TR-386

FGHC Partial Evaluator as a General Purpose Parallel Compiler

by
H. Fujita

May, 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

FGHC Partial Evaluator as a General Purpose Parallel Compiler

Hiroshi FUJITA

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

Abstract: A partial evaluator for FGHC programs has been implemented in FGHC. The partial evaluator is based on *UR-set* with constraint solving and is as powerful as Futamura's β for *generalized partial computation*. The partial evaluator succeeded to specialize programs with an order of magnitude improvement of runtime efficiency. Compiling programs with respect to its interpreter (Futamura's *first projection*) is the most promising application of the partial evaluator. The compiled programs automatically obtain opportunities of parallel execution even if the source program is written without concern of parallelism. In particular, functional programs like Backus' FP are easiest to deal with. Compiling Prolog programs is also described. An Prolog interpreter that gives *all solutions* for a Prolog source program is defined in FGHC. According to *Layered Stream* programming style, the Prolog interpreter can attain maximal parallelism, hence the compiled program will obtain good performance as well.

1. Introduction

Partial evaluator is used as a *specializer* for a general purpose program with partial information given and fixed prior to an execution of the program. In particular, one of the most promising application of the specializer is using it as a *collapser* of interpretation layers within programs written in meta-programming style, where the interpreter is specialized with respect to its fixed subject program. With regard to this, there is an excellent relation between partial evaluation and compilation known as Futamura's projection [Futamura 71]:

$PE(\overset{+}{Int}, \overset{+}{Source}, \overset{-}{Target})$ ... First projection (compilation)

$PE(\overset{+}{PE}, \overset{+}{Int}, \overset{-}{Com})$ ... Second projection (compiler generation)

$PE(\overset{+}{PE}, \overset{+}{PE}, \overset{-}{Cocom})$ ... Third projection (compiler-compiler generation)

This suggests that we are able to obtain a very general purpose compiler generation tool by realizing a self-applicable partial evaluator. Toward this ultimate goal, many researchers have been involved in developing partial evaluators within various languages. However, to date, results are restricted to a few languages, especially Lisp [Jones 85] and Prolog [Fujita 87 and Fuller 87], both for sequential programs. On the other hand, research and practice of partial evaluation for parallel programs are only recent [Safra 86 and Gallagher 87] (for FCP).

On the basis of the background and several experiences with Prolog partial evaluation, a partial evaluator for FGHC programs has been developed [Fujita 88]. It is written in FGHC, hence it is self-applicable in principle, although the *Com* and *Cocom* have not yet been obtained at present. The partial evaluator is based on a set of sound unfolding rules called *UR-set* for GHC programs [Furukawa 87], and constraint solving mechanism that makes the partial evaluator as powerful as Futamura's β for *generalized partial computation* [Futamura 87].

The partial evaluation algorithm for FGHC programs is overviewed in Section 2. In Section 3, compilation of Backus' FP programs to FGHC is described. In section 4, a Prolog all-solution interpreter is defined and compilation of Prolog programs to FGHC is described.

2. Overview of the Partial Evaluator

Informally, the essence of the partial evaluation algorithm is to collect all instances of clauses within a given program required to solve a specified query pattern. Some literals within an instantiated clause may be immediately evaluated or resolved with other clauses, other literals must remain as they are in the clause. The evaluation or resolution of a literal may cause further generation of more instantiated clauses. By providing appropriate criteria for termination condition, all of the evaluation, resolution, and instantiation of clauses will terminate with a finite set of instantiated clauses called *residual program* for the given program and query specification.

In the following subsections, the algorithm is briefly formulated.

2.1 Canonical Goal and Program

For the most general pattern of a query, that is, a query with distinct variables as its arguments, we need its defining clauses within the given program in their original form. On the other hand, if the query has static arguments, ie. non-variable term or multiple occurences of a variable at more than one argument positions, its defining clauses can be instantiated according to the static arguments by head unification with the query. Due to the GHC synchronization rule [Ueda 85], this information passing is uni-directional: from the query to the head of a clause. If the query has no remaining variable, the information passing can be accomplished at PE-time and overhead at runtime concerning head unification is saved. Otherwise, it is necessary and sufficient to pass only values bound for the remaining variables at runtime. To save head unification for static arguments at runtime, a new predicate is introduced for the special pattern of a query and corresponding clauses instantiated for the query.

Definition (canonical goal)

Let $Q = p(t_1, \ldots, t_n)$ be a goal. $\overline{Q} = \overline{p}(v_1, \ldots, v_m)$ is a *canonical goal* for Q if $\overline{p} \neq p$ and $\{v_i\}(1 \leq i \leq m)$ are all the distinct variables appearing in $\{t_j\}(1 \leq j \leq n)$.

Example $\overline{append}(X, Z)$ is a canonical goal for $append([1,2|X],[end],Z)$.

Definition (canonical program)

Let Q be a goal, $\mathbf{P}$ its program. $\overline{\mathbf{P}_Q}$ is a *canonical program* for $\mathbf{P}$ wrt Q if for any valuation of $Vars(Q)(= Vars(\overline{Q}))$, ie. variables in Q, $[\![\overline{Q}, \overline{\mathbf{P}_Q}]\!]$ is equivalent under the given semantics to $[\![Q, \mathbf{P}]\!]$.

Example Given a pair of a goal and a program for "append":

```
[  append(X,[end],Z),
  { (append([H|X],Y,Z) :- true | Z=[H|Z1], append(X,Y,Z1)),
    (append([],Y,Z) :- true | Z=Y) }  ]
```

a pair of the canonical goal and its program equivalent to the original is given as:

$$\begin{array}{l}
[\![\ \ \overline{\mathtt{append}}\mathtt{(X,Z)},\\
\quad \{\ \mathtt{(}\overline{\mathtt{append}}\mathtt{([H|X],Z)\ :\!-\ true\ |\ Z=[H|Z1],\ }\overline{\mathtt{append}}\mathtt{(X,Z1))},\\
\quad\quad \mathtt{(}\overline{\mathtt{append}}\mathtt{([],Z)\ :\!-\ true\ |\ Z=[end])}\ \}\ \ ]\!]
\end{array}$$

Head unification (and unification in a guard) in GHC is restricted so as not to bind variables on the query side. Several notions and notations are introduced for clarifying this specific nature of GHC unification.

