ICOT Technical Report: TR-344

TR-344

Partial Evaluation of GHC Programs Based on UR-set with Constraint Solving

by
H. Fujita, A. Okumura &
K. Furukawa

February, 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

Partial Evaluation of GHC Programs Based on UR-set with Constraint Solving

Hiroshi FUJITA, Akira OKUMURA and Koichi FURUKAWA

ICOT Research Center,
Institute for New Generation Computer Technology,
1-4-28 Mita, Minato-ku, Tokyo 108, Japan

Abstract: A method of partial evaluation of GHC programs based on *UR-set* [Furukawa 87] is presented. Unfolding is not allowed in GHC as freely as in Prolog beacuse of synchronization issues inherent in the language semantics [Ueda 85]. Hence, other techniques are required to enhance the UR-set and to make partial evaluation more effective. To this end, a kind of constraint solving is introduced. If the partial evaluator is equipped with a constraint solver suitable for the domain of the given program, runtime efficiency of the resultant program may be improved to a degree comparable to that obtainable by Futamura's *Generalized Partial Computation* [Futamura 87].

1. Introduction

There has been considerable effort in constructing partial evaluators for logic programming languages. There are several implementations in Prolog that have successfully optimized meta-programs [Takeuchi 86 and Levi 86] (also [Safra 86] for FCP), and generated compilers and compiler generators by self-application of the partial evaluator [Fujita 87a and Fuller 87].

To what extent partial evaluation is able to perform is strongly dependent on the specific language used for the subject programs. In particular, the ease of unfolding (expansion of a call by its definition) is the key to making partial evaluation feasible. In pure Prolog, unfolding can be performed relatively freely. However, it turns out to be very difficult in concurrent logic languages. One must be very careful about the conditions that allow application of unfolding rules in concurrent logic programs. Recently, a set of unfolding rules called the *UR-set* has been defined for GHC programs by [Furukawa 87], and shown to be feasible.

This paper presents a method of partial evaluation of GHC programs based on the UR-set. This method specializes a program with respect to a specific goal pattern, performing some simplification on the specialized program on the basis of constraint solving.

By propagation of constants, even if accompanied with reduction based on unfolding in the UR-set, only a few computations can be reduced. Hence, it is said that the

runtime efficiency of the subject programs may not be significantly improved by partial evaluation. This motivated another direction of research to make partial evaluation more effective. Futamura's *Generalized Partial Computation* [Futamura 87] is the latest result for functional languages in this approach. This paper shows that comparable results can be obtained by incorporating constraint solvers into the partial evaluation algorithm presented in this paper.

Section 2 gives a brief description of the UR-set for GHC programs. Section 3 presents a partial evaluation algorithm based on the UR-set. In section 4, the partial evaluation algorithm is extended to have some constraint solver to obtain further possibilities of optimization.

2. The UR-set

The UR-set is a set of transformation rules for GHC programs. It consists of the following four rules.

Rule 1 : *Unification Execution/Elimination*

An explicit unification (=/2) appearing in the guard or the body of a clause, C, is symbolically executed within the body part; that is, a further instantiated value substitutes corresponding variable occurrences within the body. If a unification in the guard fails, the clause is eliminated. Furthermore, if neither side of = includes any variables which also appear in any other literal of the clause, the unification goal is eliminated after the substitution. Thus, a new clause, C', is derived from the original C. A new program is derived by replacing C of the original program by C'.

Examples: `(p(X) :- X=a | q(X))` is substituted by `(p(X) :- X=a | q(a)).`

`(p :- true | X=a, q(X))` is substituted by `(p :- true | q(a)).`

Rule 2 : *Unfolding at an Immediately Executable Goal*

Before stating the body of Rule 2, several notions need to be introduced.

Definition : A clause for a given goal is called

satisfied : if its guard is already true without further argument instantiations,

candidate : if the goal is not sufficiently instantiated to judge whether the guard is true or not,

unsatisfiable : if the guard is already known to be unsatisfiable.

Example: For a goal, `p(1,A)`,

`(p(X,Y) :- X = 1 | ...)` is satisfied,
`(p(X,Y) :- Y = 1 | ...)` is candidate, and
`(p(X,Y) :- X = 2 | ...)` is unsatisfiable.

