A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.

Corin Alistair Gurr

A thesis submitted to the University of Bristol in accordance with the requirements of the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

January 1994

Abstract

Partial evaluation is a program specialisation technique that has been shown to have great potential in logic programming, particularly for the specialisation of meta-interpreters by the so-called "Futamura Projections". Meta-interpreters and other meta-programs are programs which use another program as data. In this thesis we describe a partial evaluator for meta-programs in the logic programming language Gödel.

Gödel is a declarative, general-purpose language which provides a number of higher-level programming features, including extensive support for meta-programming with a *ground representation*. The ground representation is a standard tool in mathematical logic in which object level variables are represented by ground terms at the meta-level. The ground representation is receiving increasing recognition as being essential for declarative meta-programming, although the computational expense that it incurs has largely precluded its use in the past.

This thesis extends the basic techniques of partial evaluation to the facilities of Gödel. Particular attention is given to the specialisation of the inherent overheads of meta-programs which use a ground representation and the foundations of a methodology for Gödel meta-programs are laid down. The soundness of the partial evaluation techniques is proved and these techniques are incorporated into a declarative partial evaluator.

We describe the implementation and provide termination and correctness proofs for the partial evaluator *SAGE*, an automatic program specialiser based upon sound finite unfolding that is able to specialise any Gödel meta-program (or indeed, any Gödel program at all). A significant illustration of the success of our techniques for specialising meta-programs which use a ground representation is provided by the self-application of this partial evaluator. We use the partial evaluator to specialise itself with respect to a range of meta-programs. By virtue of its self-applicability *SAGE* has been used to produce a *compiler-generator*, which we believe shall prove to be an immensely powerful and useful tool for meta-programming.

Declaration

The work in this thesis is the independent and original work of the author, except where explicit reference to the contrary has been made. No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or institution of education.

C A Gurr

Acknowledgements

Let us, then, be up and doing,
With a heart for any fate;
Still achieving, still pursuing,
Learn to labour and to wait.

From *A Psalm of Life* by Henry Wadsworth Longfellow, 1807-1882.

The past three years have been for me a time that was by turns both immensely enjoyable and immensely frustrating. Many people who have provided invaluable support and advice during this time have contributed both to my enjoyment and my frustration and I take this opportunity to thank them for their assistance and to reassure them that any frustration caused has lessened neither the sense of pleasure nor the sense of achievement that producing this thesis has given me.

First and foremost I would like to thank John Lloyd my supervisor, without whose knowledge, guidance, unshakeable optimism and merciless attention to detail, this work would not have been possible.

From the members of the Gödel group at the University of Bristol, special thanks go to Antony Bowers and Jiwei Wang for their constant hard work on the implementation and frequent fixing of the Gödel language. Particular thanks go to Antony Bowers who contributed to the design of the ground representation and who gave me invaluable advice in discussions on efficient meta-programming with the ground representation. Thanks are given also to the other members of the Gödel group over the last three years and to the many people with whom I have had interesting discussions on the Gödel language, meta-programming and program specialisation. These include Dominic Binks, John Gallagher, Ahmed Guessoum, Pat Hill, John Lloyd, Nick Moffat and André de Waal.

Finally I would like to thank my family and friends who have both supported and encouraged me over the last few years. In particular I would like to thank Nicki for putting up with me on the bad days and for being genuinely pleased for me on the good days.

This work was supported by an SERC studentship award.

Contents

List of Figures **vi**

Glossary **viii**

1 Introduction **1**

1.1 Meta-Programming . . . 3

1.2 Gödel . . . 4

1.2.1 Modules and Types . . . 4

1.2.2 Control . . . 8

1.2.3 Meta-Programming . . . 11

1.3 Partial Evaluation . . . 12

1.4 Self-Application . . . 13

1.4.1 Specialising Meta-Programs: The Three Futamura Projections . . . 14

1.4.2 What *is* a Compiler-Generator? . . . 15

2 Specialising Gödel Programs **18**

2.1 Connectives, Quantifiers and Conditionals . . . 18

2.1.1 Connectives . . . 18

2.1.2 Negation . . . 19

2.1.3 Quantifiers . . . 19

2.1.4 Conditionals . . . 20

2.2 Type Declarations . . . 22

2.3 Control . . . 22

2.3.1 Computation Rule . . . 23

2.3.2 Pruning . . . 23

2.4 Open and Closed Code . . . 39

2.5 Input/Output . . . 40

2.6 Modules . . . 41

3 Specialising Gödel Meta-Programs **45**
3.1 The Ground Representation in Gödel . . . 45
3.2 Specialising the Representation of Gödel . . . 47
3.3 Specialising Resolution in the Ground Representation . . . 48
3.3.1 Specialising Resolve . . . 50
3.3.2 Implementing `Resolve` . . . 58
3.3.3 Handling Committed Formulas with `ResolveAll` . . . 60
3.3.4 Specialising Unification . . . 64
3.4 A Framework for Meta-Programs . . . 64
3.4.1 Structure of Meta-Interpreters . . . 65
3.4.2 An (Intuitive) Ideal Specialisation . . . 65
3.4.3 Avoiding Code Explosion: Structure of Residual Programs . . . 67
3.4.4 Implementing a Selection Strategy . . . 70
3.4.5 Removing Redundant Terms . . . 73

4 The Anatomy of *SAGE* **74**
4.1 Constructing a Self-Applicable Partial Evaluator . . . 75
4.1.1 Specialising a Self-Applicable Partial Evaluator . . . 76
4.1.2 A Framework for Selection in Partial Evaluation . . . 77
4.1.3 Specialising the Selection Strategy . . . 81
4.2 The Selection Strategy for *SAGE* . . . 81
4.2.1 Overview of Static Selection Strategy for *SAGE* . . . 83
4.2.2 Abstract Partial Evaluations . . . 86
4.2.3 Computing Abstract Call Patterns . . . 90
4.2.4 Static Analysis Algorithm . . . 92
4.2.5 Updating Abstract Call Patterns . . . 94
4.2.6 The Dynamic Selection Strategy for *SAGE* . . . 97
4.2.7 Justifying *SAGE*'s Selection Strategy . . . 102
4.3 The Unfolding Strategy for *SAGE* . . . 105
4.3.1 A Helpful Representation for Resultants . . . 105
4.3.2 Unfolding Atoms . . . 105
4.3.3 Unfolding Negated Formulas . . . 107
4.3.4 Unfolding Conditional Formulas . . . 109
4.3.5 Unfolding Committed Formulas . . . 111
4.4 Assembling The Residual Script . . . 118
4.4.1 Converting Programs to Scripts . . . 118

4.4.2 Optimising Residual Code . . . 119
4.4.3 Updating Declarations . . . 121
4.4.4 Assembling the Script . . . 124
4.5 Termination and Correctness of *SAGE* . . . 124
4.5.1 Termination of *SAGE* . . . 124
4.5.2 Correctness of *SAGE* . . . 124

5 Results and Conclusions **127**
5.1 Results . . . 127
5.1.1 The First Futamura Projection . . . 127
5.1.2 The Missing Link . . . 137
5.1.3 The Second and Third Futamura Projections . . . 143
5.2 Future Work . . . 147
5.2.1 Extending *SAGE* . . . 147
5.2.2 Delays In Gödel . . . 150
5.2.3 The Full Commit . . . 150
5.2.4 Opening the System Modules . . . 150
5.3 Related Work . . . 151
5.3.1 Partial Evaluation Techniques . . . 151
5.3.2 Automatic and Sound Finite Unfolding . . . 152
5.3.3 Termination Analysis . . . 153
5.3.4 Specialising Meta-Interpreters . . . 153
5.3.5 Specialising Resolution . . . 154
5.3.6 Full Specialisation and Self-Application . . . 155
5.4 Contributions . . . 156
5.4.1 Specialising Gödel Programs . . . 156
5.4.2 Implementing the Ground Representation . . . 157
5.4.3 A Methodology of Meta-Programming . . . 157
5.4.4 A Framework for Self-Applicability . . . 158
5.4.5 Generating a Compiler-Generator . . . 159
5.4.6 Conclusions . . . 159

A The Implementation of Substitutions **161**
A.1 `UnifyTerms`, `UnifyTypes` and `UnifyAtoms` . . . 161
A.2 Representing Substitutions . . . 163
A.2.1 Applying Variable Bindings . . . 163
A.2.2 Adding Variable Bindings . . . 165

A.2.3 Implementing Substitutions . . . 166
A.2.4 Implementing `ComposeTermSubsts` . . . 168
A.3 The WAM-like Predicates . . . 169

Bibliography **172**

List of Figures

2.1 Constructive Unfolding of Negation . . . 19
2.2 Unfolding `IF-THEN-ELSE` . . . 21
2.3 A Generalised Irregular cSLDNF-tree . . . 34

