# A Guide to Specialising Gödel Programs with the Partial Evaluator *SAGE*

C.A.Gurr[*]
Human Communication Research Centre
University of Edinburgh
2 Buccluech Place, Edinburgh EH8 9LW Scotland

April 1994

**Abstract**

This document provides a description of the partial evaluator *SAGE*, a partial evaluator written in the logic programming language Gödel which specialises Gödel programs. Although capable of specialising any Gödel program *SAGE* was developed primarily to specialise meta-programs which use a ground representation. To assist users of *SAGE* this document gives an overview of *SAGE*, a manual for its use and a guide to writing Gödel programs, particularly meta-programs, in a style which makes them amenable to specialisation.

## 1 Introduction

Gödel is a declarative, general-purpose logic programming language which provides a number of higher-level programming features, including extensive support for meta-programming with a *ground representation*. The Gödel language is documented in [5] and we refer here to the current implementation of Gödel as "The Bristol Gödel".

This document provides a description of the partial evaluator *SAGE* (Self-Applicable Gödel partial Evaluator), a manual for its use and a guide to writing Gödel programs in a style which makes them amenable to specialisation by *SAGE*. A full description of the techniques employed by *SAGE* is given in [4]. A summary of that document is given in [3].

*SAGE* was originally implemented to test techniques developed during research into the self-application of a declarative partial evaluator written in a logic programming language. As such *SAGE* was and is designed primarily for the specialisation of Gödel meta-programs which use a ground representation.

Gödel meta-programs which use a ground representation suffer from a computational expense that this incurs which is in addition to the so-called 'interpretation overhead' that is recognised in all meta-programs. However, these extra overheads may be almost entirely removed by partial evaluation. We refer to this as "specialising the ground representation" and a summary of the techniques used to perform this is given by [2]. These techniques may be applied to supplement techniques used to remove the more familiar interpretation overheads.

The layout of this guide is as follows. In the following section we give a brief overview of partial evaluation as it is defined for *SAGE*. In Section 3 we provide a user-manual for *SAGE* and give some

[*]email: corin@cogsci.ed.ac.uk

pointers on good programming style for writing Gödel programs which are intended to be specialised by *SAGE*. Finally in Sections 4 and 5 we describe with examples *SAGE*'s strategy for unfolding general Gödel programs and Gödel meta-programs respectively.

## 2 An Overview of Partial Evaluation

*SAGE* is a partial evaluator based mainly on finite unfolding. Partial evaluation by unfolding was put on a firm theoretical footing by Lloyd and Shepherdson in [7] and we refer to that paper for a formal definition of partial evaluation as used by *SAGE*. In this section we give first a brief outline of the concept of partial evaluation by finite unfolding and then describe some issues in the application of this technique which relate to the Gödel language and to the *SAGE* partial evaluator in particular

### 2.1 Partial Evaluation by Finite Unfolding

In the context of [7] the basic technique for partially evaluating a program $P$ wrt a goal $G$ is to construct "partial" search trees for $P$ with suitably chosen atoms from $G$ as goals, and then extract the specialised program $P'$ from the definitions associated with the leaves of these trees.

For example, let $P$ be the following (partial) program:

```
P(x) <- Q(x,y) & R(y).
P(x) <- S(x).
Q(x,A) <- T(x).
Q(x,B) <- T(F(x)).
R(A).
R(B).
```

and `<- P(x)` the goal we wish to partially evaluate $P$ wrt. We would therefore construct a partial SLDNF-tree for this program wrt the singleton set {`P(x)`}. The following is one possible such tree:

$$\begin{tikzcd}
 & & \texttt{<-}\underline{\texttt{P(x)}} \arrow[dl] \arrow[dr] & \\
 & \texttt{<- Q(x,y) \& }\underline{\texttt{R(y)}} \arrow[dl] \arrow[dr] & & \texttt{<- S(x)} \\
\texttt{<- }\underline{\texttt{Q(x,A)}} \arrow[d] & & \texttt{<- }\underline{\texttt{Q(x,B)}} \arrow[d] & \\
\texttt{<- T(x)} & & \texttt{<- T(F(x))} &
\end{tikzcd}$$

We next extract new statements for the predicate `P` from the leaf nodes of this tree, giving us the new definition:

```
    P(x) <- T(x).
    P(x) <- T(F(x)).
    P(x) <- S(x).
```

To construct the partial evaluation of $P$ wrt `<- P(x)` we lastly replace the definition of the predicate `P` in $P$ with this new definition.

## 2.2 Partial Evaluation of Gödel Programs

Gödel provides a module system for programming and has a large library of system modules (20 in the Bristol Gödel). In this section we describe two aspects of this module system which affect and are affected by partial evaluation

### 2.2.1 Open and Closed Code in Gödel Programs

Modules in Gödel programs may be of one of two kinds. The first kind of modules are those which have been written by some Gödel user and are referred to as *user-defined modules*. The second kind of modules are those which are provided by the Gödel system and these are referred to as *system modules*. The majority of system modules are what we refer to as *closed* modules. A module is closed if its export part uses the module keyword `Closed` instead of the keyword `Export`. Any module which is not closed is an *open* module. In the Bristol Gödel only system modules may be closed modules and all system modules are closed other than the `Syntax` module.

All code in a Gödel program falls into one of two categories, which we refer to as *open* and *closed*. We define the open code of a Gödel program to be that code which is either defined in a user-defined module of the program or which is defined in an open system module. We define all code that is defined in a closed Gödel system module as being closed code.

When *SAGE* encounters an atom whose predicate or proposition symbol is declared in a closed module it interprets the execution of a call to this atom wrt the program being partially evaluated as long as this atom matches any delay declaration for the relevant predicate or proposition. Thus an atom such as `Member(x,[1,2,3])`, whose predicate is defined in the system module `Lists` will be executed by *SAGE* during a partial evaluation. An atom such as `Member(x,y)` however will not be executed as it does not match the delay declaration (in the system module `Lists`) for this predicate:

```
  DELAY  Member(_,y)  UNTIL NONVAR(x).
```

Any such atom which is not executed by *SAGE* will be left as a residual atom in the partial evaluation.

### 2.2.2 Gödel Scripts

When partially evaluating a Gödel program a process we refer to as *flattening* occurs. The details of this flattening process are not relevant here but we remark that because of this process any partial evaluator must construct the results of a partial evaluation as a Gödel *script*. A script is essentially a Gödel program from which all module structure has been removed.

Gödel supports the representation of partially evaluated programs as scripts and a file, `Program.scr`, containing the representation of a script may be compiled into executable code by calling the script-compile command `;sc Program` from the Gödel environment.

Removing the module structure of a program by constructing its partial evaluation as a script is not as drastic a measure as it may seem, if we assume that the module structure is provided primarily for software engineering purposes. Here the module structure is a useful aid to the programmer when writing and debugging the original program. It seems safe to assume that a program will only be partially evaluated once it is complete and (hopefully) bug-free. In this case the user needs only to be certain that the answers computed by the partially evaluated program are correct with respect to the original program and he/she is unlikely to be concerned with the module structure of the specialised program. In fact, taking the widely accepted view that partial evaluation may be considered as a part of the compilation process for a program, the above argument is perfectly acceptable. All that programmers will generally require from the compilation of their programs is that the compiled version of a program should be correct with respect to the original program.

## 2.3 *SAGE*'s Implementation of Partial Evaluation

In this section we give an overview of *SAGE*'s strategy for partially evaluating Gödel programs.