Definition : A goal is said to be *immediately executable* if there is no candidate clause for that goal.

Now let a clause, C, be $(H \leftarrow G \mid B)$. Then C is unfolded at an immediately executable body goal, $B_i \in B$, by all its satisfied clauses, C_i^j $(1 \leq j \leq l$; l is the number

of satisfied clauses). The resultant clause, D_i^j, is obtained from the original clause, C, by replacing goal B_i by the body of C_i^j. D_i^j is a guarded resolvent of C and C_i^j whose guard goals are the same as C, because the guards of C_i^j must be true. Thus, a new program is derived by replacing clause C of the original program by all of D_i^j.

Example: {(p :- true | q,a(1),r)} is substituted by

{(p :- true | q,b,c,r), (p :- true | q,d,e,r)}

where {(a(X) :- X=1 | b,c), (a(X) :- X>0 | d,e), (a(X) :- X=2 | f,g)}

Rule 3 : *Predicate Introduction and Folding*

Let a clause, C, be $(P \leftarrow G \mid (U \cup B))$, where $U_i \in U$ are output unifications and $B_j \in B$ are not output unifications. Furthermore, let the intersection of a set of variables appearing in $G \cup U$ and a set of those appearing in B be $X_1, X_2, \ldots, X_r$. Then, a new clause, C_1, of $newP$ is introduced as

$$newP(X1, X2, .., Xr) \leftarrow true \mid B.$$

Then, B of C is folded by C_1 and a transformed clause, C', is obtained as

$$P \leftarrow G \mid (U \cup \{newP(X1, X2, .., Xr)\}).$$

Thus, a new program is derived by replacing clause C of the original program by C_1 and C'. This rule is used to transform clauses into forms where Rule 4 can be applied.

Example: (p(X,Y) :- X>0 | Y=[X|Z], q(Z), r) is substituted by

```
{ (p(X,Y) :- X>0 | Y = [X|Z], newP(Z)),
  (newP(X) :- true | q(X), r)          }
```

Rule 4 : *Unfolding across the Guard*

Let a clause, C, be $(H \leftarrow G \mid B)$. If neither Rule 1 nor Rule 2 can be applied to C and no $B_i \in B$ is an output unification, C is unfolded at each B_i independently. That is, a set of clauses S_i is derived by unfolding C at B_i for each $i (1 \leq i \leq n)$. However, if there are clauses in any S_i which have meaningless guard goals, they are discarded. Let S_i' be a set of clauses in B_i which are not discarded. A new program is derived by replacing C by the union of all S_i'.

Some clauses are discarded because unfolding may move a guard goal to an inappropriate place. For example, if (p :- true | q(X), r(X)) is unfolded at q(X) by (q(X) :- X=0 | s), the result is (p :- X=0 | s, r(X)). However, X=0 cannot be a guard condition at that place but an assignment goal. It may cause r(X) to fail, and it must therefore be discarded.

Example: (p(X) :- true | q(X), r(X)) is replaced by

```
{ (p(X) :- X>3 | q1, r(X)),                 { (q(X) :- X>3 | q1),
  (p(X) :- X=<3 | q2, r(X)),     where       (q(X) :- X=<3 | q2),
  (p(X) :- X<2 | q(X), r1),                  (r(X) :- X<2 | r1),
  (p(X) :- X>=2 | q(X), r2) }                (r(X) :- X>=2 | r2) }
```

3. Partial Evaluation of GHC Programs

3.1 Specialization of Clauses

Each goal pattern appearing in the original program clauses as well as the top-level query pattern is examined. Some goals may be evaluated or immediately executed, others are forced to remain as they are. Even in the latter case, if there is a goal not the most general goal, its private clauses are created as the instantiated subset of the original clauses. Another partial evaluation process may be triggered in the course of the current specialization process because of substitution propagation. All the processes will terminate when no more processes can be triggered. Thus, specialized clauses are obtained, which are more efficient than the original clauses with respect to goals containing partially instantiated arguments. Fig.1 is a sketch of the algorithm.

The idea of the specialization process is that for every distinct call pattern (up to variable renaming) of a goal in the original program, P, and its descendants generated by (tentative) unfolding, specialized clauses for the call are created. Each goal pattern will be replaced by the newly introduced predicate name, and will have its own set of clauses that can be called via the new predicate name, while it is essentially an instantiated subset of clauses in P for the original goal.