Definition (idmgu)

Let E_1 and E_2 be expressions, θ, σ, and τ be substitutions. θ is called *idempotent most general unifier* (idmgu) of E_1 and E_2 if

$$\forall\sigma\exists\tau \quad E_1\sigma = E_2\sigma \supset \sigma = \theta\tau \qquad and \qquad \theta\theta = \theta.$$

```
Algorithm 1    (canonical query and program)

INPUT:   a pair of a goal and its program, $[\![Q_0, \mathbf{P}]\!]$
OUTPUT:   a pair of a canonical goal and its program, $[\![\overline{Q_0}, \overline{\mathbf{P}_{Q_0}}]\!]$

$\mathbf{T} := \phi$;    $\mathbf{R} := \phi$;   /* P, T, and R are global for function $Alg1$. */

recursive function $Alg1(Q)$
  if $Q$ is of a builtin predicate then return $Q$
  else if there is a pair, $[\overline{Q_i}, Q_i] \in \mathbf{T}$ such that $Q$ is a variant of $Q_i$
     then return $\overline{Q_i}\langle Q_i|Q\rangle\downarrow_{Vars(Q)}$
     else begin
           let $\overline{Q}$ be a canonical goal for $Q$ with a new predicate symbol;
           $\mathbf{T} := \mathbf{T} \cup \{[\overline{Q}, Q]\}$;
           for each $(H_i :- \mathbf{G}_i|\mathbf{B}_i) \in \mathbf{P}$ such that $Q$ and $H_i$ are unifiable
             begin
               $\theta_i := \langle H_i|Q\rangle\downarrow_{Vars(Q)}$;
               for each $B_{ij} \in \mathbf{B}_i$ do $\overline{B_{ij}} := Alg1(B_{ij}\theta_i)$;
               $\mathbf{R} := \mathbf{R} \bigcup \{(\overline{Q}_i\theta_i :- \mathbf{G}\theta_i|\{\overline{B_{i1}}, \ldots, \overline{B_{in_i}}\})\}$
             end ;
           return $\overline{Q}$ end ;

$\overline{Q_0} := Alg1(Q_0)$;   $\overline{\mathbf{P}_{Q_0}} := \mathbf{R}$ .
```

Figure 1 The basic algorithm for canonical query and program

We denote idmgu of two expressions, E_1 and E_2, as $\langle E_1|E_2\rangle$.

Definition (V-minimal substitution)

Let $\sigma = \{V_i \leftarrow T_i\}$ be a substitution. For any set of variables, $\mathbf{V}$, σ is called $\mathbf{V}$-*minimal* if

$$\forall (V_i \leftarrow V_j)_{(i \neq j)} \in \sigma \quad V_j \in \mathbf{V} \supset V_i \in \mathbf{V}$$

Proposition (existance of V-minimal idmgu [Ueda 88])

For any set of variables, $\mathbf{V}$, there exists $\mathbf{V}$-minimal idmgu of E_1 and E_2 if expressions E_1 and E_2 are unifiable.

We denote V-minimal idmgu of two expressions, E_1 and E_2, as $\langle E_1|E_2\rangle\downarrow_{\mathbf{V}}$.

Now we show the basic algorithm in Fig.1 that gives a canonical query and corresponding canonical program for a given query pattern and its program.

Algorithm 1 terminates, since there are only finite number of distinct goal patterns, $\{B_{ij}\}$, appearing in finite set of clauses, $\mathbf{P}$. The number of pairs in $\mathbf{T}$ increases as a recursive call of *Alg*1 is invoked, and $\mathbf{T}$ will eventually saturate with finite number of pairs for distinct goal patterns. After that, no recursive call of *Alg*1 will be invoked.

2.2 Normalization and Unfolding

In this subsection, Rule-1 and Rule-2 of UR-set [Furukawa 87] is reformulated using the notion of V-minimal idmgu as in [Ueda 88].

Definition (normalization or Rule-1 of UR-set)

Let $Cl = (H :-\mathbf{G}|\mathbf{B})$ be a clause, $\mathbf{G}_u$ a set of unification atoms in $\mathbf{G}$ and $\mathbf{G}_n = \mathbf{G} \setminus \mathbf{G}_u$, $\mathbf{B}_u$ a set of unification atoms in $\mathbf{B}$ and $\mathbf{B}_n = \mathbf{B} \setminus \mathbf{B}_u$.

A clause $\underline{Cl} = (H :-\underline{\mathbf{G}_u} \cup \underline{\mathbf{G}_n} \mid \underline{\mathbf{B}_u} \cup \underline{\mathbf{B}_n})$ is a *normalized clause* for Cl if

$$\begin{aligned}
\underline{\mathbf{G}_u} &= UnifGoal([Idmgu(\mathbf{G}_u)\downarrow_{Vars(H)}]_{Vars(H)}) \\
\underline{\mathbf{B}_u} &= UnifGoal([Idmgu(\mathbf{G}_u \cup \mathbf{B}_u)\downarrow_{Vars(H)}]_{Vars(H)}) \\
\underline{\mathbf{G}_n} &= \mathbf{G}_n Idmgu(\mathbf{G}_u)\downarrow_{Vars(H)} \\
\underline{\mathbf{B}_n} &= \mathbf{B}_n Idmgu(\mathbf{G}_u \cup \mathbf{B}_u)\downarrow_{Vars(H)}
\end{aligned}$$

where $Idmgu(\mathbf{U})$ gives the idmgu for a set of unification goals, $\mathbf{U}$, $[\sigma]_{\mathbf{V}}$ denotes the projection of a substitution, σ, on a set of variables, $\mathbf{V}$, and $UnifGoal(\sigma)$ gives the unification goal(s) for a substitution, σ.

Definition (unfolding at an immediately executable goal or Rule-2 of UR-set)

Let Q be a goal, $\mathbf{C}$ a set of clauses. $\mathbf{C}_Q^+$ is a set of *satisfied* clauses, $\mathbf{C}_Q^?$ a set of *candidate* clauses for $\mathbf{C}$ wrt Q if

$$\begin{aligned}
&\forall (H_i :-\mathbf{G}_i|\mathbf{B}_i) \in \mathbf{C}_Q^+ \\
&\qquad \exists \sigma \quad \sigma = \langle H_i|Q\rangle\downarrow_{Vars(Q)} \quad and \quad Q\sigma = Q \quad and \quad \mathbf{G}_i\sigma \Rightarrow true
\end{aligned}$$

and

$$\begin{aligned}
&\forall (H_i :-\mathbf{G}_i|\mathbf{B}_i) \in \mathbf{C}_Q^? \\
&\qquad \exists \sigma \quad \sigma = \langle H_i|Q\rangle\downarrow_{Vars(Q)} \quad and \quad \mathbf{G}_i\sigma \not\Rightarrow fail
\end{aligned}$$

If there is no set of candidate clauses in the program $\mathbf{P}$ wrt a goal Q, ie. $\mathbf{P}_Q^? = \phi$, Q is called *immediately executable*.