### 2.3.1 Non-Selectable Predicates

When requesting *SAGE* to perform a partial evaluation of some program the user may denote certain of the predicates in the program to be *non-selectable.* A predicate is *selectable* if it is not non-selectable.

Non-selectable predicates are never selected for unfolding by *SAGE*, thus they can be used to mask off parts of the object program which a user would prefer not to be specialised.

When specifying the set of non-selectable predicates it is necessary to ensure that any predicate which a non-selectable predicate depends upon is also non-selectable. A set of non-selectable predicates for which this property holds is said to be *well-structured*.

In general it is not necessary to mark all predicates which non-selectable predicates depend upon as also being non-selectable. If a particular predicate is depended upon by some non-selectable predicate in the program to be partially evaluated we need only mark this particular predicate as non-selectable if it is also depended upon by some selectable predicate.

**Note: The current implementation of *SAGE* does not check for well-structuredness of the non-selectable predicates. This property is currently the responsibility of the user.**

### 2.3.2 Unsafe Predicates

Prior to computing the partial evaluation of an object program and goal *SAGE* will construct a predicate dependency graph for that goal to identify the recursive selectable predicates which the goal depends upon and then perform a static analysis of the goal wrt the object program.

The static analysis detects those atoms with recursive predicates which will be encountered in the subsequent partial evaluation. The purpose of this analysis is to determine which of these atoms are sufficiently instantiated for them to be unfolded during partial evaluation without the risk of an infinite unfolding.

The static analysis is based upon an analysis of ground arguments in atoms. If it may be determined that each occurrence of an atom with some recursive selectable predicate has a particular argument which will always be ground and that this can guarantee the finite unfolding of this atom then this predicate is marked *safe*. Any recursive selectable predicate which is not marked safe will be marked *unsafe* by the static analysis.

For example, consider the following definition of the standard `Member` predicate:

```
  Member(x, [x|_]).
  Member(x, [_|list]) <- Member(x, list).
```

This predicate is recursive. If we may determine that every atom with this predicate which is encountered in some partial evaluation will have a ground term in the second argument, then it is obvious that such an atom may be unfolded in a finite number of unfolding steps and thus this predicate would be marked safe by the static analysis. If we were unable to determine that every atom with this predicate in some partial evaluation had a ground term in its second argument then the static analysis would mark this predicate as being unsafe.

#### 2.3.3 Partially Evaluated Atoms

Having performed the static analysis *SAGE* will identify all safe selectable atoms in the goal to be partially evaluated and compute the partial evaluation of these atoms. In addition *SAGE* will identify all instances of atoms with unsafe predicates. For each unsafe predicate *SAGE* will compute a generalisation of every atom in the partial evaluation which has this predicate. This generalisation is referred to as a *covering atom* for this predicate. A partial evaluation of each of the covering atoms is then computed.

#### 2.3.4 Selection Strategy

The current implementation of *SAGE* relies upon a depth-first leftmost safe selectable literal selection strategy. That is to say, when selecting a literal for unfolding from a formula, *SAGE* will always select the leftmost literal whose predicate symbol is both selectable and safe.

Note that by not selecting literals with unsafe predicates and computing partial evaluations of the unsafe atoms in which unsafe literals are not selected, *SAGE* produces specialised *recursive* definitions for unsafe predicates.

#### 2.3.5 Post-Partial Evaluation Optimisations

Having performed the partial evaluation *SAGE* performs a post-processing optimisation which involves the deletion of redundant terms from the partially evaluated atoms.

For example, if `P(A,x,y,B)` is an atom which is partially evaluated by *SAGE* then a new predicate `P_1` (binary, in this case) will be defined and every instance of this partially evaluated atom will be replaced by an equivalent instance of the atom `P_1(x,y)`. The definition of `P` will be replaced with the definition of this new predicate `P_1` in the specialised program.

When constructing the specialised program *SAGE* will delete all redundant selectable predicates. A predicate is redundant if it is not relied upon by any predicate appearing in some literal in the partial evaluations computed by *SAGE*.

## 3 A Guide to Using *SAGE*

In this section we give an outline of *SAGE*'s main algorithm and describe how it is used. We note that *SAGE*'s strategy for computing partial evaluations is a relatively simple one and therefore we conclude this section with some pointers on how to write Gödel programs which may be optimally specialised by *SAGE*.

### 3.1 Algorithm for *SAGE*

When invoked *SAGE* will prompt the user to input the name of the object program, the object goal this program is to be specialised wrt and the list of non-selectable predicates. Once *SAGE* has constructed the partial evaluation of this program and written the resulting script into a file it will output a message describing the partial evaluation which has been computed. We refer to this message as the *log* of the partial evaluation and a copy of this log is also written to a file.

The following list describes the main points of *SAGE*'s operation:

1. User inputs object program, object goal and list of non-selectable predicates
2. Extract selectable atoms from goal
3. Compute predicate dependency graph to determine recursive selectable predicates which atoms to be partially evaluated depend upon
4. Compute static analysis to determine which recursive predicates are unsafe and construct covering atoms for them
5. Compute partial evaluations of safe selectable atoms in goal and covering atoms
6. Construct residual script:
   - Optimise partially evaluated atoms
   - Remove superfluous code
   - Construct new predicate declarations
   - Insert specialised declarations and code
7. Write residual script to file. Write log to screen and file.

## 3.2 Running *SAGE*

In this section we provide a manual for using *SAGE* and an example of its execution.

### 3.2.1 Loading and Invoking *SAGE*

*SAGE* is comprised of four modules, `SAGE`, `PE`, `Assemble` and `Analyse`. It is loaded by loading the top-level module `SAGE`. Once loaded *SAGE* is invoked by the proposition `RunPE`.

### 3.2.2 User Inputs

Strings are input to *SAGE* without quotes and are terminated by a carriage-return. Full stops are not required to terminate a string.

Non-selectable predicates are input individually as atoms. The end of a list of them is denoted by inputting a carriage return to the prompt. When specialising a meta-program (that is, any program which imports the system module `Syntax`) the user is prompted whether they want to mark the "WAM-like" predicates as non-selectable. The relevance of these predicates is discussed in Section 5.1.

The file extensions for the object program and specialised script (that is, `.prm` and `.scr` respectively) are appended automatically by *SAGE*. So if your program's main module is `Main` then *SAGE* will write the residual script to the file `Main.scr`. This process will overwrite existing files of that name, so if you have such a file we suggest that a copy with a different name is made before the partial evaluation is initiated.

### 3.2.3 *SAGE* Specialisation Logs

Upon completion *SAGE* prints out the specialisation log describing the specialisation it has just performed. A copy of this log is also written to the file `Main.new`, where `Main` is the name of the main module of the specialised program.