3.1 Gödel code for `VarsInTerm` . . . 46
3.2 Specialised code for `VarsInTerm1(term,vars,vars1)` . . . 48
3.3 A Simple Gödel Meta-Interpreter . . . 49
3.4 A Naive Gödel Meta-Interpreter . . . 50
3.5 A Gödel Statement . . . 51
3.6 Specialised code for `Resolve` . . . 55
3.7 More specialised code for `Resolve` . . . 56
3.8 Basic Structure for Meta-Interpreters . . . 66
3.9 Specialised Structure of Meta-Interpreters . . . 67
3.10 Unfolding of `Demo1` in `Demo` . . . 68
3.11 Code for `SimpleResolve` . . . 69
3.12 A Simple Leftmost Literal Selection Function . . . 71
3.13 A Sophisticated Leftmost Literal Selection Function . . . 71
3.14 An Efficient Leftmost Literal Selection Function . . . 72

4.1 Static Selection Algorithm . . . 79
4.2 Dynamic Selection Algorithm . . . 80
4.3 Pseudo-code for top level of *SAGE* . . . 82
4.4 Static Analysis Procedure . . . 84
4.5 Predicate Dependency Graph . . . 85
4.6 Updating Pattern for Base Case . . . 95
4.7 Updating Pattern for Recursive Case . . . 96
4.8 A Basic Gödel Meta-Interpreter . . . 104
4.9 Algorithm for *SAGE* . . . 125

5.1 The First Futamura Projection 137
5.2 The Second Futamura Projection 145
5.3 The Second Futamura Projection minus Garbage-Collection 146
5.4 The Third Futamura Projection 146
5.5 The Third Futamura Projection minus Garbage-Collection 147

Glossary

{...} 9
{...}_n 9
| 9

A-closed 13
A-covered 13
abstract term 86
abstract unification 86
abstract atom 86
abstract partial evaluation 83
abstract resolution 86
abstract substitution 86
abstraction 98
accessible 8
apply pattern 91
array-substitution 164

base type 5

c-resultant 26
category of symbol 5
closed code 40
closed module 8, 40
code explosion 65
commit 9
commit label 9
complete cSLDNF-tree 25
complete unfolding 88
composition 165
computed answer 12
constructor type 5
covering atom 78, 90
cSLDNF-tree 24
cut node 25

demote 41
depends 13
dereference 165
dereferenced value 165
descendant 100
direct ancestor 87

effective self-application 1, 15
explanation 25
explicit delay 140
extension 37

failed branch 25
failure-driven programming 71
First Futamura Projection 14
flattening of modules 43
forward-driven programming 72
freeness condition 26, 27
full commit 9
fundamental selection strategy 77

ground representation 3

head atom 87
head condition 22

idempotent substitution 168
immediate parent 87

implicit delay . . . 140
import . . . 6
independent . . . 13
information-set . . . 78
input pattern . . . 90
input/output module . . . 40
irregular unfolding step . . . 29

l-child . . . 25
L-compatible . . . 77
L-selectable . . . 77
length of derivation . . . 12
list-substitution . . . 163

match pattern . . . 91

non-ground representation . . . 3

open code . . . 40
open module . . . 40
ordering of abstract terms . . . 87, 91
output pattern . . . 90

parameter . . . 6
partial evaluation . . . 12, 26
pattern . . . 90
pattern term . . . 86
promote . . . 41
prunable . . . 25
pruned . . . 25
pruning step . . . 25

rationalisation . . . 168
recursive occurrence . . . 87
regular unfolding step . . . 29
regularity condition . . . 26, 28
resultant . . . 12

safe predicate . . . 78
scope of a bar commit . . . 9
script . . . 44
Second Futamura Projection . . . 14
selectable predicate . . . 77
strict subterm . . . 91
success branch . . . 25
symbol . . . 4
system module . . . 6, 40

Third Futamura Projection . . . 15
typed representation . . . 3

uncomposed substitution . . . 165
unessential binding . . . 108
unprunable . . . 25
unsafe predicate . . . 78
user-defined module . . . 40

variable index . . . 51
variable root . . . 51
variable typing . . . 119
variable-free abstract term . . . 86

WAM-like predicate . . . 55
well-structured . . . 77

Chapter 1

Introduction

Partial evaluation is a program specialisation technique that has been shown to have great potential in logic programming, particularly for the specialisation of meta-interpreters. It was explicitly introduced into Computer Science by Futamura [20] and into logic programming by Komorowski [39], for which it was put on a sound theoretical footing by Lloyd and Sheperdson [47]. In the context of [47] 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. A recent overview and bibliography for partial evaluation are given by [34] and [63] respectively.

The logic programming community's interest in partial evaluation stems primarily from the first Futamura projection, which illustrates how partial evaluation may be used to compile programs by the specialisation of meta-interpreters. The second Futamura projection shows that if the partial evaluator is self-applicable (able to specialise itself) a compiler may be produced.[18, 33, 35, 62, 67]. This is taken one stage further in the third Futamura projection, where a *compiler-generator* is produced by specialising the partial evaluator with respect to itself.

This thesis describes the development of a partial evaluator for meta-programs in the logic programming language Gödel [28]. A key aim for this thesis has been the construction of a declarative self-applicable partial evaluator, written in a logic programming language, which was capable of specialising any program in the language in which it was written. To date this result has been achieved in a functional programming language by Neil Jones *et al* [35] and several attempts have been made to construct such a program in a logic programming language [18, 19, 51]. However, these partial evaluators have been constructed in the Prolog language and, due to the non-logical features of Prolog, have only considered restricted subsets of the language and do not generally have a declarative semantics. Specialising full Prolog and the construction of an *effective* (capable of producing efficient results) self-applicable Prolog partial evaluator are tasks that are made most difficult by Prolog's non-logical features. This is illustrated by the sophistication needed to specia-

lise these features.[59, 72]. It is our belief that an effectively self-applicable partial evaluator is an immensely powerful and useful tool for meta-programming in logic programming.

As a declarative alternative to Prolog, we have chosen to implement our partial evaluator in Gödel. Gödel is a declarative, general-purpose language which provides a number of higher-level programming features, including extensive support for meta-programming with a *ground representation.* The ground representation is a standard tool in mathematical logic in which object level variables are represented by ground terms at the meta-level. The ground representation is receiving increasing recognition as being essential for declarative meta-programming, although the computational expense that it incurs has largely precluded its use in the past.

To achieve the above aim we have extended the basic techniques of partial evaluation to the facilities of Gödel. Particular attention has been given to the specialisation of the inherent overheads of meta-programs which use a ground representation and to the development of the foundations of a methodology for Gödel meta-programs.

We may summarise our three main aims as being to:

1. develop techniques to allow the specialisation of the full Gödel language
2. develop an implementation and a methodology for meta-programming with the ground representation which was
   - efficient
   - amenable to specialisation
3. design and implement an effectively self-applicable declarative partial evaluator in Gödel.

The layout of this thesis is as follows, in this chapter we discuss the concept of meta-programming in logic programming. We also introduce the logic programming language Gödel, define the basic theoretical results underlying partial evaluation and finally discuss the potential for a self-applicable partial evaluator. In Chapter 2 we describe the techniques employed by the partial evaluator *SAGE* to specialise the facilities and features of Gödel, other than Gödel's meta-programming facilities. Chapter 3 is dedicated to describing the specialisation of meta-programs in Gödel. In Chapter 4 we examine in more detail the partial evaluator *SAGE*. We describe its approach to the partial evaluation of a program and the assembly of the specialised version of a program. Lastly in Chapter 4 we prove the soundness and termination of *SAGE*. Finally, in Chapter 5, we present the results produced by *SAGE* for the specialisation of a range of meta-programs, including *SAGE* itself. We discuss the implications of these results and the scope for further work based upon this foundation. In the appendix we present in more detail the implementation of those parts of the ground representation relevant to *SAGE* and the implementation of *SAGE*.

1.1 Meta-Programming

A meta-program is essentially any program which uses another program (the object program) as data. Clearly many of the major applications of logic programming, such as knowledge base systems, interpreters, compilers, debuggers, program transformers and theorem provers will be meta-programs.

A key issue for meta-programming is the representation of the object programs, formulas and terms. Two main approaches to representation have been identified and these are referred to as the *non-ground* and *ground* representations respectively. The key difference between these two approaches is in the representation of object level variables. In the non-ground representation object level variables are represented by variables at the meta-level, while in the ground representation object level variables are represented by ground terms at the meta-level.

The use of a ground representation for meta-programming is a standard tool in mathematical logic which first appeared in logic programming in [6] and the theoretical foundations for meta-programming were laid in [27, 26]. In [27] the differences between the non-ground, referred to in that paper as *typed*, and the ground representations were discussed and it was shown that the ground representation is the more powerful of the two. The non-ground representation has severe semantic problems which make it unlikely that it could be used for any serious meta-programming in a declarative way. Consequently considerable effort has been devoted to developing Gödel [28], a declarative logic programming language which provides facilities to support meta-programming with a ground representation.

Although the ground representation is increasingly being recognised as being essential for declarative meta-programming, the expense that is incurred by the use of such a representation has largely precluded its use in the past. Using a ground representation means that unification, particularly the binding of variables (that is, substitutions), must be handled explicitly by the meta-program. Programmers are unable to rely upon the underlying system to perform unification for them. This can cause considerable execution overheads in meta-programs. In this thesis we describe the partial evaluator *SAGE* (Self-Applicable Gödel partial Evaluator), a Gödel program specialiser which is capable of optimising Gödel meta-programs so as to remove the majority of these overheads. With *SAGE* we are able to produce optimised Gödel meta-programs which can potentially execute in a time comparable to that of the object programs which they manipulate.