Let a top-level query to the given program always be an instance of an atomic goal from a set, $\{A_i\}$. Then, partial evaluation can be started on each C_i by activating $NewClause(C_i)$, where $C_i = (Q_i \leftarrow true \mid A_i)$ and Q_i is an introduced atom of a new predicate symbol with all distinct variables appearing in A_i.

Example 1: Immediate execution on constant input value

```
C:   q1(X,Y,Z) :- true | append([1,2|X],Y,Z)
P:   { append([H|X],Y,Z) :- true | Z=[H|Zs], append(X,Y,Zs)
       append([],Y,Z) :- true | Z=Y                          }
C':  q1(X,Y,Z) :- true | Z=[1,2|Z1], append(X,Y,Z1)
```

Example 2: Propagating constant input value

```
C:   q2(X,Z) :- true | append(X,[3,4],Z)
C':  q2(X,Z) :- true | g1(X,Z)
D:   { append(X,[3,4],Z) -:: g1(X,Z) }
P+:  { g1([H|X],Z) :- true | Z=[H|Zs], g1(X,Zs)
       g1([],Z) :- true | Z=[3,4]                  }
```

Example 3 Propagating constant output value

```
C:   q3(X,Y) :- true | append(X,Y,[1,2])
C':  q3(X,Y) :- true | g1(X,Y)
D:   { append(X,Y,[1,2]) -:: g1(X,Y)
       append(X,Y,[2]) -:: g2(X,Y)   }
P+:  { g1([H|X],Y) :- true | 1=H, g2(X,Y)
       g1([],Y) :- true | [1,2]=Y
       g2([H],Y) :- []=Y | 2=H
       g2([],Y) :- true | [2]=Y            }
```

P := a set of original program clauses
D: a set of goal pattern redefinitions := ϕ
P^+: a set of specialized program clauses := ϕ

Process $SpecClause(C)$:

/* (1) */ apply Rule 1 and 2 to C as far as possible
if the resultant clause: $C_i = \phi$ **then terminate**
else if
  /* (2) */ try to apply Rule 4 (with the help of Rule 3, if needed), and
  if it was applied
  then invoke $SpecClause$ on each resultant clause, and **terminate**
else
  assume $C_i = (H \leftarrow G \mid B)$, and
  for each $B_i \in B$ **do**

    /* (3) */ look into the goal pattern redefinitions, D
    if there exists $(P \stackrel{\sim}{\rightarrow} P') \in D$ such that
      B_i is a variant of P by the renaming, ρ, ie, $B_i = P\rho$
    then add $P'\rho$ into B'

    /* (4) make private clauses for each distinct goal pattern */
    else if there exists non-null set $\{C^j : H^j \leftarrow G^j \mid B^j\} \in P$ such that
      H^j is unifiable with B_i by the substitutions,
      θ^j and σ^j, for H^j and B_i respectively
      and
      /* (5) Guard Evaluation */
        the guard $G^j\theta^j$ is not evaluated as $fail$
    then create a new goal B_i' with a new predicate symbol and
      all distinct variables occurring in B_i, and
      add B_i' into B', and
      add $(B_i \stackrel{\sim}{\rightarrow} B_i')$ into D, and

      for each C^j **do** invoke $SpecClause(C^j\theta^j)$
        add $(H \leftarrow G \mid B')$ into P^+
    else terminate
    /* if all B_i are tried successfully */
    add $C' : (H \leftarrow G \mid B')$ into P^+, and **terminate**