If goal Q is immediately executable, Q is expanded to a set of *don't care nondeterministic* alternative goals, $\{\mathbf{B}_i\theta_i\}$, for $\mathbf{C}_G^+ = \{H_i : -\mathbf{G}_i|\mathbf{B}_i\}$ and $\theta_i = \langle H_i|Q\rangle\downarrow_{Vars(Q)}$.

Rule-4 of UR-set is called *unfolding across guard* that is essentially *case splitting*. This, of cource, is a effective rule for the partial evaluator to perform further specialization. However, in this paper, this rule (and Rule-3) is ommitted since the examples described later show satisfactory results only with Rule-1 and Rule-2.

```
**Algorithm 2**    (normalized and reduced canonical program)

INPUT:  a pair of a goal and its program, $[Q_0, \mathbf{P}]$
OUTPUT:  a pair of a canonical goal and its normalized and reduced
         canonical program, $[\overline{Q_0}, \underline{\overline{\mathbf{P}_{Q_0}}}]$

$\mathbf{T} := \phi$;    $\mathbf{R} := \phi$;
/* $\mathbf{P}, \mathbf{T}$, and $\mathbf{R}$ are global for functions $Alg2, Alg2^{+}, Alg2^{*}, Alg2^{+?}$, and
  a procedure, $Alg2^{+?*}$. */

recursive function $Alg2(Q/\mathbf{C})$
  if $Q$ is of a builtin predicate then return $\{Q\}$
  else if there is a pair, $[\overline{Q_i}, Q_i] \in \mathbf{T}$ such that $Q$ is a variant of $Q_i$
    then return $\{\overline{Q_i}(Q_i|Q)\downarrow_{Vars(Q)}\}$
    else begin
        let $\mathbf{P}^{+}_{Q/\mathbf{C}}$ be a set of satisfied clauses of $\mathbf{P}$ wrt $Q$ under constraint $\mathbf{C}$,
            $\mathbf{P}^{?}_{Q/\mathbf{C}}$ a set of candidate clauses of $\mathbf{P}$ wrt $Q$ under constraint $\mathbf{C}$;
        if $\mathbf{P}^{?}_{Q/\mathbf{C}} = \phi$ then return $Alg2^{+}(Q/\mathbf{C}, \mathbf{P}^{+}_{Q/\mathbf{C}})$
        else return $Alg2^{+?}(Q/\mathbf{C}, \mathbf{P}^{+}_{Q/\mathbf{C}} \cup \mathbf{P}^{?}_{Q/\mathbf{C}})$
        end ;

$\mathbf{Q}_0^{*} := Alg2(Q_0/true)$;
if $\mathbf{Q}_0^{*} = \{\overline{Q}\}$ then $\overline{Q_0} := \overline{Q}$
else begin
  let $\overline{Q_0}$ be a canonical goal for $Q_0$ with a new predicate symbol;
  for each $\mathbf{Q}_{0i} \in \mathbf{Q}_0^{*}$ do $Alg2^{+?*}((\overline{Q_0} :\!-true|\mathbf{Q}_{0i}))$;
  end ;
$\underline{\overline{\mathbf{P}_{Q_0}}} := \mathbf{R}$;
```

Figure 2-1   The partial evaluation algorithm

2.3 Using Constraints

So far, we are concerned only with variable binding and passing it from a query down to subgoals. It is possible to take into account constraints other than bindings for variables in a query. This idea and demonstration of its effects on order changing impovement of programs have been reported in [Fujita 88]. The constraint solving for a query is exploited within guard evaluation of a clause in Rule-2 that decides whether the clause is satisfied, unsatisfied or candidate.

```
function $Alg2^{+}(Q/\mathbf{C}, \mathbf{P}^{+}_{Q/\mathbf{C}})$
  for each $(H_i :- \mathbf{G}_i|\mathbf{B}_i) \in \mathbf{P}^{+}_{Q/\mathbf{C}}$ begin
    $\theta_i := \langle H_i|Q\rangle \downarrow_{Vars(Q)}$;
    let $(\underline{H_i} :- true|\underline{\mathbf{B}_i})$ be a clause obtained by
          applying Rule-1 (normalization) on $(H_i :- \mathbf{G}_i|\mathbf{B}_i)\theta_i$;
          /* $(\mathbf{G}_i/\mathbf{C})\theta_i$ evaluates to $true$ */
    $\mathbf{B}^{*}_i := Alg2^{*}(\underline{\mathbf{B}_i}/\mathbf{C}\theta_i)$
    end ;
  return $\bigcup_i \mathbf{B}^{*}_i$;

function $Alg2^{*}(\mathbf{B}/\mathbf{C})$
  for each $B_i \in \mathbf{B}$ do $\mathbf{B}^{*}_i := Alg2(B_i/\mathbf{C})$;
  return $\bigotimes_i \mathbf{B}^{*}_i$;

function $Alg2^{+?}(Q/\mathbf{C}, \mathbf{P}^{+?}_{Q/\mathbf{C}})$
  let $\overline{Q}$ be a canonical goal for $Q$ with a new predicate symbol;
  $\mathbf{T} := \mathbf{T} \cup \{[\overline{Q}, Q]\}$;
  for each $(H_i :- \mathbf{G}_i|\mathbf{B}_i) \in \mathbf{P}^{+?}_{Q/\mathbf{C}}$ begin
    $\theta_i := \langle H_i|Q\rangle \downarrow_{Vars(Q)}$;
    $\mathbf{G}'_i := Red((\mathbf{G}_i \wedge \mathbf{C})\theta_i)$;
    let $\underline{Cl_i}$ be a clause obtained by
          applying Rule-1 (normalization) on $(H_i :- \mathbf{G}'_i|\mathbf{B}_i)\theta_i$;
    $Alg2^{+?*}(\underline{Cl_i})$
    end
  return $\{\overline{Q}\}$;

recursive procedure $Alg2^{+?*}((H :- \mathbf{G}|\mathbf{B}))$
  $\mathbf{B}^{*} := Alg2^{*}(\mathbf{B}/\mathbf{G})$;
  if $\mathbf{B}^{*} = \{\mathbf{B}\}$ then    $\mathbf{R} := \mathbf{R} \cup \{(H :- \mathbf{G}|\mathbf{B})\}$
  else for each $\mathbf{B}_i \in \mathbf{B}^{*}$ begin
    let $\underline{Cl_i}$ be a clause obtained by
          applying Rule-1 (normalization) on $(H :- \mathbf{G}|\mathbf{B}_i)$;
    $Alg2^{+?*}(\underline{Cl_i})$
    end .
```

Figure 2-2 The partial evaluation algorithm (continued)

Combining all of these features, the extended algorithm to obtain normalized and reduced canonical program is given in Fig.2-1 and Fig.2-2.

