab _______ aaab          \
       /  aaa?         aaA.        aaA              \
    /          \                    ?                 \
 6 aaab         | aaab              |                   \
 | aaab         | aaaB         ---- |          ---- |
 |              |            /      |        /      |
success         7 aaab       |      10  aaab  |      12   aaab
                |aaaB.       |      |aaaA..  |     /  aaaA...
                | ?          |      | ?      | <3>      ?    \
                |            |      |        |                 2
                8 aaab       |      11  aaab |
                |aaaB.       |     /  aaaA.. |
                |  ?         | <4>      ?   \____/
                |            |
             * 9 aaab        |
             /  aaaB.        |
           /        ?  \____/
         5
```

Figure 3 Decision tree for `match([a,a,a,b],T)`

program has obtained more than linear improvement in runtime performance, although it depends on the partial input of very specific patterns.

4. Discussion

It is stated in [3] that the prover for proving an implication of a *computational information* from another is of any kind: propositional, pedicate calculus, or whatever. Similar arguments are possible in our algorithm, that constraints can be of any kind, and consistency checking can be done to any degree of accuracy. The stronger the prover

```
1) a a a b          2) a a a b         3) a a a b
   =                     =                    X
   a a c a a a b       a a c a a a b      a a c a a a b

4)   a a a b        5)   a a a b
     =                     X
   a a c a a a b       a a c a a a b

6)     a a a b
       X
   a a c a a a b

7)       a a a b    8)       a a a b    9)       a a a b    10)      a a a b
         =                     =                     =                     =
   a a c a a a b       a a c a a a b       a a c a a a b       a a c a a a b

         (a) by the original program

             1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10

         (b) by the partially evaluated program

             1 -> 2 -> 3 ----------------> 7 -> 8 -> 9 -> 10
```

Figure 4 Example execution of `match([a,a,a,b],[a,a,c,a,a,a,b])`

with which the partial evaluator is equipped, the more opportunities of optimization there are. However, the total cost of partial evaluation will also increase. The optimal compromise of the power of the prover cannot be predicted, because it depends solely on the input data partially given.

The proof of the termination of the algorithm has not been established yet. For now, the brief outline of the proof is as follows. In STAGE 1, the recursive procedure, Spec, may go infinitely deep as it expands the subgoals of recursive predicates. However, the termination should be guaranteed if certain well founded ordering is found between the recursive calls in the course of the expansions. In STAGE 2, both rules decrease the number of clauses in S' when applied, hence, the termination should be obvious. More rigorous treatment should be developed after the algorithm is elaborated and set.

5. Conclusion

A new algorithm for partial evaluation of Prolog programs has been presented. The algorithm first specializes the original program with respect to the specialized query, then performs reductions for the specialized program on the basis of constraint evaluation. The improvement in performance of the resultant program exceed that given by a simple partial evaluator (in the sense that it does not only reduce some parameter passings, but also makes some shortcuts), because of the specificity of the partial input and its full utilization by constraint evaluation mechanism in the algorithm. As an example, the same result has been shown to be obtainable for the Prolog version of the string matching program as the results described in [3]. More work must be done to sophisticate and formalize the constraint handling part of the algorithm.

References

[1] Fujita, H. and Furukawa, K., On Automation of Partial Evaluation of Prolog Programs, ICOT TM-250, 1987, *(in Japanese)*

[2] Fujita, H. and Furukawa, K., A Self Applicable Partial Evaluator and Its Use in Incremental Compilation, ICOT TR-258, 1987

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

[4] Jones, Neil D., Challenging Problems, *private communication*, 1987

[5] Takeuchi, A. and Fujita, H., Competitive Partial Evaluation – Some Remaining Problems of Partial Evaluation –, *to appear*