1.2 Gödel

Gödel is a declarative, general-purpose logic programming language whose main facilities are:

- modules
- types
- control
  - constraint solving
  - control declarations
  - pruning operator
- meta-programming
- input/output

Clearly in this thesis we attach the most importance to the meta-programming facilities that Gödel provides and Chapter 3 is devoted to the specialisation of these facilities. The specialisation of all other Gödel facilities must also be taken into account however, and this is discussed in Chapter 2.

In this section we give a brief overview of the above facilities and illustrate with some example Gödel programs. The example programs and substantial portions of this overview are taken directly from [28].

1.2.1 Modules and Types

A Gödel module consists of module declarations, language declarations, control declarations, and statements. Module declarations name modules and declare which symbols of the language are imported and exported. Language declarations define a polymorphic many-sorted language. Control declarations constrain the computation rule. Statements are the formulas in the language which define the propositions and predicates. To begin with we shall only deal with the simplest of Gödel modules, those whose module declaration consists of the keyword `MODULE` followed by the name of the module. For example, the following module has name `M1`.

We now turn to Gödel's type system before returning to a fuller description of the module system. Gödel's type system is based on many-sorted logic with parametric polymorphism [30]. Consider module `M1` below which defines the predicates `Append` and `Append3` for appending lists of days of the week. Note that variables are denoted by identifiers beginning with a lower case letter and constants by identifiers beginning with an upper case letter.

In general, language declarations begin with the keywords `BASE`, `CONSTRUCTOR`, `CONSTANT`, `FUNCTION`, `PROPOSITION`, or `PREDICATE`. These declarations declare the *symbols* of the language,

```
MODULE        M1.

BASE          Day, ListOfDay.

CONSTANT      Nil : ListOfDay;
              Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday :
              Day.
FUNCTION      Cons : Day * ListOfDay -> ListOfDay.
PREDICATE     Append : ListOfDay * ListOfDay * ListOfDay;
              Append3 : ListOfDay * ListOfDay * ListOfDay * ListOfDay.

Append(Nil,x,x).
Append(Cons(u,x),y,Cons(u,z)) <-
              Append(x,y,z).

Append3(x,y,z,u) <-
              Append(x,y,w) &
              Append(w,z,u).
```

which belong in one of the *categories*: base, constructor, constant, function, proposition, or predicate. In module `M1`, the language declaration beginning with the keyword `BASE` gives the types of the many-sorted language of the module. It declares `Day` and `ListOfDay` to be *bases*, which are the only types of the language. The next three declarations declare the constants, functions, and predicates of the language. The first part of the `CONSTANT` declaration declares `Nil` to be a constant of type `ListOfDay`. The second part declares `Monday`, `Tuesday`, etc., to be constants of type `Day`. The `FUNCTION` declaration declares `Cons` to be a binary function which maps a tuple of arguments, where the first argument is of type `Day` and the second argument is of type `ListOfDay`, to an element of type `ListOfDay`. The `PREDICATE` declaration declares `Append` to be a ternary predicate each of whose arguments has type `ListOfDay`. It also declares `Append3` to be a quaternary predicate each of whose arguments has type `ListOfDay`. Statements and goals are written in the language defined by the language declarations.

Next we introduce *constructors* using module `M2`, which is a variation of module `M1`. The main difference between the two modules is that in module `M2` a unary constructor `List` has been declared. From the base `Day` and the constructor `List`, the set of all types of the language is obtained by forming all "ground terms" from the "constant" `Day` and the "function" `List`. Thus the types of the language are `Day`, `List(Day)`, `List(List(Day))`, ... Note that a constructor itself is not a

type.

```
MODULE        M2.

BASE          Day.
CONSTRUCTOR   List/1.

CONSTANT      Nil : List(Day);
              Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday :
              Day.
FUNCTION      Cons : Day * List(Day) -> List(Day).
PREDICATE     Append : List(Day) * List(Day) * List(Day);
              Append3 : List(Day) * List(Day) * List(Day) * List(Day).

Append(Nil,x,x).
Append(Cons(u,x),y,Cons(u,z)) <-
              Append(x,y,z).

Append3(x,y,z,u) <-
              Append(x,y,w) &
              Append(w,z,u).
```

Gödel also allows symbols to be defined which have a variety of types. For example, the `Append` predicate is normally written so that it can append lists of any type. For this purpose Gödel allows *parameters* in language declarations, where a parameter may be instantiated to any type. For example, in module `M3`, `a` is a parameter.

Gödel provides a range of *system* modules which define the types for a variety of data structures and the operations upon them. These include, among others, the modules `Integers`, `Lists`, `Sets`, and `Strings`. Gödel also provides system modules such as `Syntax`, `Programs` and `Theories` to support meta-programming and modules such as `IO`, `NumbersIO` and `ProgramsIO` to support input/output.

We now describe Gödel's module structure in more detail. In general, modules consist of two parts, a local part and an export part. The local part of a module contains the code for all the predicates and propositions declared in that module and language declarations for all symbols which are only used by that module. The export part of a module contains the language declarations for those symbols which may be used by another module. A module may use the symbols from another module by *importing* that module. A module imports another module by declaring that it

```
MODULE        M3.

BASE          Day, Person.
CONSTRUCTOR   List/1.

CONSTANT      Nil : List(a);
              Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday :
              Day;
              Fred, Bill, Mary : Person.
FUNCTION      Cons : a * List(a) -> List(a).
PREDICATE     Append : List(a) * List(a) * List(a);
              Append3 : List(a) * List(a) * List(a) * List(a).

Append(Nil,x,x).
Append(Cons(u,x),y,Cons(u,z)) <-
              Append(x,y,z).

Append3(x,y,z,u) <-
              Append(x,y,w) &
              Append(w,z,u).
```

imports that module.

For example, the module `M5` has a local part which begins with the module declaration `LOCAL` and an export part which begins with the module declaration `EXPORT`. `M5` imports the Gödel system module `Lists` by means of the import declaration `IMPORT Lists`. As the system module `Lists` declares the predicate `Append` in its export part, this symbol may be used by the module `M5`. We say that this symbol is *accessible* to module `M5`. The concept of the *accessibility* of symbols is described in more detail in [28].

```
EXPORT       M5.

IMPORT       Lists.

BASE         Day, Person.

CONSTANT     Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday :
             Day;
             Fred, Bill, Mary : Person.
PREDICATE    Append3 : List(a) * List(a) * List(a) * List(a).
```

```
LOCAL        M5.

Append3(x,y,z,u) <-
             Append(x,y,w) &
             Append(w,z,u).
```

There are two other kinds of modules worth mentioning at this point. The first is a module which consists of a local part only. The local part of such a module will begin with the module declaration `MODULE`, as in the modules `M1`, `M2` and `M3`. The second kind of module is one which has the module declaration `CLOSED` instead of `EXPORT` in its export part. We refer to such a module as a *closed module*. Only system modules may be closed. The implications of closed modules for partial evaluation is discussed in the next chapter.

1.2.2 Control

There are three aspects to control in Gödel: constraints, the computation rule and pruning.

Gödel has constraint-solving capabilities in the domains of integers and rationals. These are provided via closed Gödel system modules and therefore their specialisation is dealt with as with all closed modules. This is described in more detail in the following chapter and we shall not dwell on it here.

The second aspect of control in Gödel is Gödel's flexible computation rule for determining which literal in the current goal will be selected. The computation rule is partly under the control of the programmer through implicit `DELAY` declarations, which may be defined for any predicate in a program. The `DELAY` declarations specify conditions under which an atom may be selected by the computation rule and the relevance of this to partial evaluation is discussed in the following chapter.

The third aspect of control in Gödel is Gödel's pruning operator, called *commit*, and this bears a more detailed overview than the previous two aspects.

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* and to the above integer n as the *commit label*. We begin by explaining the more familiar `|` commit and then we discuss the `{...}` commit.

The module `P1` defines the predicate `Quicksort` which is intended to be called with its first argument instantiated. Module `P1` uses the `|` version of the commit, which we call the *bar* commit. Declaratively, `|` is just conjunction. However, for convenience, if either argument of `|` is `True` it can be omitted, as in the first statement for `Quicksort3`. Each statement can contain at most one `|`. The *scope* of `|` is the formula to its left in the body of the statement. The order in which the statements are tried is not specified, so that commit does not have the sequentiality property of Prolog's cut. The procedural meaning of `|` is that only one solution is found for the formula in its scope and all other branches arising from the other statements in the definition which contain a `|` are pruned. Thus the meaning of `|` is close to the commit of the concurrent logic programming languages. Note that, while a `|` commit would normally appear in *every* statement of a definition for which at least one `|` appears, this is not obligatory.

The Gödel one-solution commit written `{...}`, is used to prune the formula within its scope so that only one answer for this formula is computed.