### 3.2.4 An Example Run: Specialising the Program `Append`

The module `Append.loc` is a Gödel program which gives a user-defined version of the standard `Append` predicate. This program defines `MyAppend`, an open-code version of the closed predicate `Append` which is defined in the system module `Lists`.

```
MODULE  Append.

IMPORT Lists.

PREDICATE  MyAppend : List(a) * List(a) * List(a).

MyAppend([],x,x).
MyAppend([a|x],y,[a|z]) <- MyAppend(x,y,z).
```

We show below the listing of a sample run of *SAGE* in which the `Append` program is specialised wrt the goal `MyAppend(x,y,[1,2,3])`.

```
unix% goedel
Goedel 1.3
Type ;h. for help.
[] <- ;load SAGE.
Loading module "SAGE" ...
Loading module "PE" ...
Loading module "Assemble" ...
Loading module "Analyse" ...
[SAGE] <- RunPE.

         SAGE  Partial Evaluator.
        --------------------------

  Welcome :
  ---------

Object Program Name ? Append
Loading program...done
Object goal ? MyAppend(x,y,[1,2,3])

Non-Selectable Atoms :
----------------------

Input non-selectable predicates as atoms, terminating with empty string.
Next atom ?
```

```
Constructing predicate dependency graph...done
Performing static analysis...done
Performing partial evaluation...done
Optimising residual code...done
Writing script in file Append.scr...done

% --------------------------------------
  Script: Append.scr
  Partial evaluation of program:  Append
  wrt goal:  MyAppend(x,y,[1,2,3])
% --------------------------------------

The following atoms were partially evaluated:
MyAppend(x,y,[1,2,3])

No predicates were marked non-selectable.
No predicates were marked unsafe.
The following new predicates have been defined:
Predicate MyAppend/3, defined in module Append,
has been replaced by:

PREDICATE  MyAppend_1 : List(Integer) * List(Integer).

as if re-defined by the statement:
  MyAppend(v,v_1,[1,2,3]) <- MyAppend_1(v,v_1).

These changes have been recorded in the file Append.new
Do you wish SAGE to perform another partial evaluation (Y/N)? n

  Goodbye.
  --------
Yes
[SAGE] <-;quit.
unix%
```

Figure 1 gives the specialised code produced by *SAGE* following this run.

```
    MyAppend_1([],[1,2,3]).
    MyAppend_1([1],[2,3]).
    MyAppend_1([1,2],[3]).
    MyAppend_1([1,2,3],[]).
```

Figure 1: Specialised Code for Predicate `MyAppend`

The specialised version of the `Append` program has been written to the file `Append.scr`. This file may be compiled into executable code using the `;sc` command (script-compile) in the Gödel environment, as the following listing demonstrates:

```
unix% goedel
Goedel 1.3
Type ;h. for help.
```

```
[] <- ;sc Append.
Compiling script "Append" ...
Module "Append" compiled.
[] <- ;load Append.
Loading module "Append" ...
[Append] <- ;type Append_1.
PREDICATE
   Append_1 : List(Integer) * List(Integer).

[Append] <- Append_1(x,y).

x = [],
y = [1,2,3] ? ;

x = [1],
y = [2,3] ?
Yes
[Append] <- ;quit.
unix%
```

## 3.3 Viewing Residual Scripts

The Bristol Gödel does not currently provide a script-viewer, so the means of viewing the residual code produced by *SAGE* are somewhat limited at the moment.

Support for viewing scripts is currently provided by the program `View`. This program consists of the single module `View` and once loaded may be invoked with the atom `ViewScript(<script_name>)`, where `<script_name>` is the name (a string) of the top level module of the program that has been specialised. For example, the specialised code for the `Append` program given above, which is stored in the file `Append.scr`, may be viewed via the call `ViewScript("Append")`.

## 3.4 Some Tips on Writing Programs to be Specialised

Recall that atoms which are selectable and not unsafe are always unfolded by *SAGE*. It must be stressed that this, unfortunately, is not always a good thing as the following example illustrates.

Let $P$ be the following program:

```
FilterList([],[]).
FilterList([x|rest], list) <-
  Filter(x, list, list1) &
  FilterList(rest, list1).

Filter(A, list, list).
Filter(x, [x|list], list) <-
  x ~= A.
```

which we are specialising wrt the goal `<- FilterList([x,y,z],list)`. If the predicates `FilterList` and `Filter` were both marked selectable and safe then *SAGE* would produce the following specialised definition for `FilterList`:

```
FilterList([A,A,A],[]).
FilterList([A,A,x],[x]) <- x ~= A.
FilterList([A,x,A],[x]) <- x ~= A.
FilterList([A,x,y],[x,y]) <- x ~= A & y ~= A.
FilterList([x,A,A],[x]) <- x ~= A.
FilterList([x,A,y],[x,y]) <- x ~= A & y ~= A.
FilterList([x,y,A],[x,y]) <- x ~= A & y ~= A.
FilterList([x,y,z],[x,y,z]) <- x ~= A & y ~= A & z ~= A.
```

The above new definition, while highly specialised, is probably not ideal as we have introduced a large element of non-determinism which makes this residual code less efficient than it might be. The problem is caused by the fact that we have unfolded the call to `Filter` when we probably would have preferred not to.

To solve this problem we must find a means of preventing such unfoldings from being made. There are three ways in which this may be achieved, the first of which is that we mark a predicate such as `Filter` as being non-selectable. For the program and goal above such a partial evaluation would lead to the following specialised code:

```
FilterList([x,y,z], list) <-
  Filter(x, list, list1) &
  Filter(y, list1, list2) &
  Filter(z, list2, []).

Filter(A, list, list).
Filter(x, [x|list], list) <-
  x ~= A.
```

However, marking a predicate non-selectable means that *SAGE* will not be able to perform any specialisation of this predicate. If we wish to both specialise a predicate and prevent unwanted unfoldings of calls with this predicate we have two further options.

The first option is to ensure that the relevant predicate will be marked unsafe by *SAGE*. This will happen whenever this predicate is recursive and insufficiently instantiated at the time of partial evaluation. This strategy of *SAGE*'s will often work successfully for recursive predicates in meta-programs, as examples in Section 5 demonstrate. However, for some predicates, particularly non-recursive predicates such as `Filter` above, this is not sufficient.

A second option for preventing unwanted unfoldings makes use of *SAGE*'s technique for specialising conditional formulas in Gödel programs. This technique is described in Section 4.2 and is guided primarily by a test of whether the condition of a conditional is sufficiently instantiated at the time of partial evaluation for *SAGE* to be able to determine whether or not it will succeed when the specialised program is executed.

If the condition of a conditional is sufficiently instantiated at partial evaluation time then it will be unfolded together with the then_part or else_part, as appropriate. If the condition is not sufficiently instantiated at partial evaluation time then the conditional will still be specialised, but the residual code will remain as a single conditional formula.

For example, we might modify the above definition of the predicate `FilterList` to replace the call to `Filter` with a conditional formula as follows:

```
FilterList([],[]).
FilterList([x|rest], list) <-
  IF  x = A
    THEN  FilterList(rest, list)
    ELSE  list = [x|list1] &
          FilterList(rest, list1).
```

or, alternatively, as follows:

```
FilterList([],[]).
FilterList([x|rest], list) <-
  (
  IF  x = A
    THEN  list = list1
    ELSE  list = [x|list1]
  ) &
  FilterList(rest, list1).
```

Of these two alternatives we comment that the second is preferable as the recursive call to `FilterList` only appears once in the second statement. A general rule of thumb is that the then_part and else_part of conditionals used in this way should be kept simple wherever possible.

When this second alternative definition is specialised wrt the goal `<- FilterList([x,y,z],list)`, *SAGE* will produce the following new definition:

```
FilterList([x,y,z], list) <-
  (
  IF  x = A
    THEN  list = list1
    ELSE  list = [x|list1]
  ) &
  (
  IF  y = A
    THEN  list1 = list2
    ELSE  list1 = [y|list2]
  ) &
  (
  IF  z = A
    THEN  list2 = []
    ELSE  list2 = [z]
  ).
```

## 4 Specialising Gödel Programs

In this section we give an overview of the extensions to the definition of partial evaluation in [7] which permit the specialisation of Gödel programs containing logical connectives in addition to `&`, conditional formulas, commit operators and set terms.