In $Alg2^{+?}$, $Red(\mathbf{C})$ is a function that returns a reduced set of guard goals (constraints) for **C**. The operators, $\bigcup$ and $\bigotimes$, denote "addition" and "multiplication" of sets of sets of goals defined as:

$$\bigcup_i \mathbf{B}_i^* = \{\mathbf{B}_{ij} | \ \mathbf{B}_{ij} \in \mathbf{B}_i^*\} \qquad \bigotimes_i \mathbf{B}_i^* = \left\{ \bigcup_i \mathbf{B}_{ij_i} \middle| \mathbf{B}_{ij_i} \in \mathbf{B}_i^* \right\}$$

$$where \quad \mathbf{B}_i^* = \{\mathbf{B}_{i1}, \mathbf{B}_{i2}, \ldots, \mathbf{B}_{in_i}\}$$

Algorithm 2 may not terminate, since there may be infinite number of distinct goal patterns, $\{B\}$, due to application of Rule-2 (unfolding). The number of pairs in **T** may increase without limit as a recursive call of $Alg2^{+?}$ is invoked. Therefore, some extra mecahnism ensuring the algorithm termination is required. Further dicussion is given later.

3. Compiling FP programs to FGHC

In this section, it is shown that Backus' FP programs are easily compiled to FGHC programs by the partial evaluator, and that parallelism can be automatically attained for FP programs if the interpreter is written to exploit parallelism in evaluating FP programs.

3.1 FP Interpreter

Backus' FP program [Backus 78] comprises a set of unary functions. No variable is allowed in a user function definition. The domain of FP functions is either primitive data such as integers, atomic symbols, or sequences whose elements are primitive data or sequences. Some of the primitive functions are defined as follows:

bottom: $bottom : X = \bot$

identity: $id : X = X$

equality: $eq : X = \begin{cases} true & \text{if } X = [Y, Y]; \\ false & \text{if } X = [Y, Z] \text{ and } Y \neq Z; \\ \bot & \text{otherwise.} \end{cases}$

subtract one: $sub1 : X = \begin{cases} X - 1 & \text{if } X \text{ is an integer;} \\ \bot & \text{otherwise.} \end{cases}$

multiplication: $times : X = \begin{cases} Y * Z & \text{if } X = [Y, Z] \text{ and} \\ & \quad \text{both } Y \text{ and } Z \text{ are integers;} \\ \bot & \text{otherwise.} \end{cases}$

where ':' is the operator for function application.

FP system also provides some functionals that allow users to define composite programs with primitive functions. Some of them are defined as follows:

$$\begin{aligned}
\text{Composition:}\quad & (F_1 * F_2) : X = F_1 : (F_2 : X) \\
\text{Construction:}\quad & [F_1, \ldots, F_n] : X = \begin{cases} [(F_1 : X), \ldots, (F_2 : X)] & \text{if } \forall i\ (F_i : X) \neq \bot; \\ \bot & \text{otherwise.} \end{cases} \\
\text{Conditional:}\quad & if(F_1, F_2, F_3) : X = \begin{cases} F_2 : X & \text{if } (F_1 : X) = true; \\ F_3 : X & \text{if } (F_1 : X) = false; \\ \bot & \text{otherwise.} \end{cases} \\
\text{Constant:}\quad & \#C : X = \begin{cases} C & \text{if } X \neq \bot; \\ \bot & \text{otherwise.} \end{cases} \\
\text{ApplyToAll:}\quad & @F : X = \begin{cases} [(F : X_1), \ldots, (F : X_n)] & \text{if } X = [X_1, \ldots, X_n] \text{ and} \\ & \quad \forall i\ (F : X_i) \neq \bot; \\ [\,] & \text{if } X = [\,]; \\ \bot & \text{otherwise.} \end{cases}
\end{aligned}$$

The above definitions are coded straightforwardly within FGHC as a FP interpreter for `apply/3` shown in Fig.3.

3.2 Compiling FP Programs

For example, the FP factorial program is given as:

```
def(fact,B) :- true | B=if(eq*[id,#0],#1,times*[id,fact*sub1]).
def(F,B) :- F\=fact | B=bottom.
```

The factorial program is compiled with respect to the FP interpreter. The result is shown in App.1. Because of the very deterministic nature of the program, as is the case with most of functional programs, it might be doubtful whether the compiled code has obtained significant improvement of efficiency or not. However, it is true that the expanded goals in the bodies of `p0/2` and `p1/3` contribute to gain opportunities for parallel execution. Even in the sequential execution, the compiled code runs twice as fast as the original interpreter with the source program does.

Besides the compilation for function part, fixed information on argument part can also be compiled (as in *mixed computation* [Ershov 82]). For example, the next program:

```
def(sub1all,B) :- true | B=(@sub1).
def(F,B) :- F\=sub1all | B=bottom.
```

can be compiled to a FGHC program specialized with respect to an argument pattern of fixed length list, say $[V1, V2, V3]$, as:

$$p1(V1, V2, V3) \equiv apply(sub1all, [V1, V2, V3], Res)$$

```
apply(bottom,_,  Res) :- true | Res=bottom.
apply(id,    Arg,Res) :- true | Res=Arg.

apply(eq,[V,V],Res) :- true       | Res=true.
apply(eq,[U,V],Res) :- U\=V       | Res=false.
apply(eq,Arg,  Res) :- Arg\=[_,_] | Res=bottom.

apply(sub1,Arg,Res) :- integer(Arg)    | Res:=Arg-1.
apply(sub1,Arg,Res) :- noninteger(Arg) | Res=bottom.

apply(times,[X,Y],Res) :- integer(X), integer(Y) | Res:=X*Y.
apply(times,[X,Y],Res) :- noninteger(X)          | Res=bottom.
apply(times,[X,Y],Res) :- noninteger(Y)          | Res=bottom.
apply(times,Arg,  Res) :- Arg\=[_,_]             | Res=bottom.

apply(F1*F2,Arg,Res) :- true | apply(F2,Arg,R2), apply(F1,R2,Res).

apply(if(Cond,Then,Else),Arg,Res) :- true |
  apply(Cond,Arg,RC), i(RC,Then,Else,Arg,Res).

  i(bottom,_,   _,     _,Res) :- true | Res=bottom.
  i(true,  Then,_,   Arg,Res) :- true | apply(Then,Arg,Res).
  i(false, _,   Else,Arg,Res) :- true | apply(Else,Arg,Res).

apply([F1|Fs],Arg,Res) :- true | m(R1,Rs,Res),
                                 apply(F1,Arg,R1), apply(Fs,Arg,Rs).
apply([],       _,Res) :- true | Res=[].

apply(@Fn,[A1|As],Res) :- true | m(R1,Rs,Res),
                                 apply(Fn,A1,R1), apply(@Fn,As,Rs).
apply(@_, [],     Res) :- true | Res=[].
apply(@_, Arg,    Res) :- Arg\=[], Arg\=[_|_] | Res=bottom.

  m(bottom,     _,Res) :- true                         | Res=bottom.
  m(     _,bottom,Res) :- true                         | Res=bottom.
  m(R1,    Rs,    Res) :- R1\=bottom, Rs\=bottom | Res=[R1|Rs].

apply(#_,bottom,Res) :- true       | Res=bottom.
apply(#V,Arg,   Res) :- Arg\=bottom | Res=V.

apply(Fn,Arg,Res) :-
    Fn\=bottom, Fn\=id, Fn\=eq, Fn\=sub1, Fn\=times,
    Fn\=(_*_), Fn\=if(_,_,_), Fn\=[_|_], Fn\=[], Fn\=(@_), Fn\=(#_) |
  def(Fn,Body), apply(Body,Arg,Res).
```