The more general form of the commit, the *full commit*, is not directly supported by Gödel and is intended mainly to be used in the results of partial evaluation. The need for the full commit in partial evaluation and the theoretical foundation for it are presented in [47]. In that paper the partial evaluation theorem of [29] is extended to cover programs containing commits. In Section 2.3 we present this extension to the partial evaluation theorem and describe in detail *SAGE*'s specialisation of Gödel's commit operator.

```
MODULE      P1.

IMPORT      Lists.

PREDICATE   Quicksort : List(Integer) * List(Integer).
DELAY       Quicksort(x,_) UNTIL NONVAR(x).

Quicksort(x,y) <-
            Quicksort3(x,y,[]).

PREDICATE   Quicksort3 : List(Integer) * List(Integer) * List(Integer).
DELAY       Quicksort3(x,_,_) UNTIL NONVAR(x).

Quicksort3([],xs,xs) <-
            |.
Quicksort3([x|xs],ys,zs) <-
            |
            Partition(xs,x,l,b) &
            Quicksort3(l,ys,[x|ys1]) &
            Quicksort3(b,ys1,zs).

PREDICATE   Partition : List(Integer) * Integer * List(Integer) * List(Integer).
DELAY       Partition([],_,_,_) UNTIL TRUE;
            Partition([u|_],y,_,_) UNTIL NONVAR(u) & NONVAR(y).

Partition([],_,[],[]) <-
            |.
Partition([x|xs],y,[x|ls],bs) <-
            x =< y |
            Partition(xs,y,ls,bs).
Partition([x|xs],y,ls,[x|bs]) <-
            x > y |
            Partition(xs,y,ls,bs).
```

1.2.3 Meta-Programming

Gödel provides a ground representation for meta-programming which enables users to write meta-programs that:

- Have a declarative semantics.
- Are clearly readable and straightforward to write.
- Are potentially comparable, in execution time, to Prolog meta-programs which use the non-ground representation.

Gödel's ground representation is presented to the user via an abstract data type, thus avoiding the need for the user to have knowledge of its implementation and therefore not confusing the user with a profusion of constant and function symbols. In addition to this, the development of large meta-programming applications such as interpreters, theorem provers, partial evaluators and debuggers in Gödel have influenced the development of Gödel's ground representation, so that a natural and clearly readable style of meta-programming with the ground representation is now emerging. This is exemplified by the comparison between the 'naive' Gödel meta-interpreter in figure 3.4, where unification and resolution are handled explicitly in the code, and the more natural meta-interpreter of figure 3.3, where resolution is handled implicitly by the Gödel system predicate `Resolve`, discussed in more detail in Section 3.3.1. Other example meta-programs may be seen in Section 5.1.1.

The partial evaluator *SAGE* is designed primarily to specialise Gödel meta-programs and therefore a detailed overview of meta-programming in Gödel and *SAGE*'s approach to specialising meta-programs is provided in Chapter 3.

The development of *SAGE* has progressed concurrently with the development of Gödel and so *SAGE* has been both influenced by and has influenced upon the Gödel language, most notably in the implementation of the ground representation. What has been produced is an implementation of the representation of substitutions and the operations upon them, for example unification, composition, application and resolution, which is designed in such a way that it should be both efficient and capable of being specialised, by a program specialiser such as *SAGE*, in order to produce significantly more efficient residual code. This implementation of the representation of substitutions and the operations upon them was originally developed as a part of the implementation of *SAGE* and has since been incorporated into the implementation of Gödel's ground representation so that all Gödel meta-programs might take advantage of the efficiences of this implementation. Details of this implementation are presented in Appendix A.

1.3 Partial Evaluation

The program specialisation technique that we use is partial evaluation[1], a specialisation technique that has been shown to have great potential, particularly in functional and in logic programming [4, 9, 19, 37, 60, 64, 69, 72]. It was first explicitly introduced into computer science by Futamura [20] and into logic programming by Komorowski [39]. A recent overview and bibliography for partial evaluation are given by [63] and [34] respectively.

The main specialisation techniques employed by most logic programming partial evaluators are *folding* and *unfolding* [37, 60, 63, 69]. Partial evaluation based upon finite unfolding was put on a firm theoretical basis in [47]. In this section we shall repeat from [47] the definitions and the main theorem that we need to prove the soundness of the partial evaluation techniques utilised by *SAGE*.

A concept that will be needed in the definition of partial evaluation is that of a resultant, which we now define.

Definition A *resultant* is a first order formula of the form $Q_1 \leftarrow Q_2$, where Q_i is either absent or a conjunction of literals ($i = 1, 2$). Any variables in Q_1 or Q_2 are assumed to be universally quantified at the front of the resultant.

Definition Let P be a normal program, G a normal goal $\leftarrow Q$, and $G_0 = G, G_1, \ldots, G_n$ an SLDNF-derivation of $P \cup \{G\}$, where the sequence of substitutions is $\theta_1, \ldots, \theta_n$ and G_n is $\leftarrow Q_n$. Let θ be the restriction of $\theta_1 \ldots \theta_n$ to the variables in G. Then we say the derivation has *length* n with *computed answer* θ and *resultant* $Q\theta \leftarrow Q_n$. (If n=0, the *resultant* is $Q \leftarrow Q$.)

Next we give the definition of a partial evaluation of a normal program wrt a set of atoms.

Definition Let P be a normal program, A an atom, and T an SLDNF-tree for $P \cup \{\leftarrow A\}$. Let $G_1, \ldots, G_r$ be (non-root) goals in T chosen so that each non-failing branch of T contains exactly one of them. Let R_i ($i = 1, \ldots, r$) be the resultant of the derivation from $\leftarrow A$ down to G_i given by the branch leading to G_i. Then the set of clauses $R_1, \ldots, R_r$ is called a *partial evaluation of* A *in* P.

If $\mathbf{A} = \{A_1, \ldots, A_s\}$ is a finite set of atoms, then a *partial evaluation of* $\mathbf{A}$ *in* P is the union of partial evaluations of $A_1, \ldots, A_s$ in P.

A *partial evaluation of* P *wrt* $\mathbf{A}$ is a normal program obtained from P by replacing the set of clauses in P whose head contains one of the predicate symbols appearing in $\mathbf{A}$ (called the *partially evaluated predicates*) by a partial evaluation of $\mathbf{A}$ in P.

A *partial evaluation of* P *wrt* $\mathbf{A}$ *using SLD-trees* is a partial evaluation of P wrt $\mathbf{A}$ in which all

[1]Also referred to as partial deduction.

the SLDNF-trees used for the partial evaluation are actually SLD-trees (i.e., no negation as failure steps are allowed during the partial evaluation process).

Next we provide definitions for the preconditions that must be met for the partial evaluation theorem.

Definition Let $\mathbf{A}$ be a finite set of atoms. We say $\mathbf{A}$ is *independent* if no pair of atoms in $\mathbf{A}$ have a common instance.

Definition Let S be a set of first order formulas and $\mathbf{A}$ a finite set of atoms. We say S is $\mathbf{A}$-*closed* if each atom in S containing a predicate symbol occurring in an atom in $\mathbf{A}$ is an instance of an atom in $\mathbf{A}$.

Definition Let P be a normal program and G a normal goal. We say G *depends* upon a predicate p in P if there is a path from a predicate in G to p in the dependency graph for P.

Definition Let P be a normal program, G a normal goal, $\mathbf{A}$ a finite set of atoms, P' a partial evaluation of P wrt $\mathbf{A}$, and P^* the subprogram of P' consisting of the definitions of predicates in P' upon which G depends. We say $P' \cup \{G\}$ is $\mathbf{A}$-*covered* if $P^* \cup \{G\}$ is $\mathbf{A}$-closed.

Finally we present the partial evaluation theorem relevant to the *SAGE* partial evaluator. This theorem originally appeared as Theorem 4.3 of [47].

Theorem 1.3.1 *Let P be a normal program, G a normal goal, $\mathbf{A}$ a finite, independent set of atoms, and P' a partial evaluation of P wrt $\mathbf{A}$ such that $P' \cup \{G\}$ is $\mathbf{A}$-covered. Then the following hold.*
(i) $P' \cup \{G\}$ has an SLDNF-refutation with computed answer θ iff $P \cup \{G\}$ does.
(ii) $P' \cup \{G\}$ has a finitely failed SLDNF-tree iff $P \cup \{G\}$ does.

1.4 Self-Application

The logic programming community's interest in partial evaluation stems primarily from the first Futamura projection [20], which illustrates how partial evaluation may be used to compile programs by the specialisation of meta-interpreters. The second Futamura projection shows that if the partial evaluator is self-applicable (able to specialise itself) a compiler may be produced [18, 33, 35, 62, 67]. This is taken one stage further in the third Futamura projection, where a *compiler-generator* is produced by specialising the partial evaluator with respect to itself. We argue that in order for the second and third Futamura projections to be achieved what is needed is a partial evaluator that is capable of producing efficient code upon the specialisation of general meta-programs.

1.4.1 Specialising Meta-Programs: The Three Futamura Projections

Perhaps the best-known example of the specialisation of a meta-program is that of the first Futamura projection, where an interpreter is specialised with respect to a particular object program in order to produce a 'compiled' version of that program. Here, as the specialised interpreter is a program that emulates the object program, it can be considered to be a version of the object program compiled into the language in which the interpreter is written.

Definition The First Futamura Projection.

$PE(I, P) = I_P$

That is to say, the partial evaluation of interpreter I with respect to the program P produces I_P. This is a specialised form of the interpreter which is capable of interpreting P with increased efficiency.