### 4.1 Quantifiers and Connectives

Unfolding formulas of the form `A <- B`, `A -> B`, and `A <-> B` is handled simply by transforming them to the forms `A \/ ~B`, `~A \/ B` and `(A & B) \/ (~A & ~B)`, respectively. Disjunctions may be unfolded to produce two resultants, one for each of the disjuncts. For example, the disjunction in the resultant `H <- A & (B \/ C) & D` can be unfolded to produce the two new resultants `H <- A & B & D` and `H <- A & C & D`.

During *SAGE*'s partial evaluations existentially quantified variables are renamed so as to have names different from all other variables in the formula. Consequently *SAGE* may generally ignore existential quantifiers, as the names of the quantified variables are guaranteed not to occur outside the scope of the quantifier. The cases where we are interested in knowing whether variables are free or quantified, in negated formulas for example, are dealt with by the unfolding strategy for those particular cases.

Universally quantified formulas such as `ALL [x] P(x,y)`, are transformed to the form `~(SOME [x] ~P(x,y))`.

## 4.2 Constructive Negation and Conditional Formulas

For negated formulas in Gödel programs we have extended the definition of partial evaluation in [7] to include the concept of constructive negation [1]. Constructive negation may be described as follows. First a partial evaluation of the formula that has been negated is computed. If there are no residual resultants for this partial evaluation then the formula has failed finitely and the negation has therefore succeeded. If at least one of residual resultants has an empty body and has not bound any variables in the original formula then the negation fails safely, otherwise the negations of these resultant bodies and the bindings they compute are conjoined to produce a specialised version of the negation. This specialised version of the negation is recorded in the residual script as the definition of a new predicate.

For example, let $P$ be the program:

```
P(x) <- Q(x).
Q(A).
Q(B).
```

where `A`, `B` and `C` are constants. The partial evaluations of the formulas `P(C)`, `P(A)` and `P(x)` would be the sets {}, {`True`} and {`x=A,x=B`} respectively, where `True` is the truth-proposition. The specialisations of the formulas `~P(C)`, `~P(A)` and `~P(x)` would therefore be `True`, `False` and `~Not_0(x)` respectively, where `False = ~True` and the definition of the new predicate `Not_0` is:

```
Not_0(A).
Not_0(B).
```

Gödel also provides a conditional operator of the form

`IF` *Condition* `THEN` *Formula1* `ELSE` *Formula2*

which is defined to mean

(*Condition* `&` *Formula1*) `\/` (`~`*Condition* `&` *Formula2*)

and is used to avoid the need to compute the formula *Condition* twice. We use constructive negation in the specialisation of this operator in the following manner. First a partial evaluation of *Condition* is computed. If there are no residual resultants for this partial evaluation then *Condition* has failed finitely and the conditional is replaced by *Formula2*, which may subsequently be specialised further. Alternatively the specialised version of *Condition* will be the disjunction of all the residual resultant bodies of this partial evaluation and the bindings they compute. If any of these disjuncts is an empty formula then, in at least one case, *Condition* has terminated successfully and bound no free variables. In this case we may replace the conditional with the conjunction of the specialised *Condition* and *Formula1*, which may subsequently be specialised further. Otherwise the specialisation of *Condition* has not indicated whether *Condition* will succeed or fail and so *Formula1* and *Formula2* are both partially evaluated and a new conditional is constructed in which *Condition*, *Formula1* and *Formula2* are replaced by their specialised versions. As for negated formulas, these specialised versions of *Condition*, *Formula1* and *Formula2* are represented in the specialised conditional as calls to new predicates.

For example, suppose that the predicate `P` was defined as in the above example, {`R(x)`} was (in all cases) the partial evaluation of the atom `Q(x,y)` and the definition of the predicate `S` was:

```
S(x,y,z) <- IF P(x) THEN Q(y,z) ELSE Q(z,y).
```

then the specialisations of the formulas `S(A,A,B)`, `S(C,A,B)` and `S(x,A,B)` would be `R(A)`, `R(B)` and `IF Test_0(x) THEN Then_1 ELSE Else_2` respectively, where the new predicate `Test_0` and new propositions `Then_1` and `Else_2` are defined as:

```
    Test_0(A).
    Test_0(B).
    Then_1 <- R(A).
    Else_2 <- R(B).
```

## 4.3 Pruning

While there is a strong argument in favour of allowing pruning within logic programs, it is well established that the 'cut' operator of Prolog is unsound. Hill *et al* [6] propose an alternative pruning operator for logic programming, the *commit*, which, while naturally affecting completeness, is proved to be sound. The commit operator is supplied by Gödel to allow pruning and has three forms. The most general form of the Gödel pruning operator has the form `{...}_`$n$, of which two special cases have the form `|` and `{...}`. We refer to the most general form as the *full commit.*

It is also shown in [6] that, provided two conditions are met, the computational equivalence of a program and its partial evaluation wrt a given goal (that is, [7, theorem 4.3]) can be extended to encompass programs containing commits. These conditions are restrictions on the structure of the SLDNF-trees used to obtain the partial evaluation and are referred to as the *freeness* and *regularity* conditions.

A detailed description of the commit operator and proof of its soundness would unfortunately be far too long for a paper of this form. Suffice it to say that it is shown in [4] that, while the freeness condition is acceptable, the regularity condition imposes restrictions on the computation of partial evaluations which are both expensive to enforce and lead to a significant reduction in the amount of specialisation that may be performed. In [4] a technique for partially evaluating programs containing commit operators is presented which allows us to strengthen the partial evaluation theorem for such programs ([6, theorem 3.2]) by entirely removing the regularity condition.

While techniques have been developed that enable a partial evaluator to perform unfolding without respecting the regularity condition, the full commit is necessary to implement these techniques. Unfortunately the Bristol Gödel does not currently support the full commit and therefore the version of *SAGE* documented here cannot implement these techniques. In view of the restrictive nature of unfolding while respecting the regularity condition we have decided that the best approach under the circumstances is that the current implementation of *SAGE* should remove all commits in any program statements which it specialises. This approach will not affect the soundness of the partial evaluation although it does imply that a partially evaluated program may return more answers than the unspecialised program for certain queries, as pruning is effectively disabled for all selectable predicates.

## 4.4 Set Terms and the `Sets` Module

**Note: The current implementation of *SAGE* does not support programs which import the Sets module - therefore the correctness of partially evaluated programs which import `Sets` cannot be guaranteed.**

## 5 Specialising Gödel Meta-Programs

*SAGE* is a partial evaluator which was developed primarily to specialise Gödel meta-programs which use the ground representation. Therefore it is fitting that we devote a section of this guide to illustrating, with examples, the features of Gödel's ground representation which allow suitable meta-programs to be effectively specialised by *SAGE*.

### 5.1 The Interpreter `SLD`

We first present the meta-interpreter `SLD` which is analogous to the well-known vanilla or *solve* meta-interpreter of languages such as Prolog. The difference between `SLD` and a vanilla-interpreter is that `SLD` uses a ground representation and a vanilla interpreter uses a non-ground representation.

The meta-interpreter `SLD` is the most simple ground meta-interpreter which could be written and we use it here to illustrate the basis issues involved in specialising Gödel meta-programs with *SAGE*.