Figure 1 Main process of the partial evaluation algorithm

3.2 Virtual Backward Propagation

Suppose a clause, $(H \leftarrow G \mid B)$, has a pair of body goals, B_i and $B_j (i \neq j)$, such that the defining clauses are

$$H_i^k(\ldots X \ldots) \leftarrow G_i^k \mid (B_i^k \cup \{X = c\}) \quad (1 \leq k \leq N)$$

and,

$$H_j^l(\ldots X \ldots) \leftarrow G_j^l \mid B_j^l \quad (1 \leq l \leq M)$$

respectively, where X is an output variable of B_i, while it is an input variable of B_j. It is often the case that such clauses are found in P^+. Then, whichever clause from $H_i^k (1 \leq k \leq N)$ is commited, the output value for X will always be the same, c. This fact can be used freely in Prolog, and further reduction on all of the caller, producer and consumer clauses is possible [Levi 87]. However, it may not be guaranteed to be correct to export the value of X to B_j because of possible changes of synchronization condition. Even if this is the case, it can be used to find and remove an unsatisfied clause, $(H_j^l \leftarrow G_j^l \mid B_j^l)$, if G_j^l turns out to be unsatisfiable by $X = c$. Such removal of a clause is safe with respect to preservation of synchronization conditions as well as the set of success patterns of goals.

This is called *virtual backward propagation* (of a substitution), and is incorporated into the basic partial evaluation algorithm described in the previous subsection, as a post process for *SpecClause*.

4. Simplification Based on Constraints

4.1 Constraint Propagation

Looking more closely into specialized program P^+, another redundancy may be found. For instance, there may be a clause in P^+ that cannot be called by any goal in P^+. 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.

For example, consider the clauses:

```
(1)   p(X,Y) :- X\=a | q(X), Y=c.
(2)   q(a) :- true | r.
(3)   q(A) :- A\=b | s.
(4)   q(A) :- A\=a | t.
```

The constraint for `q(X)` in (1) is `X\=a`. When (2) is selected, the mgu, {`a/X`}, is inconsistent with the constraint, hence, the clause is never used for solving `p(X,Y)`. The guard, `A\=b` in (3), is not inconsistent with the constraint so far. The guard, `A\=a` in (4), is identical to the constraint, hence, it can be eliminated as far as the clause is exclusive to clause (1). Thus, the clauses are simplified as follows:

```
(1)   p(X,Y) :- X\=a | q(X), Y=c.
(2)       removed
(3)   q(A) :- A\=b | s.
(4')  q(A) :- true | t.
```

Note that Rule 4 may not be applied at clause (1) in the example.

4.2 Constraint Reduction

for each clause $C : (H \leftarrow G \mid B)$ in P^+
  if there exists a constraint, C_c, common to every global constraint for C
  then re-evaluate the guard, G, as well as head unification,
        with constraint C_c, and
      if the clause turns out to be unsatisfied
      then remove it from P^+
      else if the guard G is simplified to G',
          with possible changes from B to B' (via local variables)
      then remove it from P^+, and
          invoke a process $SpecClause(H \leftarrow G' \mid B')$

Figure 2 Use of constraint solving for specialization

In the course of tentative goal expansion, a variable may be constrained to a single possible value under the accumulated constraints so far. If this is the case, then the further specialization of clauses may be triggered.

For example, consider the clauses:

```
(1)   p(X,Y) :- X>0 | q(X), Y=c.
(2)   q(X) :- X=<1 | r(X).
(3)   r(X) :- X<1 | s.
(4)   r(X) :- X>=1 | t.
```

None of the guards in (1) to (4) can be reduced, and each one should remain as it is. However, assume that `q` is called only by `p`, then `X` is constrained to have a value `1` by the conjunction of guards, `X>0` and `X=<1` (`X` is assumed to be an integer). Guard `X=<1` should still be there in clause (3); however, the fact that `X` should be `1` can be used in the body, `r(X)`. Thus, in (2), the body can be replaced by `r(1)`. Then clause (3) turns out to be unsatisfied, while clause (4) is immediately executable. The result after specialization process on clause (2) is

```
(1)   p(X,Y) :- X>0 | q(X), Y=c.
(2')  q(X) :- X=<1 | t.
```

4.3 Constraint Solving for Specialization

To be more formal, more concrete definitions of constraint are given, and the algorithm to use the constraint for specialization in the way described above is shown.

Definition : *atomic constraint*

An atomic constraint is an atom of a special predicate in a predefined set. Practically, the set is of some built-in predicates for GHC, say, $\{\neq, <, \leq, >, \geq, :=\}$.

Definition : *local constraint* for a goal

Let a clause, C, be $(H \leftarrow G \mid B)$. For a guard, $G_i \in G$, the conjunction of atomic constraints from the rest of the guards $G_j (j \neq i) \in G$ is called the local constraint for G_i. For a body goal, $B_i \in B$, the conjunction of atomic constraints from guard G and those from the body goals, $B_{j \neq i} \in B$, is called the local constraint for B_i.