It could be the case that, for example, the interpreter simulates the execution of queries with respect to some program utilising a computation rule that differs from that of the underlying system. The extra expense that the ground representation imposes upon the execution of these queries can be largely removed by partial evaluation and we will be left with a version of the interpreter that executes the object program, utilising the control rule of the interpreter, in a time comparable to that of executing the original object program directly.

Suppose we wished to specialise some interpreter as above, with respect to a range of object programs, so as to produce compiled versions of a number of programs. In specialising the interpreter we would be removing the overheads of the ground representation in that meta-program, but the actual process of such a specialisation, performed by a partial evaluator, would itself suffer from these overheads. Here we see that we are continually performing a partial evaluation of the same meta-program (the interpreter) with respect to different queries (the different object programs). The obvious next step is to specialise the process of partially evaluating the interpreter. This will produce a version of the partial evaluator that is, through the removal of the expense of the ground representation, capable of specialising the interpreter in a time comparable to that of executing (the evaluable parts of) the interpreter directly. This process is known as the second Futamura projection, and the specialised version of the partial evaluator so produced, can be considered to be a compiler as the results it computes are compiled code in the sense of the first Futamura projection.

Definition The Second Futamura Projection.

$PE(PE, I) = PE_I$
$PE_I(P) = I_P$

Here the specialisation of the partial evaluator PE with respect to interpreter I produces the compiler PE_I.

To achieve the second Futamura projection we need the partial evaluator to be capable of specialising itself. The partial evaluator therefore needs to be able to specialise programs in the language in which it is written. We refer to this ability as self-applicability. However, it is not sufficient for a partial evaluator merely to be able to specialise itself, as we must also be certain that the partial evaluator will produce *efficient* results upon self-application. By efficient results we mean to say that the residual program produced, by applying the partial evaluator to itself, must have a significantly faster execution time compared to the original partial evaluator for equivalent queries. If, after self-application, the residual program is not significantly improved over the original program then there can have been little point in performing the partial evaluation. We refer to a partial evaluator that is capable of producing efficient results upon self-application, as an *effective* self-applicable partial evaluator.

Ultimately our aim is to specialise the partial evaluator with respect to itself. This is known as the third Futamura projection and the program produced is a compiler-generator.

Definition The Third Futamura Projection.

$PE(PE, PE) = PE_{PE}$
$PE_{PE}(I) = PE_I$

Here, the specialised partial evaluator generated by the third Futamura projection is used to *generate* a compiler, by specialising the partial evaluator with respect to an interpreter (i.e. performing the second Futamura projection).

1.4.2 What *is* a Compiler-Generator?

A compiler-generator is a meta-program which is, by the third Futamura projection, automatically generated by partially evaluating a partial evaluator with respect to itself. A compiler generator may be used to produce compilers by the specialisation of meta-interpreters. These compilers may be used to specialise the interpreters they embody, with respect to object programs, to produce compiled programs.

The three Futamura projections illustrate primarily the specialisation of meta-interpreters. However, this concept can easily be extended to encompass all meta-programs, as all meta-programs may be described as being interpreters in one form or another. With this generalisation we see that the first Futamura projection describes the specialisation of a general meta-program with respect to the representation of a program. In the second Futamura projection we specialise a specific meta-program, the partial evaluator itself, with respect to some, general, meta-program. The second Futamura projection is thus simply *one instance* of the first Futamura projection. Similarly the third Futamura projection, where we specialise the partial evaluator with respect to itself, is simply one instance of the second, and thus of the first, Futamura projection.

To reiterate:

- Specialising meta-programs is the first Futamura projection
- *One application* of this is the second Futamura projection
- *One application* of the second Futamura projection is the third Futamura projection

The compiler-generator is simply a tool for achieving our major aim of *specialising meta-programs* in a more efficient manner.

With this argument we see that the crucial step is to satisfactorily achieve the first Futamura projection. That is, to produce a partial evaluator that is capable of *effectively* specialising *any* meta-program in the language in which it is written. Once this goal is achieved then the second and third Futamura projections, and thus the generation of a compiler-generator, are natural consequences.

Although the third Futamura projection is a natural consequence of effectively achieving the first Futamura projection this in no way diminishes the importance of a compiler-generator as a tool for meta-programming. Suppose that, for example, a Gödel programmer develops a meta-program, *MP* say, which he/she wishes to use for a range of applications (that is, object programs). The meta-program *MP* suffers considerably from the expense of Gödel's ground representation and so the programmer would wish to specialise *MP* with respect to the various object programs in order to remove this inefficiency.

The programmer could perform this specialisation of *MP* by a call to *SAGE(MP,P)*, for each object program *P*. However, the same results could be achieved in a greatly reduced time by utilising the compiler $SAGE_{MP}$, produced by specialising *SAGE* to *MP*, to perform the same task. Thus the Gödel programmer, having developed *MP*, would wish firstly to produce the compiler $SAGE_{MP}$ so that he/she could set about specialising *MP* to its various applications.

$SAGE_{MP}$ could be produced by a call to *SAGE(SAGE,MP)*. However, when we consider the development of an entire family of meta-programs, then it would be as easy, and much faster, to use the compiler-generator $SAGE_{SAGE}$ to automatically produce a compiler from any given meta-program. Thus we see that the compiler-generator, while not ultimately producing any results that we could not achieve by considering the first Futamura projection only, is an immensely useful aid in speeding up the development of efficient meta-programs.

To conclude, the automatic generation of a compiler-generation by the third Futamura projection is a major achievement for two reasons. Firstly, a compiler-generator is a tremendously powerful and useful tool, allowing as it does far greater efficiency in the automatic application of the first and second Futamura projections. Secondly, the automatic generation, by the third Futamura projection, of a compiler-generator serves as perhaps the most significant test case in

proving that a partial evaluator successfully achieves the first Futamura projection, the specialisation of general meta-programs.

Chapter 2

Specialising Gödel Programs

In this chapter we present the techniques employed by *SAGE* for specialising the various facilities of Gödel, apart from the meta-programming facilities which we leave to the next chapter. Although *SAGE* is a partial evaluator for Gödel programs, certain of the facilities Gödel provides may be found in other languages also. The techniques described below are therefore applicable to a wider range of languages than just Gödel.

2.1 Connectives, Quantifiers and Conditionals

Gödel allows for the use of arbitrary formulas in the bodies of statements and goals. In Gödel, conjunction is denoted by `&`, disjunction by `\/`, negation by `~`, (left) implication by `<-`, (right) implication by `->`, and equivalence by `<->`. The universal quantifier is denoted by `ALL` and the existential quantifier is denoted by `SOME`. Each quantifier has two arguments, the first being a list of the quantified variables and the second the scope of the quantifier.

2.1.1 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`.

Naturally, the unrestricted unfolding of disjunctions in the above manner can potentially lead to code explosion and redundant computation in the residual code. In Section 3.4 we discuss *SAGE*'s approach to these problems.

Input:

a program P

a negative formula `~`F

Output:

F^*, the specialisation of `~`F

Initialisation

$Vars := \{\texttt{v}_0, \ldots, \texttt{v}_\texttt{n}\}$, the set of free variables in F

$R :=$ partial evaluation of F wrt P

If $R = \emptyset$

Then % F has finitely failed

$F^* =$ `True`

Else

$R = \{R_0, \ldots, R_m\}$

$i = 0, \ldots, m, B_i := \{\texttt{v} = \texttt{t} : \texttt{v}$ is bound to `t` in computing $R_i\}$

$i = 0, \ldots, m$, $F'_i := B'_i$ `&` R_i, where $B'_i =$ the conjunction of the set of atoms B_i

If for some i, $F'_i =$ `True`

Then % F has safely succeeded

$F^* =$ `False`

Else % residual code for `~`F

$F^* =$ `~`F'_0 `&...&` `~`F'_m

Figure 2.1: Constructive Unfolding of Negation

2.1.2 Negation

Figure 2.1 illustrates the algorithm employed by *SAGE* to unfold negated formulas. This algorithm is based upon the concept of constructive negation [13]. 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.

2.1.3 Quantifiers

When *SAGE* renames variables in the representations of Gödel formulas, by using either `RenameFormulas`, `StandardiseFormulas` or `ResolveAll`, existentially quantified variables are ren-

amed 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,y] P(x,y,z)`, are transformed to the form `~(SOME [x,y] ~P(x,y,z))`.

2.1.4 Conditionals

When specialising conditionals in Gödel we need only to consider the `IF-THEN-ELSE` construct, as formulas of the form `IF` *Condition* `THEN` *Formula* may be transformed to `IF` *Condition* `THEN` *Formula* `ELSE True`.

The first form of the `IF-THEN-ELSE` construct is

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

and Figure 2.2 illustrates the algorithm employed to specialise a formula of this form. 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 *Conditional*, *Formula1* and *Formula2* are replaced by their specialised versions.