#### 5.1.1 Using *SAGE* on the Module `SLD`

The module `SLD.loc` is the main module for this interpreter and illustrates two things:

1. A potential approach to I/O for for meta-programs.
2. The pre- and post-interpretation instructions.

```
MODULE      SLD.

IMPORT      Solve, ProgramsIO.

PREDICATE   Demo : String * String * String.

Demo(program_string, goal_string, answer_string) <-
  FindInput(program_string++".prm", In(is)) &
  GetProgram(is, program) &
  MainModuleInProgram(program, module) &
  StringToProgramFormula(program, module, goal_string, [goal]) &
  StandardiseFormula(goal, 0, var, goal1) &
  EmptyTermSubst(empty_substitution) &
  Solve(program, goal1, var, _, empty_substitution, answer) &
  ApplySubstToFormula(goal1, answer, answer_goal) &
  ProgramFormulaToString(program, module, answer_goal, answer_string).
```

This module declares the top-level predicate `Demo` for this interpreter. Demo has three arguments, each of type list, which give the name of the object program, the body of the object goal and the interpreted answer respectively.

#### Using I/O predicates in Meta-Programs

`Demo` calls the system predicates `FindInput`, `GetProgram` and `MainModuleInProgram` to deduce, from the string giving the name of an object program, the representation of the object program and the name of its main module. `Demo` then uses this information to deduce the representation of the object goal by using the system predicate `StringToProgramFormula`. Once this goal has been interpreted wrt the object program to return a formula containing the answer-substitution, `Demo` converts this formula into a string which may be displayed to the user by calling the system predicate `ProgramFormulaToString` (we shall demonstrate a somewhat more sophisticated way to display an answer-substitution with the program `Coroutine` in Section 5.2). Together these calls perform the I/O operations necessary to translate the users choice of object program and goal into the ground representation and to translate the interpreters computed answer back into a string which the user can comprehend.

**Performing Pre- and Post-Processing of Interpretations**

The module `SLD` imports the module `Solve` which defines the predicate `Solve`. This predicate performs the actual interpretation of the object goal wrt the object program and is called by the `Demo` predicate. Before it is called `Demo` will call the predicates `StandardiseFormula` and `EmptyTermSubstitution` to initialise the variables in the object goal and to get the representation of an empty term-substitution. `Demo` then calls `Solve` which performs the interpretation, returning the representation of the answer-substitution. To enable this answer-substitution to be converted into a string which may be returned to the user, `Demo` will apply the substitution to the interpreted goal by using the predicate `ApplySubstToFormula`. This formula is then converted to a string as described above.

**Specialising the Module `SLD`**

Examining the specialisation of the module `SLD` allows us to illustrate the difference in *SAGE*'s handling of open and closed code.

Let us assume that we wish to specialise the program `SLD` wrt a known object program, `Append` for example, and an unknown goal. We can see that we would wish to partially evaluate `SLD` wrt the goal `<- Demo("Append",g,a)`. We shall consider first what will happen when *SAGE* unfolds the I/O predicates called by `Demo`.

In the Bristol Gödel these I/O predicates are all defined in closed modules and have the delay declarations given in Figure 2. Therefore *SAGE* will execute calls which match these declarations and

```
DELAY      FindInput(x,_) UNTIL GROUND(x).
DELAY      GetProgram(x,_) UNTIL GROUND(x).
DELAY      MainModuleInProgram(x,_) UNTIL GROUND(x).
DELAY      StringToProgramFormula(x,y,z,_) UNTIL GROUND(x) & GROUND(y) &
                                                       GROUND(z).
DELAY      ProgramFormulaToString(x,y,z,_) UNTIL GROUND(x) & GROUND(y) &
                                                       GROUND(z).
```

Figure 2: Delay Declarations for I/O Predicates

leave as residual those calls which fail to match these declarations.

As the string giving the name of the object program is known, we see that the call `FindInput` will be executed and that therefore the calls `GetProgram` and `MainModuleInProgram` will also be executed. In this way *SAGE* is able to deduce the representation of the object program that `SLD` is to be specialised wrt. However, the string giving the representation of the object goal is unknown and therefore the call `StringToProgramFormula` will not be executed as the third argument is not ground.

Similarly, as after the partial evaluation of the interpretation the object goal and answer substitution will still be unknown, the call `ProgramFormulaToString` will also be left as residual.

We now consider the partial evaluation of the pre- and post-processing instructions. It is essential to note that in the Bristol Gödel the system module `Syntax` is an open module and that therefore the definitions of the predicates called during pre- and post-processing consist of open code.

The system module `Syntax`, being the only open system module in the Bristol Gödel, deserves particular care when specialised. Much of the code in `Syntax` is already as efficient as we would wish and should not be specialised. Those predicates we would wish to specialised are `Resolve`, `ResolveAll` and the 'data-type' predicates `EmptyTermSubst`, `And`, `Or`, `Not`,... and so on. The approach taken in the current implementation of *SAGE* is to treat `Syntax` as a closed module for all predicates other than `Resolve` and `ResolveAll`. This approach means that the predicates `EmptyTermSubst`, `And`, `Or`, `Not`,... will always be unfolded as they have no delay declarations.

When a user wishes to specialise a meta-program, *SAGE* will ask if the "WAM-like predicates" are to be marked non-selectable. In general the user should answer "yes" to this question, as this assists *SAGE* in specialising the ground representation by specialising calls to `Resolve`. This specialisation is described in more detail below for the module `Solve`.

Of the two pre-processing calls and the single post-processing call we see that `SAGE` will unfold the call `EmptyTermSubst` but not the calls `StandardiseFormula` and `ApplySubstToFormula`, as these latter two calls do not match their delay declarations.

To end our illustration of the partial evaluation of the module `SLD` we give the specialised definition of the predicate `Demo`, specialised wrt the object program `Append`. Figure 3 gives this specialised

```
PREDICATE   Demo_1 : String * String.

Demo_1(goal_string, answer_string) <-
  StringToProgramFormula(<Append>, "Append", goal_string, [goal]) &
  StandardiseFormula(goal, 0, var, goal1) &
  Solve_1(goal, var, _, EmptyTermSubst, answer) &
  ApplySubstToFormula(goal1, answer, answer_goal) &
  ProgramFormulaToString(<Append>, "Append", answer_goal, answer_string).
```

Figure 3: Specialised Version of Module `SLD`

definition for `Demo`, in which we have denoted the representation of the object program `Append` by `<Append>`. In addition we have replaced calls to the predicates `Demo` and `Solve` by calls to the new predicates `Demo_1` and `Solve_1`. This is a post-processing optimisation performed by *SAGE* to remove redundant terms. We shall explain it in more detail when we consider the partial evaluation of the module `Solve`.