Figure 3 The Backus' FP interpreter in FGHC

```
p1(V1,V2,V3,Res) :- true | m(R1_126,Rs_126,Res), p4(V1,R1_126),
  m(R1_171,Rs_171,Rs_126), p4(V2,R1_171), p3(R1_192,Rs_171), p4(V3,R1_192).

p4(V1,R1_126) :- integer(V1) | R1_126 := V1 - 1.
p4(V1,R1_126) :- noninteger(V1) | R1_126 = bottom.

p3(bottom,Rs_63) :- true | Rs_63 = bottom.
p3(R1_84,Rs_63) :- R1_84 \= bottom | Rs_63 = [R1_84].
```

This demonstrates clearly that parallelism can be obtained very naturally by the compilation.

4. Compiling Prolog programs to FGHC

In this section, a Prolog interpreter is defined in FGHC. Then, it is shown that Prolog programs are compiled with respect to the interpreter using the FGHC partial evaluator.

4.1 Partial Evaluation of Unification

In the Prolog interpreter, a logical variable is represented as a ground term on FGHC level, and unification and substitution application are explicitly performed on the interpreter level rather than implicitly on FGHC level. An enormous overhead will be caused by the explicit unification. Hence, it is the crucial part of the compilation to decompose as much unification tasks as possible to a bunch of FGHC primitives.

The unifier is defined as usual as shown in App.2. For example, a unification goal in a subject program, `unify(f(x!),f(y!),S)`, is partially evaluated to a canonical goal, `p1(S)`, with the residual program for `p1/1` as:

```
p1(S) :- true | S=[x! - y!].
p1(S) :- true | S=[y! - x!].
```

where $x!$ denotes a Prolog variable.

Further, if the unification goal is followed by a minimization goal such as:

```
... unify(f(x!),f(y!),S), minimize(S,[x!],Sm) ...
```

and `S` is not used elsewhere, the pair of goals is reduced to

```
... Sm=[y! - x!] ...
```

without leaving any residual clauses for the original goals. The resultant unification goal will cause further specialization of sibling goals after application of normalization (Rule-1 of UR-set).

```
solve(Q,So) :- true | freshVars(Q,0,Q1), ps(Ps,0), s(Q1/[],Ps,S1),
  varsIn(Q1,QVars), minimizeSS(S1,QVars,S2), splitSS(S2,QVars,So,_).

s(true/Si,    Ps,So) :- true | Ps=[], So=[Si].
s((G1,G2)/Si,Ps,So) :- true | merge(Ps1,Ps2,Ps),
                              s(G1/Si,Ps1,So1), s1(G2/So1,Ps2,So).
s(G/Si,       Ps,So) :- G\=true, G\=(_,_) |
  functor(G,F,N), Ps=[clauses(F/N,Cls)|Ps1], s2(Cls,G,Si,Ps1,So).

  s1(G/[S1|Ss],Ps,So) :- true | merge(Ps1,Ps2,Ps), o(So1,So2,So),
                                s(G/S1,Ps1,So1), s1(G/Ss,Ps2,So2).
  s1(_/[],      Ps,So) :- true | Ps=[], So=[].

  s2([(H:-B)|Cls],G,Si,Ps,So) :- true | merge(Ps1,Ps2,Ps), o(So1,So2,So),
    unify(G,H,U), c(U,Si,S1), derefSubst(S1,S2),
    s3(S2,B,Ps1,So1), s2(Cls,G,Si,Ps2,So2).
  s2([],          _, _,Ps,So) :- true | Ps=[], So=[].
  s2(fail,        _, _,Ps,So) :- true | Ps=[], So=[fail].

    s3(S,   B,Ps,So) :- S\=fail | applySubst(B,S,B1), s(B1/S,Ps,So).
    s3(fail,_,Ps,So) :- true    | Ps=[], So=[].
```

Figure 4-1 Main procedures under solve/2

4.2 All-solution Prolog Interpreter in FGHC

The Prolog interpreter defined here gives a set of all answer substitutions satisfying an input query. The main procedures under solve/2 are shown in Fig.4-1.

A top level query to the interpreter is solve($\overset{+}{Query}$,$\overset{-}{AnswerSubsts}$). For example, the following query:

```
?- solve(append(x!,y!,[1,2]),S).
```

gives a set of all answer substitutions as:

```
S = [[x!-[1,2], y!-[]],
     [x!-[1], y!-[2]],
     [x!-[], y!-[1,2]]]
```

The answer substitutions are collected by the recursive procedure, s/3, for every subgoal for a query to the subject program. a/3 in Fig.4-2 combines sets of sets of substitutions in AND-manner, whereas o/3 combines in OR-manner in a way like a *merge* operator. c/3 combines two sets of substitutions taking care of the consistency of the two sets of substitutions. In this implementation, c/3 reconstructs a single unification goal from the two sets of substitutions, first making a list of variables in the sets as the

```
a([S11|S1s],S2,So) :- true | a1(S11,S2,So1), a(S1s,S2,So2), o(So1,So2,So).
a([],        _,So) :- true | So=[].

  a1(S11,[S21|S2s],So) :- true | o([So1],So2,So),
                                 c(S11,S21,So1), a1(S11,S2s,So2).
  a1(_,  [],       So) :- true | So=[].

o(fail,      S2,        So) :- true      | So=S2.
o(S1,        fail,      So) :- true      | So=S1.
o([fail|S1s],S2,        So) :- true      | o(S1s,S2,So).
o([S11|S1s], S2,        So) :- S11\=fail | So=[S11|So1], o(S1s,S2,So1).
o(S1,        [fail|S2s],So) :- true      | o(S1,S2s,So).
o(S1,        [S21|S2s], So) :- S21\=fail | So=[S21|So1], o(S1,S2s,So1).
o([],        S2,        So) :- true      | So=S2.
o(S1,        [],        So) :- true      | So=S1.

c(fail,   _,So) :- true                 | So=fail.
c(_,   fail,So) :- true                 | So=fail.
c(S1,    S2,So) :- S1\=fail, S2\=fail | c1(S1,S2,L,R), unify(L,R,So).

  c1([V-T|S1],S2,      L,R) :- true | L=[V|L1], R=[T|R1], c1(S1,S2,L1,R1).
  c1(S1,      [V-T|S2],L,R) :- true | L=[V|L1], R=[T|R1], c1(S1,S2,L1,R1).
  c1([],      [],      L,R) :- true | L=[], R=[].
```

Figure 4-2 Procedures combining substitutions

LHS term and a list of substituted terms for the variables as the RHS term. Then, it computes the mgu of the two list.