The second form of the `IF-THEN-ELSE` construct is

`IF SOME` $[x_1, \ldots, x_n]$ *Condition* `THEN` *Formula1* `ELSE` *Formula2*

and a specialisation of this second form of the `IF-THEN-ELSE` construct is performed in much the same manner as for the first form, the only difference being the treatment of variables bound in the specialisation of *Condition*. If the specialisation indicates that *Condition* has failed finitely, then the conditional is replaced by *Formula2* as before. If this is not the case then we construct the specialised conditional C^* as before, but we consider only the free variables of the formula $\exists V C^*$, where V is the list of variables $[x_1, \ldots, x_n]$, when determining whether any free variables have

Input:

a program P

a conditional formula $F =$ `IF` C `THEN` T `ELSE` E

Output:

F^*, the specialisation of F

Initialisation

Vars $:= \{\mathtt{v_0}, \ldots, \mathtt{v_n}\}$, the set of free variables in C

$R^C :=$ partial evaluation of C wrt P

If $R^C = \emptyset$

Then % C has finitely failed

$F^* =$ the partial evaluation of E

Else

$R^C = \{R_0, \ldots, R_m\}$

$i = 0, \ldots, m, B_i := \{\mathtt{v} = \mathtt{t} : \mathtt{v}$ is bound to $\mathtt{t}$ in computing $R_i\}$

$i = 0, \ldots, m$, $C'_i := B'_i$ `&` R_i, where $B'_i =$ the conjunction of the set of atoms B_i

$C^* = C'_0$ `\/`$\ldots$`\/` C'_m

If for some i, $C'_i =$ `True`

Then % C has safely succeeded

$F^* =$ the partial evaluation of C^* `&` T

Else % residual code for F

$R^T :=$ partial evaluation of T wrt P

$R^E :=$ partial evaluation of E wrt P

construct T^* and E^* from R^T and R^E respectively

(T^* and E^* are constructed as C^* is constructed from R^C above)

$F^* =$ `IF` C^* `THEN` T^* `ELSE` E^*

Figure 2.2: Unfolding `IF-THEN-ELSE`

been bound in C^*. If C^* cannot be safely determined to have either failed or succeeded then a new conditional is constructed as before, although using `IfSomeThenElse` rather than `IfThenElse`.

2.2 Type Declarations

Gödel provides a type system based on a parametric many-sorted logic. In this type system we may declare polymorphic symbols, such as the declaration for the `Append` predicate:

```
PREDICATE  Append : List(a) * List(a) * List(a).
```

Here, `a` is a parameter which can be instantiated to any type of the language of the logic.

To ensure that Gödel goals may be run with no run-time type checking Gödel imposes two conditions on the type declarations for Gödel programs. The first of these is the *head condition* which is as follows:

- A statement satisfies the *head condition* if the tuple of types of the arguments of the head in the statement is a variant of the type declared for the predicate in the head.

The head condition is the one aspect of Gödel's type system that *SAGE* particularly needs to concern itself with. In partially evaluating an atom whose predicate symbol has a polymorphic declaration it is perfectly possible for arguments in the heads of residual statements to become sufficiently instantiated to require a specialisation of the predicate declaration.

For example, partially evaluating the atom `Append([1],[],z)` leads to the new statement `Append([1],[],[1])`. With this statement, the above declaration for `Append` no longer satisfies the head condition and would need to be replaced by the new declaration:

```
PREDICATE  Append : List(Integer) * List(Integer) * List(Integer).
```

This specialisation of polymorphic predicate declarations is performed at the point at which *SAGE* uses the specialised code and the original program to construct the specialised program. This is discussed in more detail in Section 4.4.

2.3 Control

There are two aspects to control in Gödel that are considered by *SAGE*. The first of these is the computation rule. The second is Gödel's pruning operator which determines those subtrees that will be pruned from the search tree. We examine the specialisation of each of these in turn.

2.3.1 Computation Rule

Gödel supports a flexible computation rule by means of DELAY declarations, which may appear either in user-defined or system modules. In general we would expect that the DELAY declarations for a program should be sufficient to ensure that the computation of some goal will not proceed at any point in the execution of some query with respect to this program, unless that goal is sufficiently instantiated to ensure that the computation will behave correctly and will terminate.

In partial evaluation we generally assume that we are specialising a program with respect to a particular class of queries. That class is usually represented by some partially instantiated query for this program. Our aim is to perform as much computation as possible, based upon the partial knowledge that we have of the queries which the specialised program is intended to answer. Consequently, when performing a partial evaluation we would generally expect to be specialising goals which are only partially instantiated and will therefore be unlikely to satisfy any DELAY declarations. To insist that the partial evaluation of some partially instantiated query with respect to some program respected the DELAY declarations for that program will generally prove unacceptably prohibitive to the partial evaluation process. There seems therefore to be little value in examining the DELAY declarations during partial evaluation.

Although *SAGE* pays no attention to DELAY declarations when computing partial evaluations, when constructing the specialised program it is possible that certain declarations may have to be updated to match specialised predicate declarations. This issue is discussed in more detail in Chapter 4.4.

2.3.2 Pruning

Gödel provides a pruning operator, *commit*, which we introduced in Section 1.2. In [29] Hill *et al* provide a more detailed description of the commit and prove that, while naturally affecting completeness, it is a sound operator for logic programming.

It is also shown in [29] that, provided two conditions are met, the computational equivalence of a program and its partial evaluation wrt a given goal (that is, [47, 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.

In this section we shall show 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. We present the technique implemented by *SAGE* to avoid the need for the regularity condition and prove the correctness of this technique for partial evaluation. To enable this proof we first repeat

from [29] the formal definitions of freeness and regularity and the extended partial evaluation theorem.

Definitions for the Commit

The definitions of a c-program, c-program clause and c-goal naturally extend the usual definitions for programs, program clauses and goals by allowing commits within formulas.

Definition Let P be a normal c-program and G a normal c-goal. A *cSLDNF-tree* for $P \cup \{G\}$ is a tree satisfying the following conditions.

1. Each node of the tree is a (possibly empty) normal c-goal.

2. The root node is G.

3. Suppose $\leftarrow Q$ is a non-leaf node. Then a literal, say L, is selected in Q and either

   (a) L is an atom,

   $$\begin{array}{lcl} A_1 & \leftarrow & M_1 \\ \vdots & & \\ A_r & \leftarrow & M_r \end{array}$$

   is the set of (standardised apart) c-program clauses such that L and A_j unify with mgu θ_j $(1 \leq j \leq r)$, $\{M'_1, \ldots, M'_r\}$ is the set obtained from $\{M_1, \ldots, M_r\}$ by replacing each commit label in $\{M_1, \ldots, M_r\}$ by a label not in $\leftarrow Q$ such that if l and m are commit labels replaced by l' and m', respectively, then $l = m$ iff $l' = m'$, and

   $$\begin{array}{l} \leftarrow Q_1\theta_1 \\ \vdots \\ \leftarrow Q_r\theta_r \end{array}$$

   are the r children of the node $\leftarrow Q$, where Q_j is obtained from Q by replacing the selected atom in Q by M'_j and removing any empty commits $(1 \leq j \leq r)$, or

   (b) L is a ground negative literal $\neg A$ and there is a finitely failed SLDNF-tree for$P^- \cup \{\leftarrow A\}$, where P^- is the normal program underlying P. In this case, there is only one child $\leftarrow Q_1$, where Q_1 is obtained from Q by removing the selected literal $\neg A$ and removing any empty commits. The associated substitution is the identity substitution.

4. Suppose $\leftarrow Q$ is a leaf node. Then either

   (a) Q is empty, in which case the node $\leftarrow Q$ is said to be *empty*, or

(b) Q is non-empty and there is a selected literal L, and either

i. L is an atom and there is no c-program clause (variant) in P whose head unifies with L, or

ii. L is a ground negative literal $\neg A$ and there is an SLDNF-refutation of $P^- \cup \{\leftarrow A\}$, where P^- is the normal program underlying P,

in which case the node $\leftarrow Q$ is said to be *failed*, or

(c) Q is non-empty and no literal is selected, in which case the node $\leftarrow Q$ is said to be *incomplete*.

A *complete* cSLDNF-tree is a cSLDNF-tree with no incomplete leaf nodes. If the cSLDNF-tree for $P \cup \{G\}$ consists of just the incomplete leaf node G, we say it is *trivial*. Otherwise, we say it is *non-trivial*. We say a branch in a cSLDNF-tree is a *success branch* if it ends in an empty leaf node and a *failed branch* if it ends in a failed leaf node. A subtree of a cSLDNF-tree is said to be *failed* if every leaf node of the subtree is failed.

Definition Let T be a cSLDNF-tree, G_0 a non-leaf node in T, and G_1 a child of G_0. Then G_1 is an *l-child* of G_0 if **either**

1. G_0 contains a commit labelled l and the selected literal in G_0 is in the scope of this commit, **or**

2. G_1 is derived from G_0 using as input clause a c-program clause containing a commit labelled l (after standardisation apart of the commit labels).

We say that G_1 is an l-child *of the first kind* (resp., *of the second kind*) if G_0 satisfies condition 1 (resp., 2) above.

Definition Let S be a subtree of a cSLDNF-tree. We say that the tree S' is obtained from S by a *pruning step* in S at G_0 if the following conditions are satisfied.

1. S has a node G_0 with distinct l-children G_1 and G_2, and there is an l-free node G_2' in S which is either equal to or below G_2.