### 5.1.2 Using *SAGE* on the Module `Solve`

In this section we illustrate the specialisation of the ground representation by an examination of the partial evaluation of the module `Solve`. We first comment that while the definition for the predicate `Solve` only covers empty formulas, atoms and conjunctive formulas, extending the definition of this predicate to cover other kinds of formulas would be very straightforward. As an example, Figure 4 gives one possible extension covering disjunctive formulas and negated formulas.

```
LOCAL      Solve.

Solve(program, goal, v, v, subst, subst) <-
  EmptyFormula(goal).
Solve(program, goal, v_in, v_out, subst_in, subst_out) <-
  And(left, right, goal) &
  Solve(program, left, v_in, new_v, subst_in, new_subst) &
  Solve(program, right, new_v, v_out, new_subst, subst_out).
Solve(program, goal, v_in, v_out, subst_in, subst_out) <-
  Atom(goal) &
  MyStatementMatchAtom(program, goal, statement) &
  Resolve(goal, statement, v_in, new_v, subst_in, new_subst, new_goal) &
  Solve(program, new_goal, new_v, v_out, new_subst, subst_out).

PREDICATE  MyStatementMatchAtom : Program * Formula * Formula.

MyStatementMatchAtom(program, atom, statement) <-
  { PredicateAtom(atom, pred, _) \/
    PropositionAtom(atom, pred) } &
  DefinitionInProgram(program, _, pred, defn) &
  Member(statement, defn).
```

```
Solve(program, goal, v_in, v_out, subst_in, subst_out) <-
  Or(left, right, goal) &
  ( Solve(program, left, v_in, v_out, subst_in, subst_out) \/
    Solve(program, right, v_in, v_out, subst_in, subst_out) ).
Solve(program, goal, v, v, subst, subst) <-
  Not(goal1, goal) &
  ~Solve(program, goal1, v, _, subst, _).
```

Figure 4: Extension to the Definition of `Solve`

### `DefinitionInProgram` **versus** `StatementMatchAtom`

In the third statement in the definition of the predicate `Solve` we have used the user-defined predicate `MyStatementMatchAtom` where one might have expected to use the system predicate `StatementMatchAtom`. The reasoning behind this choice concerns the delay declarations for the two system predicates `StatementMatchAtom` and `DefinitionInProgram` and the fact that, during partial evaluation of the program `SLD`, the object program will be known but the object goal will be unknown.

`StatementMatchAtom` and `DefinitionInProgram` are both defined in the closed system module `Programs` and have the following declarations:

```
PREDICATE  StatementMatchAtom :

  Program         % Representation of a program.
* String          % Name of an open module in this program.
* Formula         % Representation of an atom in the flat language of
                  % this program.
* Formula.        % Representation of a statement in this module whose
                  % proposition or predicate in the head is the same
                  % as the proposition or predicate in this atom.

DELAY      StatementMatchAtom(x,_,z,_) UNTIL GROUND(x) & GROUND(z).

PREDICATE  DefinitionInProgram :

  Program         % Representation of a program.
* String          % Name of an open module in this program.
* Name            % Representation of the flat name of a proposition
                  % or predicate declared in this module.
* List(Formula).  % List (in a definite order) of representations of
                  % statements in the definition of this proposition
                  % or predicate.

DELAY      DefinitionInProgram(x,_,_,_) UNTIL GROUND(x).
```

As the object program is known during partial evaluation we would expect that *SAGE* would encounter calls to these predicates for which the first argument, and only the first argument, were instantiated. Examining the above delay declarations we see that this would mean that calls to `DefinitionInProgram` would be executed by *SAGE* and that calls to `StatementMatchAtom` would be left as residual. For this reason we provide the user-defined predicate `MyStatementMatchAtom`. The declarative meaning of this predicate is equivalent to that of `StatementMatchAtom`, the difference being that when specialising the program `SLD` wrt a known object program *SAGE* may unfold the call to `MyStatementMatchAtom` and thereby deduce the representations of all open statements in the object program.

### The Partial Evaluation of `Resolve`

It is in the third statement in the definition the predicate `Solve` that we see a call to the predicate which is possibly the most important meta-programming predicate of all of those provided by Gödel: the predicate `Resolve`.

The atom `Resolve(atom,st,v,v1,s,s1,body)` is called to perform the resolution of the atom `atom` with the statement `st`. The integers `v` and `v1` are used to rename the statement with `v` being the integer value used in renaming before the resolution step is performed and `v1` being the corresponding value after the resolution step has been performed. The representations of term substitutions `s` and `s1` represent respectively the answer substitution before and after the resolution step. The last argument, `body`, is the representation of the body of the renamed statement.

The predicate `Resolve` has been designed to meet the needs of practically any Gödel meta-program which performs logical inference steps. To achieve this the implementation of this predicate handles the following operations:

- Renaming the statement to ensure that the variables in the renamed statement are different from all other variables in the current goal.
- Applying the current answer substitution to the atom to ensure that any variables bound in the current answer substitution are correctly instantiated.
- Unifying the atom with the head of the renamed statement.
- Composing the mgu of the atom and the head of the statement with the current answer substitution to return the new answer substitution.

Each of these four operations is potentially very expensive when we are dealing with the explicit representation of substitutions, therefore it is vital that they are handled in as efficient manner as possible. The implementation of `Resolve` is designed to meet this need by being both as efficient as possible *and* capable of being highly specialised by *SAGE*.

In the previous section we demonstrated how the module `SLD` would be specialised wrt an object program such as `Append`. In the specialisation of this module, assuming that the predicate `Solve` were marked as unsafe, we see that *SAGE* would determine that a specialisation of the atom `Solve(<Append>,g,v,v1,s,s1)` should be performed, where `<Append>` is the representation of the object program `Append`. As the object program is known in the call to `Solve` *SAGE* will unfold the call to `MyStatementMatchAtom` in the third statement for `Solve`. In this manner *SAGE* may then specialise `Resolve` with respect to each statement in the object program. Specialising a call to `Resolve` with respect to a known statement will remove the vast majority of the expense of the ground representation.

When a call to `Resolve` is specialised wrt a known object statement the specialised code is guaranteed to be a single (possible empty) conjunction of atoms, provided that the user has specified that the WAM-like predicates were to be marked non-selectable. The predicates of these atoms will each be one of `UnifyTerms`, `GetConstant`, `GetFunction`, `UnifyValue`, `UnifyVariable`, `UnifyConstant` or `UnifyFunction`. These predicates perform highly specialised unification operations and together form a crucial part of the implementation of `Resolve`.

The above seven predicates we refer to as the *WAM-like predicates*, as they are analogous to emulators for the WAM instructions GetValue (in the case of `UnifyTerms`), GetConstant, GetFunction, UnifyValue, UnifyVariable and UnifyConstant, after which they are named[1]. As such, these operations could be implemented by Gödel at a very low level, leading to a computation time for the specialised form of a meta-program which would be comparable to that of the object program itself.

[1]Note that a subtle difference in the manner in which the WAM implements the unification of nested function terms and the manner in which `Resolve` implements it means that the WAM does not have an equivalent to the `UnifyFunction` instruction.

**Promoting Symbols from the Ground Representation**

When performing the partial evaluation of a meta-program we recommend that *SAGE* be permitted to unfold any of the 'data-type' predicates `EmptyTermSubst`, `And`, `Or`, `Not`,…etc. The unfolding of such predicates in the definition of `Solve`, for example, will cause the symbols used by Gödel's ground representation to be *promoted* into the heads of the three statements, as can be seen in Figure 5. This would permit the Bristol Gödel to perform first-argument indexing on the specialised definition of this predicate and thus make the execution of this specialised code more efficient.

An important point to note is that the Bristol Gödel only performs indexing on the first argument of the head of a statement. Therefore it is most important to ensure that the argument on which the user of *SAGE* would wish the specialised version of a meta-program to perform indexing will be the first argument in the specialised meta-program. In the unspecialised meta-interpreter `SLD` the desired argument is in fact the second argument in the heads of the statements defining `Solve`. However, as the first argument to the specialised call is instantiated to a term which will be discarded in the specialised program (that is, the representation of the object program which `SLD` is specialised wrt) then we can see that this second argument in the unspecialised definition of `Solve` will be the first argument of the specialised definition of `Solve`, as we would wish.