Definition : *C-graph*

C-graph is defined by a set of nodes, N, and a set of arcs, A, where each node in N is for a clause, $(H \leftarrow G \mid B)$, and there is a directed arc in A from each body goal, $B_i \in B$, to another node (a clause) in N of which head is the same predicate as B_i.

Definition : *global constraint* for a clause

Let S be a source node in N of a C-graph, and D be a destination node (target clause) in N, then each path from S down to D, which is a sequence of arcs in A of the C-graph, is called a global constraint for the target clause.

Then, the use of constraint solving for specialization is sketched as in Fig.2.

4.4 Example: String Matching

Consider the program for string matching:

```
P:  { match(P,T) :- true | match1(P,T,P,T)
      match1([A|Ps],[A|Ts],P,T) :- true | match1(Ps,Ts,P,T)
      match1([A|_],[B|_],P,[_|T]) :- A\=B | match1(P,T,P,T)
      match1([],_,_,_) :- true | true                          }
```

The first argument of `match`, `P`, is a pattern string which is matched against the second argument, `T`, a target string.

Now, suppose that the pattern is fixed to the list `[a,a,a,b]`. Then the *NewClause* process will terminate with the following result without the help of the constraint solver.

```
C:  query(T) :- true | match([a,a,a,b],T)
C': query(T) :- true | g1(T)

D:  { match1([a,a,a,b],A,[a,a,a,b],A) -:: g1(A)
      match1([a,a,b],A,[a,a,a,b],[a|A]) -:: g2(A)
      match1([a,b],A,[a,a,a,b],[a,a|A]) -:: g3(A)
      match1([b],A,[a,a,a,b],[a,a,a|A]) -:: g4(A)
      match1([a,a,a,b],[a,a,A|B],[a,a,a,b],[a,a,A|B]) -:: g5(A,B)
      match1([a,a,b],[A|B],[a,a,a,b],[a,A|B]) -:: g6(A)
      match1([a,a,a,b],[A|B],[a,a,a,b],[A|B]) -:: g7(A)          }

P+: { g1([a|A]) :- true | g2(A)          g5(a,A) :- true | g4(A)
      g1([A|B]) :- a\=A | g1(B)          g5(A,B) :- a\=A | g6(A,B)
      g2([a|A]) :- true | g3(A)          g6(a,A) :- true | g3(A)
      g2([A|B]) :- a\=A | g7(A,B)        g6(A,B) :- a\=A | g7(A,B)
      g3([a|A]) :- true | g4(A)          g7(a,A) :- true | g2(A)
      g3([A|B]) :- a\=A | g6(A,B)        g7(A,B) :- a\=A | g1(B)
      g4([b|A]) :- true | true
      g4([A|B]) :- b\=A | g5(A,B)                                }
```

There are seven redefinitions of goal patterns, each of which corresponds to a distinct goal pattern that will appear in computation under the top-level query, `match([a,a,a,b],T)`.

Looking into P^+, it is found easily that every call of g6(A,B) is constrained as a\=A, and that it is inconsistent with the head unification for the first clause of g6, while the guard in the second clause of g6 becomes identical after head unification. Hence, the first clause of g6 is unsatisfiable, while the second is satisfied. Thus, every call of g6(A,B) becomes immediately executable. The same thing is found for g7. After applying Rule 2 at the clauses which contain g6 or g7, P^+ is simplified as:

```
P+': { g1([a|A]) :- true | g2(A)          g5(a,A) :- true | g4(A)
       g1([A|B]) :- a\=A | g1(B)          g5(A,B) :- a\=A | g1(B)
       g2([a|A]) :- true | g3(A)
       g2([A|B]) :- a\=A | g1(B)
       g3([a|A]) :- true | g4(A)
       g3([A|B]) :- a\=A | g1(B)
       g4([b|A]) :- true | true
       g4([A|B]) :- b\=A | g5(A,B)                                    }
```

The partially evaluated program behaves as the Knuth-Morris-Platt algorithm does when pattern string contains a run of identical elements. In general, in the original program, no matter what the pattern is, if the target has an unmatching prefix of length M, then unsuccessful elementwise matching is repeated M times, while, in the specialized program, if x is repeated N times in the pattern, and the target contains a run of x, repeated L times within its unmatching prefix of length M, $min(L, N)$ times of elementwise matching for x is saved.