2. S' is obtained from S by removing the subtree of S rooted at G_1.

We say that G_1 is the *cut node*, the pair (G_2, G_2') is an *explanation* for the pruning step and that G_2' has *pruned* the subtree rooted at G_1.

Definition A subtree S of a cSLDNF-tree is said to be *prunable* if there is a pruning step in S which can be applied at a node in S. Otherwise, S is said to be *unprunable*.

Definition Let T be a cSLDNF-tree and S a pruned subtree of T. We say that S is *regular* if, for each node in S with more than one child in S, the selected literal is in the scope of all the commits at that node.

Note that by this definition, if S is regular, then any non-root node in S with an l-commit is either the only child or an l-child of its parent node.

Definition Let T be a cSLDNF-tree and S a pruned subtree of T. We say that R is a *c-resultant for* S if R is the c-resultant of a branch in S which ends at a non-failed leaf node. If T is an SLDNF-tree and R is the resultant of a branch in T which ends at a non-failed leaf node, then we call R a *resultant for* T.

Definition Let P be a normal c-program, A an atom, and T a finite, non-trivial cSLDNF-tree for $P \cup \{\leftarrow A\}$ such that, if the selected literal at a node $\leftarrow Q$ is an atom, then the commit labels in the matching clauses are standardised apart not only wrt the labels in $\leftarrow Q$ but also all normal c-goals in T which are not descendants of $\leftarrow Q$. Let S be a pruned subtree of T and $\{R_1, \ldots, R_r\}$ the set of all the c-resultants for S. Then the set $\{R_1, \ldots, R_r\}$ of normal c-program clauses is called a *partial evaluation of* A *in* P.

The partial evaluation is said to be *regular* if S is regular. The partial evaluation is said to be *prunable* (resp., *unprunable*) if the subtree of S obtained by removing all failed branches is prunable (resp., unprunable). The partial evaluation is said to be *free* if $S = T$.

The following theorem extends the partial evaluation theorem, Theorem 1.3.1, to c-programs. This theorem appeared as Theorem 3.2 in [29].

Theorem 2.3.1 *Let P be a normal c-program, G a normal c-goal, $\mathbf{A}$ a finite, independent set of atoms, and P' a partial evaluation of P wrt $\mathbf{A}$ such that $P' \cup \{G\}$ is $\mathbf{A}$-closed. If the partial evaluation is free and regular, and S' is a pruned subtree of a cSLDNF-tree T' for $P' \cup \{G\}$, then there is a pruned subtree S of a cSLDNF-tree T for $P \cup \{G\}$ which has the same set of c-computed answers as S'.*

The above theorem states that the partial evaluation should be $\mathbf{A}$-closed. In Theorem 1.3.1 this condition was replaced with the less restrictive condition that the partial evaluation be $\mathbf{A}$-covered. The extension of the above theorem is relatively straightforward. Note that this theorem states only that for any potential pruning step in a partially evaluated program an equivalent pruning step may be applied to the original program. No assumptions are made that the strategy employed to construct the partially evaluated program will guarantee that equivalent pruning steps are actually *applied* in both the original and partially evaluated programs, only that they exist.

Freeness of *SAGE* Partial Evaluations

The freeness condition says that we are not permitted to perform pruning during a partial evaluation. The following example shows that without this condition, we cannot guarantee the soundness of the partially evaluated program with respect to the original program.

Example Let P be the normal c-program

```
P(A) <- {Q}_1.
P(B) <- {Q}_1.
Q.
```

G the normal c-goal `<- ~P(A)`, and $\mathbf{A} = \{$`P(x)`$\}$. Then the normal c-program P'

```
P(B).
```

can be obtained by a non-free partial evaluation of P wrt $\mathbf{A}$. However, the identity substitution is a c-computed answer for $P' \cup \{G\}$, but not for $P \cup \{G\}$.

It should be noted that there is one case in which we might permit a partial evaluation to perform pruning. If we can guarantee that performing a pruning step will only prune failing computations then we might allow that pruning step to proceed.

For example, suppose that during a partial evaluation the atom `AnalyseTerm(`$\mathcal{T}$`)` were unfolded, where the definition of `AnalyseTerm` is:

```
AnalyseTerm(term) <-
  Variable(term) |
  AnalyseVariable(term).
AnalyseTerm(term) <-
  ConstantTerm(term, name) |
  AnalyseConstant(term).
AnalyseTerm(term) <-
  FunctionTerm(term, name, args) |
  AnalyseFunctionTerm(term).
```

If the term $\mathcal{T}$ were instantiated to some ground term, then we may guarantee that at most one of the three statements above will succeed. Therefore, once we have determined which of the three atoms `Variable(`$\mathcal{T}$`)`, `ConstantTerm(`$\mathcal{T}$`,name)` and `FunctionTerm(`$\mathcal{T}$`,name,args)` succeeds we may prune the other two branches in the cSLDNF-tree. Naturally we would only be able to perform this pruning step in the case where the arguments of the committed formula were sufficiently instantiated for us to determine which of the branches is the correct one. If the term $\mathcal{T}$ were a variable for example, we would not be able to correctly perform a pruning step.

There is one version of *SAGE* which allows the user to state prior to performing a partial evaluation whether freeness should be enforced or not. Note that allowing pruning (by not enforcing

freeness) is purely an efficiency measure and is not permitted in general, as the soundness of the results may be affected. *SAGE* will enforce freeness by default in order to ensure soundness in the general case and it is expected that the freeness condition will be enforced for the majority of partial evaluations.

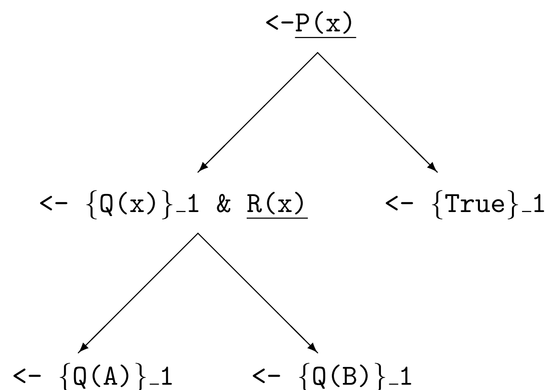
Regularity of *SAGE* Partial Evaluations

The regularity condition says that, in a partial evaluation, *SAGE* should only select literals that are in the scope of all commits occurring in the current resultant. The following example illustrates the need for this condition.

Example Let P be the normal c-program

```
P(x) <- {Q(x)}_1 & R(x).
P(C) <- {True}_1.
R(A).
R(B).
Q(x).
```

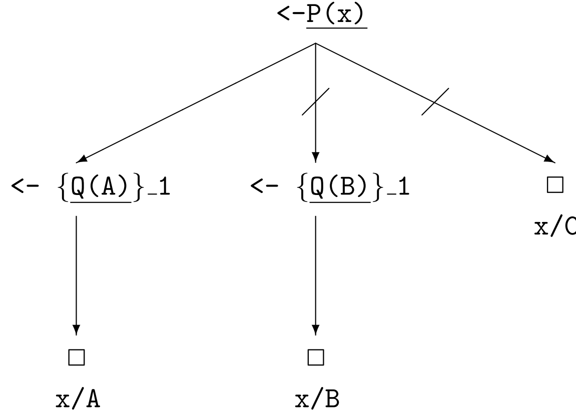
G the normal c-goal `<- P(x)` and $\mathbf{A} = \{$`P(x)`$\}$. Using the cSLDNF-tree T_0



we obtain a partial evaluation P' of P wrt $\mathbf{A}$ such that $P' \cup \{G\}$ is $\mathbf{A}$-closed and the definition of `P` in P' is

```
P(A) <- {Q(A)}_1.
P(B) <- {Q(B)}_1.
P(C) <- {True}_1.
```

The partial evaluation is free, but not regular. There is a cSLDNF-tree T_1



for $P' \cup \{G\}$ that can be pruned to have just the c-computed answer {x/A}. There is, however, no tree for $P \cup \{G\}$ which can be pruned to have just the answer {x/A}. We denote the subtree removed by a pruning step by placing a 'cut-line' across the branch leading to the cut node.

Regularity is an extremely strong condition which places firm restrictions on the amount of unfolding permitted for most committed programs. An examination of almost any program containing commits (for example the `Quicksort` program defined by module `P1` in Section 1.2 or any of the meta-programs presented in Section 5.1) will demonstrate that to restrict unfolding in a partial evaluation to only those literals which appear in the scope of all commits in a resultant will generally be unacceptably restrictive. In addition, restricting the selection strategy of a partial evaluator to satisfy the regularity condition is potentially very expensive in computational terms. In order to decide for which literals in a resultant the regularity condition is satisfied requires explicit knowledge of the scopes of all commits in each resultant. Therefore we have found it necessary to develop a technique for unfolding formulas containing commits which permits us to disregard regularity while ensuring the correctness of the partial evaluations.