**Specialising the Module `Solve`**

Figure 5 illustrates the result of specialising the call `Solve(<Append>,g,v,v1,s,s1)`, where `<Append>`

```
Object program:
    MyAppend([],x,x).
    MyAppend([a|x],y,[a|z]) <- MyAppend(x,y,z).

Specialised interpreter:
    Solve_1(Empty, v, v, subst, subst).
    Solve_1(left &' right, v_in, v_out, subst_in, subst_out) <-
      Solve_1(left, v_in, new_v, subst_in, new_subst) &
      Solve_1(right, new_v, v_out, new_subst, subst_out).
    Solve_1(MyAppend'(arg1, arg2, arg3), v_in, v_out, subst_in, subst_out) <-
      GetConstant(arg1, Nil', subst_in, s1) &
      GetValue(arg2, arg3, s1, new_subst) &
      Solve_1(Empty, v_in, v_out, new_subst, subst_out).
    Solve_1(MyAppend'(arg1, arg2, arg3), v_in, v_out, subst_in, subst_out) <-
      GetFunction(arg1, .'(sub11, sub12), mode, subst_in, s1) &
      UnifyVariable(mode, sub11, var, v_in, v1) &
      UnifyVariable(mode, sub12, a1, v1, v2) &
      GetFunction(arg3, .'(sub21, sub22), mode1, s1, s2) &
      UnifyValue(mode1, var, sub21, s2, new_subst) &
      UnifyVariable(mode1, sub22, a2, v2, new_v) &
      Solve_1(MyAppend'(a1, arg2, a2), new_v, v_out, new_subst, subst_out).
```

Figure 5: Specialisation of `Solve` wrt `Append`

is the representation of the `Append` program. In this specialised definition of the `Solve` predicate *SAGE* has made three optimisations:

1. the calls to `Resolve` have been specialised wrt the two statements in the object program to

produce the third and fourth statements respectively in the new predicate `Solve_1`

2. symbols (such as `Empty` and `&'`), which are ordinarily hidden by Gödel's implementation of the ground representation as an abstract data type, have been promoted into the specialised program

3. the representation of the object program `Append`, which is now redundant, has been removed by replacing the predicate `Solve/6` by the new predicate `Solve_1/5`.

## 5.2 The Interpreter `Coroutine`

The second meta-interpreter we consider is, while still a relatively small meta-program, significantly more complex than `SLD` and serves to illustrate more sophisticated meta-programming techniques such as the explicit manipulation of substitutions and implementation of a selection rule.

### 5.2.1 Using *SAGE* on the Module `Coroutine`

The module `Coroutine.loc` is the main module for this interpreter and performs the same I/O and pre- and post-processing functions that were performed by the module `SLD` in the previous section. The two main differences between this module and the `SLD` module are that this module:

1. calls `MySucceed` where the `SLD` module calls `Solve`

2. produces a more sophisticated presentation of the answer-substitution.

The first of these two points merely emphasises that the interpretation performed by `Coroutine` is different to that of `SLD`. We shall examine this in more detail below. The second point, a more sophisticated presentation of the answer-substitution, is illustrative of certain aspects of the manipulation of substitutions in meta-programs and the specialisation of conditionals. We consider this next.

**Partial Evaluation of the Predicate `AnswerString`**

Suppose that we permit *SAGE* to unfold any of the predicates appearing in the definitions for `AnswerString`, `AnswerString1` and `BindingToString`. When specialising the initial call to `AnswerString` the representation of the object program and main module name (that is, the first two arguments) will be known but the other arguments will be unknown. It is apparent therefore that *SAGE* will mark the predicate `AnswerString1` as being unsafe in this partial evaluation. When performing the partial evaluation *SAGE* will therefore unfold the call to `AnswerString` in `Go` but not the call to `AnswerString1`. Instead a separate call to `AnswerString1` with the first two arguments instantiated will be specialised to produce a new recursive definition of this predicate in which the call to `BindingToString` will have been unfolded.

The condition of the conditional in the definition of `BindingToString` will not be unfolded as it not sufficiently instantiated to be executed without binding any free variables. Examining the formulas in the then-part and else-part of this conditional we see that only the equality atoms will be unfolded. This conditional will be replaced by a specialised version which will appear in the definition of the specialised `AnswerString1` predicate. To illustrate these specialisations Figure 6 shows the partial evaluation of the `Coroutine` module wrt the object program `Append`.

As comparatively little specialisation of the predicates `AnswerString1` and `BindingToString` may actually be performed it is possible that a user of *SAGE* might prefer simply to mark either `AnswerString` or `AnswerString1` as non-selectable, thus leaving the residual definitions of these predicates the same as their original definitions. A selection of more effective specialisations of conditionals is illustrated by the partial evaluation of the module `Conc`.

```
MODULE      Coroutine.

IMPORT      Conc, ProgramsIO.

PREDICATE   Go : String * String * String.

Go(program_string, goal_string, answer_string) <-
  FindInput(program_string++".prm", In(is)) &
  GetProgram(is, program) &
  MainModuleInProgram(program, module) &
  StringToProgramFormula(program, module, goal_string, [goal]) &
  FormulaMaxVarIndex([goal], var) &
  EmptyTermSubst(empty) &
  EmptyFormula(e) &
  MySucceed(goal, program, var, _, e, e, empty, answer) &
  AnswerString(program, module, goal, answer, answer_string).

PREDICATE AnswerString : Program * String * Formula * TermSubst * String.

AnswerString(program, module, goal, answer, "{" ++ string ++ "}") <-
  FormulaVariables(goal, vars) &
  AnswerString1(vars, program, module, answer, "", string).

PREDICATE  AnswerString1 : List(Term) * Program * String * TermSubst * String * String.

AnswerString1([], _, _, _, _, "").
AnswerString1([var|rest], program, module, answer, comma, string ++ rest_string) <-
  ApplySubstToTerm(var, answer, var_answer) &
  BindingToString(var, var_answer, program, module, comma, comma1, string) &
  AnswerString1(rest, program, module, answer, comma1, rest_string).

PREDICATE  BindingToString : Term * Term * Program * String * String * String * String.

BindingToString(var, var_answer, program, module, comma, comma1, string) <-
  IF  var = var_answer
    THEN  comma1 = comma &
          string = ""
    ELSE  ProgramTermToString(program, module, var, var_string) &
          ProgramTermToString(program, module, var_answer, answer_string) &
          comma1 = "," &
          string = comma ++ var_string ++ "/" ++ answer_string.
```

```
PREDICATE   Go_1 : String * String.

Go_1(goal_string, "{" ++ answer_string ++ "}") <-
  StringToProgramFormula(<Append>, "Append", goal_string, [goal]) &
  MySucceed_1(goal, var, _, Empty, _, EmptyTermSubst, answer) &
  FormulaVariables(goal, vars) &
  AnswerString1_1(vars, answer, "", answer_string).

PREDICATE  AnswerString1_1 : List(Term) * TermSubst * String * String.

AnswerString1_1([], _, _, "") <-
AnswerString1_1([var|rest], answer, comma, string ++ string1) <-
  ApplySubstToTerm(var, answer, var_answer) &
  (
  IF  Test_0(var, var_answer)
    THEN  Then_1(comma, comma1, string)
    ELSE  Else_2(var, var_answer, comma, comma1, string)
  ) &
  AnswerString1(rest, answer, comma1, string1).

PREDICATE  Test_0 : Term * Term.

Test_0(var, var).

PREDICATE  Then_1 : String * String * String.

Then_1(comma, comma, "").

PREDICATE  Else_2 : Term * Term * String * String * String.

Else_2(var, var_answer, comma, ",", comma ++ var_string ++ "/" ++ answer_string) <-
  ProgramTermToString(<Append>, "Append", var, var_string) &
  ProgramTermToString(<Append>, "Append", var_answer, answer_string).
```