4.5 Example: McCarthy's 91-function

McCarthy's 91-function is the least fixpoint of the following non-linear recursive functional equation for integers as its domain.

$$m91(X) = \textit{if } X > 100 \textit{ then } X - 10 \textit{ else } m91(m91(X + 11))$$

Consider the GHC clauses corresponding to the above definition, in the domain of positive integers.

```
m91(X,Y) :- X>100 | Y := X-10.
m91(X,Y) :- X>0, X=<100 | W := X+11, m91(W,Z), m91(Z,Y).
```

After the process roughly shown in Fig.3, the following clauses are obtained.

```
m91(A,B) :- A>100 |B := A-10.
m91(100,B) :- true | B := 91.
m91(99,B) :- true | B := 91.
...
m91(1,B) :- true | B := 91.
```

5. Relation to Other Research

This work was inspired by the work of [Futamura 87] for functional programs. He considered that unevaluable residual conditional tests, obtained in the *if-then-else* construct, should be used at most in the expansion of each conditional branch. In his

```
{} m91(A,B) ?1
   --> (1.1) {} A>100 ? ==> A>100 ..residual
       (1.2) {A>100} B := A-10 ? ==> B := A-10 ..residual
   ==> m91(A,B) :- A>100 | A := B-10 ..residual clause

?1 --> (2.1) {A=<100} A>0 ? ==> A>0 ..residual
       (2.2) {A>0} A=<100 ? ==> A=<100 ..residual
       (2.3) {A>0, A=<100} D := A+11 ? ==> D := A+11 ..residual
       (2.4) {A>0, A=<100, D := A+11} m91(D,C) ?2
             --> (1.1) {A>0, A=<100, D := A+11} D>100 ? ==> D>100 ..residual
                 (1.2) {A>0, A=<100, D := A+11, D>100} C := D-10 ?
                     = {A>89, A=<100, D := A+11} C := D-10 ?
                       ==> C := D-10 ..residual
       (2.5) {A>0, A=<100, D := A+11, D>100, C := D-10} m91(C,B) ?3
           = {A>89, A=<100, C := A+11-10} m91(C,B) ?3
             --> (1.1) {A>89, A=<100, C := A+11-10} C>100 ?
                     = {A>89, A=<100, A+11-10 > 100} C>100 ?
                     = A=100 ..solved
                 (1.2) {A=100, C := A+11-10} B := C-10 ?
                       ==> B := 100+11-10-10 = 91 ..residual
             ==> m91(100,B) :- true | B := 91 ..residual clause

          ?3 --> (2.1) {A>89, A=<100, C := A+11-10} C>0 ? ==> C>0 ..residual
                 (2.2) {A>89, A=<100, C := A+11-10} C=<100 ?
                     = {A>89, A=<100, A+11-10 =< 100} C=<100 ?
                     = {A>89, A=<99} C=<100 ? ==> C=<100 ..residual
                 (2.3) {A>89, A=<99, C := A+11-10} D1 := C+11 ?
                       ==> D1 := C+11 ..residual
                 (2.4) {A>89, A=<99, D1 := A+11-10+11} m91(D1,C1) ?4
                       --> (1.1) {A>89, A=<99, D1 := A+11-10+11} D1>100 ?
                                 ==> D1>100 ..residual
                           (1.2) {A>89, A=<99, D1 := A+11-10+11} C1 := D1-10 ?
                                 ==> C1 := D1-10 ..residual
                 (2.5) {A>89, A=<99, C1 := A+11-10+11-10} m91(C1,B) ?5
                       --> (1.1) {A>89, A=<99, C1 := A+11-10+11-10} C1>100 ?
                                   = {A<89, A=<99, A+11-10+11-10 >100 } C1>100 ?
                                   = A=99 ..solved
                           (1.2) {A=99, C1 := A+11-10+11-10} B := C1-10 ?
                                 ==> B := 99+11-10+11-10-10 = 91 ..residual
                       ==> m91(99,B) :- true | B := 91 ..residual clause

          ?5 --> ...
```

Figure 3 Derivation of McCarthy's 91-function

partial evaluation, called β, a certain theorem prover is used to reduce conditional tests during successive expansion of *if-then-else* in depth. A similar idea is realized in logic languages [Fujita 87b], even more naturally and in a more generalized manner by using the notion of constraint solving.