Source Prolog programs are given in a way shown below:

```
clauses(append/3,Cls) :- true | Cls=[
  (append([],x!,x!) :- true),
  (append([h!|x!],y!,[h!|z!]) :- append(x!,y!,z!))].

clauses(X,Cls) :- X\=append/3 | Cls=fail.
```

These program clauses are retrieved by issuing an `clauses`$(\overset{+}{Pred/Arity},\overset{-}{Clauses})$ message to the program server, `ps`$(\overset{?}{RequestStream},\overset{+}{NextClauseId})$, shown in Fig.4-3. `ps/2` gives an identification number to each requested clause to distinguish variables in the clause from those in other clauses.

4.3 Suspension Control Clauses

The partial evaluator is able to detect a loop when the same pattern (variant) of a goal as previously visited one is about to be partially evaluated, by maintaining and looking up a table dedicated to memoing pairs of goal and its canonical goal. This

```
ps([clauses(Goal,Cls1)|Req],Id) :- true | clauses(Goal,Cls0),
  freshCls(Cls0,Id,Id1,Cls1), ps(Req,Id1).
ps([],                        _) :- true | true.

  freshCls([C1|Cls],Id,Idn,FreshCls) :- true | inc(Id,Id1),
    freshVars(C1,Id1,FreshCl1), FreshCls=[FreshCl1|FreshCls1],
    freshCls(Cls,Id1,Idn,FreshCls1).
  freshCls([],       Id,Idn,FreshCls) :- true | Idn=Id, FreshCls=[].
  freshCls(fail,     Id,Idn,FreshCls) :- true | Idn=Id, FreshCls=fail.

    inc(C,C1) :- integer(C) | C1:=C+1.

freshVars(V!,Id,VI) :- true     | VI=V!Id.
freshVars(T, Id,TI) :- T\=(_!) | functor(T,F,N), rotcnuf(TI,F,N),
                                 freshVars1(N,T,Id,TI).

  freshVars1(N,T,Id,TI) :- N>0  | N1:=N-1, arg(N,T,A), arg(N,TI,B),
    freshVars(A,Id,B), freshVars1(N1,T,Id,TI).
  freshVars1(0,_, _, _) :- true | true.
```

Figure 4-3 Program server

mechanism contributes to ensuring the partial evaluation process termination for a class of goal patterns: the goal of *consumer* type or *passive* type that waits concrete data, and when recursion, arguments to the recursive call are made *decreased* compared to the input data received by the goal. However, it will not work for a *producer* type or *active* type goal, that spontaneously generates data, and when recursion, arguments to the recursive call are made *increased* compared to the input data received by the goal. For instance, a perpetual process that generates integer sequences, defined as:

```
integers([X|Y],N) :- true | X=N, integers(Y,s(N)).
integers([],   _) :- true | true.
```

is producer type, which increments the counter in the second argument according to a request for the next integer value in the stream in the first argument.

In the Prolog interperter, the program server, `ps/2`, is a perpetual process that receives program requests from the solver, `s/3`, incrementing a counter for clause identification. A goal of `ps/2` should be suspended if the request stream is unbound in partial evaluation time. This type of suspenion control is provided by special clauses for `'Suspend'/2`, called *control clauses* for the partial evaluator given as follows:

```
'Suspend'(ps((_'_),_),Com) :- true      | Com=suspend.
'Suspend'(ps(X,     _),Com) :- X\=(_'_) | Com=expand.
```

where *Name'Id* denotes a variable with a print name, *Name*, and an identification number, *Id*.

For a predicate that allows *don't care nondeterminism*, even if it is passive, special attention is required in partial evaluation. That is, the decision which satisfied clause is committed should be done at runtime rather than partial evaluation time (altough this strategy may be too conservative if the don't care nondeterminism *don't care* the commitment even at partial evaluation time.) For instance, we provide control clauses for `merge/3` as:

```
'Suspend'(merge([], _,_),Com) :- true           | Com=expand.
'Suspend'(merge(_, [],_),Com) :- true           | Com=expand.
'Suspend'(merge(X,  Y,_),Com) :- X\=[], Y\=[]   | Com=suspend.
```

The similar control clauses to these are provided for `o/3` in Fig.3-2, since it is also a don't care nondeterministic predicate.

There are other possibilities that cause the partial evaluator to get swamped into useless inifinite computation when the subject program has some recursive constructs. In the Prolog interpreter, in particular, `s_1/4` is a critical predicate that may cause infinite computation expanding a recursive predicate in the source program for the interpreter. For instance, the interpreter will run infinitely opening up recursive calls given the most general pattern of a goal, say `append(x!,y!,z!)`. To avoid this, however, the mechanism using a table for memoing canonical goals will suffice as mentioned above as far as `append/3` is concerned.

There may be other reasons why a goal should be suspended. For each such goal pattern, appropriate suspension control clauses should be given. In the Prolog interpreter, `inc/2` for incrementing a counter for clause identification should be suspended, since it is not known at partial evaluation time how many clauses with distinct identification will be used for a query given at runtime. Hence, we provide the following control clause:

```
'Suspend'(inc(_,_),Com) :- true | Com=suspend.
```

In reality, it is this control clause that, together with the memoing table, leads the partial evaluatior to terminate for the Prolog interpreter with respect to its subject program.

Other goals may always be expanded, thus the following control clause is placed at the end.

```
'Suspend'(P,Com) :- P\=ps(_,_), P\=merge(_,_,_), P\=o(_,_,_), P\=inc(_,_),
    % and P is not any other goal to which control clauses are specified
   | Com=expand.
```

The compiled code for `append/3` is shown in App.3. The code shows that many of the subpredicates have been expanded in the body of the clause for `p3/4`, corresponding to `append/3` under `solve/2`, thereby obtaining significant amounts of parallelism.