Figure 6: Specialised Version of Module Coroutine

### 5.2.2 Using *SAGE* on the Module `Conc`

**Note: This section of the guide has yet to be completed.**

```
EXPORT      Conc.

PREDICATE  MySucceed : Formula * Program * Integer * Integer * Formula * Formula
                       * TermSubst * TermSubst.
```

```
LOCAL       Conc.

IMPORT      Programs.

BASE Bind.

FUNCTION ! : xFx(200) :  Term * Term -> Bind.

MySucceed(formula, _, var, var, _, _, answer, answer) <-
  EmptyFormula(formula) |.
MySucceed(formula, program, var, var1, l, r, answer_so_far, answer) <-
  Atom(formula) |
  MyStatementMatchAtom(program, formula, clause) &
  Resolve(formula, clause, var, new_var, answer_so_far,
                                        new_answer, new_body) &
  (
  IF  EmptyFormula(new_body)
    THEN  AndWithEmpty(l, r, nb)
    ELSE  AndWithEmpty(l, new_body, nb1) &
          AndWithEmpty(nb1, r, nb)
  ) &
  Select(nb, new_answer, program, l1, s1, r1) &
  MySucceed(s1, program, new_var, var1, l1, r1, new_answer, answer).

PREDICATE  MyStatementMatchAtom : Program * Formula * Formula.

MyStatementMatchAtom(program, atom, statement) <-
  { PredicateAtom(atom, pred, _) \/
    PropositionAtom(atom, pred) } &
  DefinitionInProgram(program, _, pred, defn) &
  Member(statement, defn).

PREDICATE  Select : Formula * TermSubst * Program * Formula * Formula * Formula.

Select(formula, _, _, formula, formula, formula) <-
   EmptyFormula(formula) |.
Select(formula, subst, program, left, selected, right) <-
   And(l, r, formula) |
   IF SOME [l1, s1, r1] Select(l, subst, program, l1, s1, r1)
     THEN
```

```
       left = l1 &
       selected = s1 &
       AndWithEmpty(r1, r, right)
     ELSE
       Select(r, subst, program, l1, selected, right) &
       AndWithEmpty(l, l1, left).
Select(formula, subst, program, empty, formula, empty) <-
  Atom(formula) |
  EmptyFormula(empty) &
  CanRunAtom(program, formula, subst).

PREDICATE  CanRunAtom : Program * Formula * TermSubst.

CanRunAtom(program, atom, sss) <-
  PredicateAtom(atom, predicate, _) &
  ProgramPredicateName(program, module, _, _, predicate) &
  IF SOME  [heads, conditions, n]
    ( ControlInProgram(program, module, predicate, heads, conditions) &
      Length(heads, n) & n > 0 )
  THEN
    GetDelay(heads, conditions, atom, sss).

PREDICATE GetDelay :
   List(Formula) * List(Condition) * Formula * TermSubst.

GetDelay([head|_], [condition|_], atom, subst) <-
  InstanceOfHead(head, atom, subst, sss) &
  ConditionSatisfied(condition, sss).
GetDelay([_|rh], [_|rc], atom, sss) <-
  GetDelay(rh, rc, atom, sss).

PREDICATE InstanceOfHead : Formula * Formula * TermSubst * List(Bind).

InstanceOfHead(head, atom, subst, bind) <-
   PredicateAtom(head, name, head_args) &
   PredicateAtom(atom, name, args) &
   InstanceOfHead2(head_args, args, subst, [] , bind).

PREDICATE  InstanceOfHead1 : Term * Term * List(Bind) * List(Bind).

InstanceOfHead1(head_arg, arg, b, [head_arg ! arg|b]) <-
  Variable(head_arg) |.
InstanceOfHead1(head_arg, arg, subst, subst) <-
  ConstantTerm(head_arg, name) |
  ConstantTerm(arg, name).
InstanceOfHead1(head_arg, arg, subst, new_subst) <-
  FunctionTerm(head_arg, name, head_args) |
  FunctionTerm(arg, name, args) &
  InstanceOfHead3(head_args, args, subst, new_subst).

PREDICATE InstanceOfHead2 : List(Term) * List(Term) * TermSubst
                            * List(Bind) * List(Bind).
```

```
InstanceOfHead2([], [], _, subst, subst).
InstanceOfHead2([head_arg|head_args], [arg|args], sss, subst, new_subst) <-
  ApplySubstToTerm(arg, sss, arg1) &
  InstanceOfHead1(head_arg, arg1, subst, subst1) &
  InstanceOfHead2(head_args, args, sss, subst1, new_subst).

PREDICATE InstanceOfHead3 : List(Term) * List(Term) * List(Bind) * List(Bind).

InstanceOfHead3([], [], subst, subst).
InstanceOfHead3([head_arg|head_args], [arg|args], subst, new_subst) <-
  InstanceOfHead1(head_arg, arg, subst, subst1) &
  InstanceOfHead3(head_args, args, subst1, new_subst).

PREDICATE  ConditionSatisfied : Condition * List(Bind).

ConditionSatisfied(cond, subst) <-
  GroundCondition(var, cond) |
  Member(var ! term, subst) &
  GroundTerm(term).
ConditionSatisfied(cond, subst) <-
  OrCondition(cond1, cond2, cond) |
  OrConditionSatisfied(cond1,cond2, subst).
ConditionSatisfied(cond, subst) <-
  AndCondition(cond1, cond2, cond) |
  ConditionSatisfied(cond1, subst) &
  ConditionSatisfied(cond2, subst).
ConditionSatisfied(cond, subst) <-
  NonVarCondition(var, cond) |
  Member(var ! term, subst) &
  NonVariable(term).
ConditionSatisfied(cond, subst) <-
  TrueCondition(cond).

PREDICATE  OrConditionSatisfied : Condition * Condition * List(Bind).

OrConditionSatisfied(cond, _, subst) <-
  ConditionSatisfied(cond, subst).
OrConditionSatisfied(_, cond, subst) <-
  ConditionSatisfied(cond, subst).
```

---

## References

[1] D Chan and M Wallace. A treatment of negation during partial evaluation. In H D Abramson and M H Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 299–318. MIT Press, 1989.

[2] C A Gurr. Specialising the Ground Representation in the Logic Programming Language Gödel. In *Proceedings of the Third International Workshop on Logic Program Synthesis and Transformation.* Springer-Verlag, July 1993.

[3] C A Gurr. A Self-Applicable Partial Evaluator for the Logic Programming Language Goedel (Extended Abstract). Technical Report Draft, Human Communication Research Centre, University of Edinburgh, 1994.

[4] C A Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[5] P M Hill and J W Lloyd. The Gödel Programming Language. Technical Report CSTR-92-27, Department of Computer Science, University of Bristol, 1992. Revised May 1993. To be published by MIT Press.

[6] P M Hill, J W Lloyd, and J C Shepherdson. Properties of a pruning operator. *Journal of Logic and Computation*, 1(1):99–143, 1990.

[7] J W Lloyd and J C Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.