This research seems to be closely related to CLP [Jaffar 87]. However, the definition of the constraints and their handling are different from ours. The difference comes from

not only the base language syntax and semantics, but also the use of constraint; it is viewed as an answer in real execution of CLP, while it is just a residual goal in our case and it still needs to be solved or evaluated at execution time. Nonetheless, CLP and our method are closely related. That is, some constraint solving techniques used in CLP may be applied to our method, and vice versa. Moreover, the two may be unified, and the partial evaluation of CLP programs may be introduced in a very natural way.

In [Gallagher 87], a method of partial evaluation for FCP programs is given. It seems to be a very promising idea to use abstract interpretation as a formal basis of partial evaluation. However, it is not very clear, by now, how much improvement can be obtained by the method applied to some real applications. The idea of using constraint solvers may be applied also to their framework.

There is another view of combining or unifying concurrent logic programming and constraint logic programming [Saraswat 87]. This suggests that all of these relatively orthogonal ideas and techniques about concurrency, constraint and partial evaluation may neatly fit together in the same logic programming framework.

6. Conclusion

This paper described a method of partial evaluation of GHC programs. A given program is specialized with respect to some specific query pattern, by performing reductions as much as possible on the basis of immediate execution defined in the UR-set together with some constraint solver. The degree of performance improvement of the resultant program depends on the given program and partial information in the query pattern. In the worst case, the resultant program will be identical to the original program, although the initial steps in the execution may be saved. However, there is a possibility of performance being drastically improved, if the partial evaluator is equipped with an approapriate constraint solver.

The method presents only a general framework; any kind of constraint, its domain and its solver, can be considered. Constraint solving can be done at any degree of eagerness. The stronger the solver 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 solver cannot be predetermined, because it depends solely on the input data partially given as well as the subject program. Extensive work on real and typical GHC progams should be required to prove the method being really a useful framework.

References

[Fujita 87a] Fujita, H. and Furukawa, K., A Self-applicable Partial Evaluator and Its Use in Incremental Compilation, ICOT TR-258, 1987

[Fujita 87b] Fujita, H., An Algorithm for Partial Evaluation with Constraints, ICOT TM-367, 1987

[Fuller 86] Fuller, D.A. and Abramsky, S., Mixed Computation of Prolog Programs, Technical Report, Dept. of Computing, Imperial College of Science and Technology, London, 1986

[Furukawa 87] Furukawa, K. and Okumura A., Unfolding Rules for GHC Programs, ICOT TR-277, 1987

[Futamura 87] Futamura, Y., Generalized Partial Computation, in US-Japan AI Symposium 87

[Gallagher 87] Gallagher, J. and Codish, M., Specialisation of Prolog and FCP programs, in Proc. of Workshop on Partial Evaluation and Mixed Computation, Gl. Avernaes, Denmark, 1987

[Jaffar 87] Jaffar, J. and Lassez, J.-L., Constraint Logic Programming, in Proc. 14th ACM POPL Conf., Munich, 1987

[Levi 86] Levi, G., Object Level Reflection of Inference Rules by Partial Evaluation (extended abstract), in P. Maes and D. Nardi, eds., Workshop on Meta-Level Architectures and Reflection, Sardinia, 1986

[Levi 87] Levi, G., 1987, *private communication*

[Safra 86] Safra, S. and Shapiro, E., Meta Interpreters for Real, in Information Processing 86, Dublin, Ireland, 271-278, North-Holland, 1988

[Saraswat 87] Saraswat, V.A., CP as a general-purpose constraint-language, AAAI-87, 1987

[Takeuchi 86] Takeuchi, A. and Furukawa, K., Partial Evaluation of Prolog Programs and Its Application to Meta Programming, in Information Processing 86, Dublin, Ireland, 415-420, North-Holland, 1986

[Ueda 85] Ueda, K., Guarded Horn Clauses, in Proc. Logic Programming '87, 1986