5. Relation to Other Work

It is well known that partial evaluation is particularlly effective for optimizing programs that are written in *meta-programming* style [Takeuchi 86 and Levi 86]. There are also successful works generating compilers and compiler-compiler by self-application of a partial evaluator [Fujita 87 and Fuller 87].

The motivation and background of the work implementing functional languages on parallel logic languages described here are almost on the same line as those in the work on CFL [Levy 87], where a special concurrent functional language is defined and embedded in a concurrent logic programming environment. The advantage of our method over Levy's is generality and flexibility in the compiling method. Levi's method is largely based on the hand-written translator that translates CFL programs into FCP. Although they also uses a partial evaluator, its use is limited to a help for performing local optimization on the translated program. On the other hand, our method needs an interpreter instead of a translator, and the general purpose partial evaluator performs all the tasks of compilation. Even if construction of a translater is not a very difficult task, it seems easier to construct an interpreter. Moreover, it bocomes far easier to enhance compilation (translation) with additional functionalities by using *flavor mixing* method [Gallagher 86], if the interpreter approach is chosen.

Several works have been reported on compilation of Prolog or OR-parallel programs into AND-parallel logic languages [Ueda 86, Codish 86 and Shapiro 87]. Their methods have the advantage that the resultant programs are very efficient. However, Ueda's method imposes rather severe restrictions on the source program, that is, predicates must be strongly typed (producer or consumer). Codish's and Shaipro's methods allow fairly wide class of source programs including *cut*. However, their compilation schemes and execution algorithms are very specific and strongly dependent on the base language (FCP). Compared to these, our compilation scheme is more general and flexible. Similar partial evaluators to the FGHC partial evaluator would be constructed for other parallel logic languages (FCP and PARLOG as well as KL/1) without serious difficulties.

6. Conclusion

A partial evaluator for FGHC programs has been implemented in FGHC. The partial evaluator is based on *UR-set* with constraint solving and is as powerful as Futamura's β for *generalized partial computation*. The partial evaluator succeeded to specialize programs with an order of magnitude improvement of runtime efficiency.

One of the most promising application is compiling programs with respect to its (meta-)interpreter. By eliminating an extra interpretation layer, an order of magnitude improvement of runtime efficiency can be obtained.

Backus' FP programs are easily compiled to efficient parallel programs in FGHC with reasonable cost. On the other hand, compiling Prolog programs with respect to the all-solution interpreter costs rather expensive (consuming much time and space), since the interpreter is fairly complex due to the heavy tasks concerning variable management: unification, substitution application and so on. Further investigation is required to make the partial evaluation more economical, as well as to achieve more performance improvement for subject programs.

From several experiences, the author had a feeling that partial evaluation is less effective for programs written carefully so as to exploit maximal parallelism from the beginning, especially in *Layered Stream* programming style [Okumura 87]. However, it would be promising to write an interpreter using Layered Stream and to compile source programs with respect to the interpreter using the partial evaluator, thereby obtaining target programs enjoying maximal parallelism.

Acknowledgements

The author would like to thank Dr. Koichi Furukawa, deputy director of ICOT Research Center, and Dr. Ryuzo Hasegawa, chief of First Research Lab. of ICOT, for their encouragement. Special thanks are due to Professor Giorgio Levi for valuable discussions, and Dr. Philippe Devienne for helpful comments and suggestions. Thanks also go to my colleagues in First Research Lab. for discussions and support for implementation environment.

References

[Backus 78] J.W. Backus, Can programming be liberated from Von Neumann style? A functional style and its algebra of programs, *CACM* 21(8) 1978

[Codish 86] M. Codish and E. Shapiro, Compiling Or-parallelism into And-parallelism, in *The Journal of New Generation Computing,* 5(1) pp.45-61, 1987

[Ershov 82] A.P. Ershov, Mixed Computation: Potential Applications and Problems for Study, *Theoretical Computer Science,* (18) pp.41-67, 1982

[Fujita 87] H. Fujita and K. Furukawa, A Self-applicable Partial Evaluator and Its Use in Incremental Compilation, in *Workshop on Partial Evaluation and Mixed Computation, Gl. Avernaes, Denmark 1987, The Journal of New Generation Computing,* Vol.6 (to appear)

[Fujita 88] H. Fujita, A. Okumura and K. Furukawa, Partial Evaluation of GHC Programs Based on UR-set with Constraint Solving, in *The Fifth International Conference Simposium on Logic Programming,* (to appear) 1988

[Fuller 87] D.A. Fuller and S. Abramsky, Mixed Computation of Prolog Programs, in *Workshop on Partial Evaluation and Mixed Computation, Gl. Avernaes, Denmark 1987, The Journal of New Generation Computing,* Vol.6 (to appear)

[Furukawa 87] K. Furukawa and A. Okumura, Unfolding Rules for GHC Programs, ibid.

[Futamura 71] Y. Futamura, Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler, *Systems Computers Controls,* 2(5) pp.45-50, 1971

[Futamura 87] Y. Futamura, Generalized Partial Computation, in A.P. Ershov, D. Bjørner and N.D. Jones eds., *Workshop on Partial Evaluation and Mixed Computation, Gl. Avernaes, Denmark 1987,* North-Holland, (to appear)

[Gallagher 86] J. Gallagher, Transforming Logic Programs by Specialising Interpreters, in *ECAI-86 The Seventh European Conference on Artificial Intelligence, Brighton Centre, United Kingdom,* pp.109-122, 1986

[Gallagher 87] J. Gallagher and M. Codish, Specialisation of Prolog and FCP programs, in *Workshop on Partial Evaluation and Mixed Computation, Gl. Avernaes, Denmark 1987, The Journal of New Generation Computing,* Vol.6 (to appear)

[Jones 85] N.D. Jones, P. Sestoft and H. Søndergard, An Experiment in Partial Evaluation: The Generation of a Compiler Generator. in J.-P. Jouannaud ed., *Rewriting Techniques and Applications,* Lecture Notes in Computer Science, Vol.202, Springer-Verlag, pp.124-130, 1985

[Levi 86] G. Levi, 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

[Levy 87] J. Levy and E. Shapiro, CFL — A Concurrent Functional Language Embedded in a Concurrent Logic Programming Environment, in E. Shapiro ed., Concurrent Prolog, Vol.1, pp.442-469, 1987

[Okumura 87] A. Okumura and Y. Matsumoto, Parallel Programming with Layered Stream, in *The Fourth Symposium on Logic Programming,* pp.224-232, 1987

[Safra 86] S. Safra and E. Shapiro, Meta Interpreters for Real, in *Information Processing 86, Dublin, Ireland,* North-Holland, pp.271-278, 1986

[Shapiro 87] E. Shapiro, Or-parallel Prolog in Flat Concurrent Prolog, in J.-L. Lassez ed., *The Fourth International Conference on Logic Programming,* MIT Press, pp.311-337, 1987

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

[Ueda 85] K. Ueda, Guarded Horn Clauses, in *Proc. Logic Programming '85,* E. Wada ed., Lecture Notes in Computer Science, Vol.221, Springer-Verlag, pp.168-179, 1986

[Ueda 86] K. Ueda, Making exhaustive search programs deterministic, in *The Third International Conference on Logic Programming,* Lecture Notes in Computer Science, Vol.225, Springer-Verlag, pp.270-282, 1986

[Ueda 88] K. Ueda, internal memo, 1988

Appendix

A.1 FP "factorial" Program Compiled to FGHC

$$p0(Arg, Res) \equiv apply(fact, Arg, Res)$$

```
p0(Arg,Res) :- true | m(Arg,Rs_82,R2_56), p6(R1_187,Rs_82), p5(Arg,R1_187),
                      p2(R2_56,RC_36), p1(RC_36,Arg,Res).

p6(bottom,Rs_82) :- true | Rs_82 = bottom.
p6(R1_187,Rs_82) :- R1_187 \= bottom | Rs_82 = [R1_187].

p5(bottom,R1_187) :- true | R1_187 = bottom.
p5(Arg,R1_187) :- Arg \= bottom | R1_187 = 0.

p2([V_92,V_92],RC_36) :- true | RC_36 = true.
p2([U_93,V_93],RC_36) :- U_93 \= V_93 | RC_36 = false.
p2(R2_56,RC_36) :- R2_56 \= [V2_94,V3_94] | RC_36 = bottom.

p1(bottom,Arg,Res) :- true | Res = bottom.
p1(true,Arg,Res) :- true | p3(Arg,Res).
p1(false,Arg,Res) :- true | m(Arg,Rs_208,R2_143), p6(R1_313,Rs_208),
                            p7(Arg,R2_332), p0(R2_332,R1_313), p4(R2_143,Res).

p7(Arg,R2_332) :- integer(Arg) | R2_332 := Arg - 1.
p7(Arg,R2_332) :- noninteger(Arg) | R2_332 = bottom.

p4([X_223,Y_223],Res) :- integer(X_223),integer(Y_223) | Res := X_223 * Y_223.
p4([X_224,Y_224],Res) :- noninteger(X_224) | Res = bottom.
p4([X_225,Y_225],Res) :- noninteger(Y_225) | Res = bottom.
p4(R2_143,Res) :- R2_143 \= [V2_226,V3_226] | Res = bottom.

p3(bottom,Res) :- true | Res = bottom.
p3(Arg,Res) :- Arg \= bottom | Res = 1.
```

A.2 Prolog Unifier

```
unify(T1,T2,MGU) :- true | unify1(T1,T2,[],MGU).

unify1(V!I,V!I, Si,So) :- true                | So=Si.
unify1(V!I,  T, Si,So) :- T\=(V!I)            | updateS(V!I,T,Si,So).
unify1(  T,V!I, Si,So) :- T\=(V!I)            | updateS(V!I,T,Si,So).
unify1(  A,  B, Si,So) :- A\=(_!_), B\=(_!_) |
  functor(A,FA,NA), functor(B,FB,NB), unify2(FA/NA,FB/NB,A,B,Si,So).

  unify2(FA/NA,FB,A,B,Si,So) :- FA/NA  = FB | unify3(NA,A,B,Si,So).
  unify2(FA,   FB,A,B,Si,So) :- FA    \= FB | So=fail.

  unify3(N,A,B,Si,So) :- N>0  | N1:=N-1, arg(N,A,A1), arg(N,B,B1),
                                unify1(A1,B1,Si,Sm), unify4(N1,A,B,Sm,So).
  unify3(0,_,_,Si,So) :- true | Si=So.

    unify4(_,_,_,fail,So) :- true     | So=fail.
    unify4(N,A,B,  Si,So) :- Si\=fail | unify3(N,A,B,Si,So).

  updateS(V,T,Si,So) :- true | lookupS(V,Si,R), updateS1(R,V,T,Si,So).

    lookupS(V,[V-U!I|_], R) :- true        | R=rvar(U!I).
    lookupS(V,[V-T|_],   R) :- T\=(_!_)    | R=term(T).
    lookupS(V,[U-V!_],   R) :- true        | R=lvar(U).
    lookupS(V,[U-T|Si],  R) :- V\=U, V\=T | lookupS(V,Si,R).
    lookupS(_,[],        R) :- true        | R=newvar.

    updateS1(newvar,  V,T,Si,So) :- true | So=[V-T|Si].
    updateS1(lvar(U), V,U,Si,So) :- true | So=Si.
    updateS1(lvar(U), V,T,Si,So) :- T\=U | So=[V-T|Si].
    updateS1(rvar(U), V,U,Si,So) :- true | So=Si.
    updateS1(rvar(U), V,T,Si,So) :- T\=U | So=[U-T|Si].
    updateS1(term(X), V,T,Si,So) :- true | unify1(X,T,Si,So).
```

A.3 Prolog "append" Program Compiled to FGHC

$$p3(X,Y,Z,Sol) \equiv solve(append(X,Y,Z),Sol)$$

```
p3(X,Y,Z,Sol) :- true |
  p7(Z,V3_19), p7(Y,V2_19), p7(X,V1_19),
  inc(0,Id1_119), inc(Id1_119,Id1_163), ps(Ps1_48,Id1_163),
  merge(Ps1_266,Ps2_266,Ps1_48), o(So1_266,So2_266,S1_1),
  p97(V3_19,Id1_119,Sm_449), p142(V1_19,V2_19,V3_19,Id1_119,Sm_449,U_266),
  p9(U_266,S1_266), derefSubst(S1_266,S2_266),
  p57(S2_266,Ps1_266,So1_266), p48(Ps1_272,Ps2_266), p49(So1_272,So2_266),
  p132(V3_19,Id1_163,Sm_451), p173(V1_19,V2_19,V3_19,Id1_163,Sm_451,U_272),
  p9(U_272,S1_272), derefSubst(S1_272,S2_272),
  p99(S2_272,Id1_163,Ps1_272,So1_272), p18(V3_19,Vm_81),
  varsIn(V2_19,Vm_81,Vm_166), varsIn(V1_19,Vm_166,Vm_218),
  minimizeSS(S1_1,Vm_218,S2_1), splitSS(S2_1,Vm_218,Sol,V7_1).
```

Due to the limited space, subpredicates for `p3/4` are omitted.