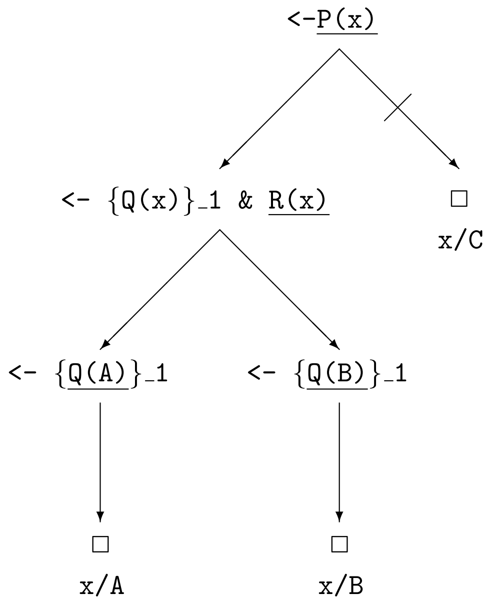
To justify our solution to this problem we must first understand exactly how it has occurred. In the tree T_0 we have performed two unfolding steps. We first unfolded the root node `<- P(x)` to produce two new nodes, each of which is a 1-child (of the second kind) of the root node. We next unfolded the node, `<- {Q(x)}_1 & R(x)` to produce two children. However, as the selected atom in that node, `R(x)`, is outside the scope of the commit, the two children of not 1-children of it. We refer to the first unfolding step in T_0 as a *regular* unfolding step and to the second as an *irregular* unfolding step.

Definition Let T be a cSLDNF tree and N a non-leaf node in T. If the children of N are l-children, for some commit label l, we say that we have performed a *regular* unfolding step at N in T. Otherwise we say that we have performed an *irregular* unfolding step at N in T. When the

cSLDNF-tree may be inferred from the context we shall say simply that an unfolding step is regular (respectively, irregular).

From the first condition of the definition of a pruning step we see that a cSLDNF-tree S' may be obtained from a cSLDNF-tree S by a pruning step at some node G iff we have performed a regular unfolding step at G in S. This point is crucial to understanding where the problem in the above example arises.

Let T_2 be the cSLDNF-tree for $P \cup \{G\}$ which is a natural extension for the above tree T_0

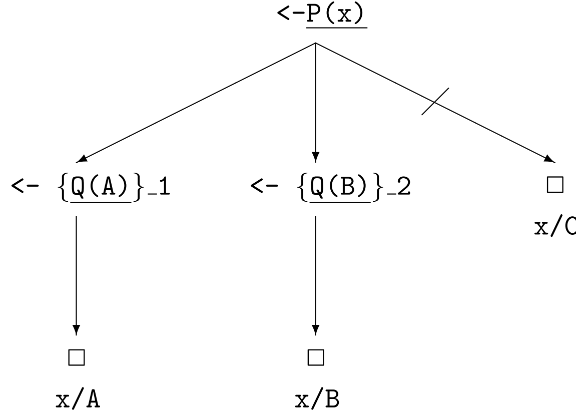


Ignoring pruning for the moment we see that the leaf-nodes of T_2 are equivalent to those of T_1, as we would expect. The crucial difference between these two trees is that while in T_1 we may perform two pruning steps to leave just the c-computed answer {`x/A`}, we may not do this in T_2. In T_2 we may only prune at the root node where we have performed a regular unfolding step. At the node `<- {Q(x)}_1 & R(x)` in T_0 we performed an irregular unfolding step which gave rise to the two distinct leaf nodes. The problem with the tree T_1 arises because the information that the two nodes `<- {Q(A)}_1` and `<- {Q(B)}_1` were produced by an irregular unfolding step, and thus may not prune each other, has been lost.

We may solve this problem by introducing a new commit label to rename apart the commits in the nodes `<- {Q(A)}_1` and `<- {Q(B)}_1` to prevent them from pruning each other. From this we can see that a specialised definition for the predicate `P` could be:

```
P(A) <- {Q(A)}_1.
P(B) <- {Q(B)}_2.
P(C) <- {{True}_1}_2.
```

Let P^* be the program P with the above definition for predicate `P` replacing that of P. Then the cSLDNF-tree T_3



for $P^* \cup \{G\}$ is correct wrt T_2, in the sense that for any pruning step in T_2 there is a pruning step in T_3 which prunes the equivalent nodes and vice versa.

SAGE 's Approach to the Relabelling of Commits

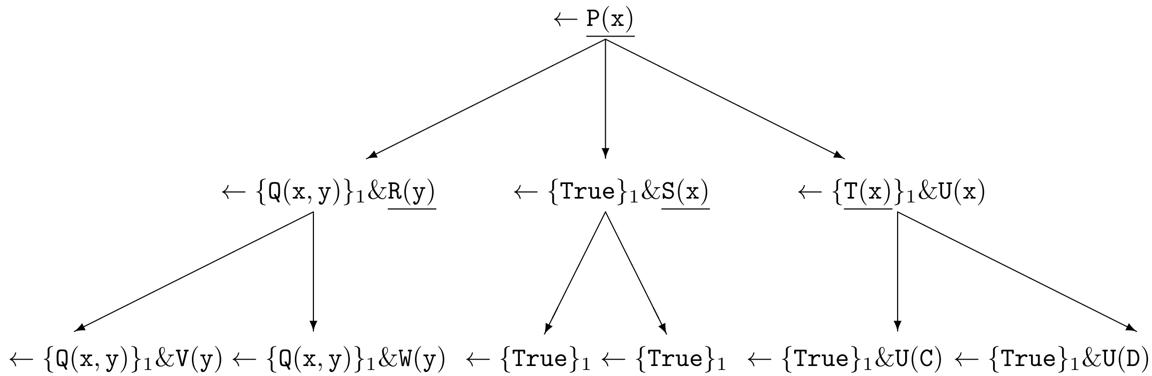
When unfolding a formula which contains a committed subformula, *SAGE* will perform the specialisation of the committed subformula as a self-contained sub-computation. Having specialised a committed formula, *SAGE* will not attempt to perform any more unfolding upon any residual literals within the scope of the specialised committed formula. This means that the unfolding of such a formula has three stages. Firstly a number of unfolding steps are performed outside the scope of the commit. Secondly the committed subformula is unfolded and finally a number of unfolding steps are performed outside the scope of the commit once again. For example, when specialising a formula of the form `A & {B & C} & D` *SAGE* will perform the specialisation of the formula `{B & C}` as a self-contained sub-computation. Assuming, for example, that *SAGE* used an exclusively left-to-right selection strategy, it would thus specialise the formulas `A`, followed by `{B & C}` and finally `D`.

A more general approach which allows unrestricted unfolding (either regular, irregular or any combination thereof) is possible and is a relatively straightforward extension of the techniques described below. However, the restricted approach described here is sufficient to adequately specialise most reasonable cases and so for the sake of simplicity we do not consider the most general case in this thesis.

We illustrate the implementation of the above relabelling of commits by *SAGE* with the following example. Given the program:

```
P(x) <- {Q(x,y)}_1 & R(y).
P(x) <- {True}_1 & S(x).
P(x) <- {T(x)}_1 & U(x).
R(x) <- V(x).
R(x) <- W(x).
S(A).
S(B).
T(C).
T(D).
```

the cSLDNF-tree used to compute the partial evaluation (ignoring relabelling of commit labels) of the atom `P(x)` with respect to this program, where it was not permitted to unfold atoms with the predicates `Q`, `U`, `V` or `W`, is:



leaving us the six residual statements:

```
1.      P(x) <- {Q(x,y)}_1 & V(y).
2.      P(x) <- {Q(x,y)}_1 & W(y).
3.      P(A) <- {True}_1.
4.      P(B) <- {True}_1.
5.      P(C) <- {True}_1 & U(C).
6.      P(D) <- {True}_1 & U(D).
```

Consider firstly just the statements `1`–`4`. Statements `1` and `2` were produced by unfolding the formula $\leftarrow \{\texttt{Q(x,y)}\}_1\&\texttt{R(y)}$ outside the scope of the commit and thus they must not be allowed to prune each other, although they will each prune the statements `3` and `4`. Statements `3` and `4` were similarly produced by unfolding $\leftarrow \{\texttt{True}\}_1\&\texttt{S(x)}$ outside the scope of the commit and thus they also will not prune each other, although they will both prune `1` and `2`.

To implement the above restrictions on pruning *SAGE* augments the label of each commit in a resultant by an extra integer value when that commit is first introduced. In the above example

the commit with label 1 is introduced in the nodes $\leftarrow \{\texttt{Q(x,y)}\}_1\&\texttt{R(y)}$, $\leftarrow \{\texttt{True}\}_1\&\texttt{S(x)}$ and $\leftarrow \{\texttt{T(x)}\}_1\&\texttt{U(x)}$. We therefore record the commit labels for these three nodes as 1_1, 1_2 and 1_3 respectively. For statements `1–4` these augmented labels are copied from the parent node to the relevant child. Thus the augmented labels for the statements `1–4` are 1_1, 1_1, 1_2 and 1_2 respectively.

We can see therefore that pruning will be performed correctly for the four statements `1–4` if we follow the rule that when committing to a statement with augmented commit label n_i, we prune precisely those other statements with augmented commit label n_j, such that $i \neq j$.

When we consider also the two statements `5` and `6` the situation becomes a little more complex and we must augment the commit labels further to deal with this correctly. Statements `5` and `6` were produced by unfolding *within* the scope of the commit and thus they will each prune all other statements, including each other. The following table indicates, for each new statement, which other statements should be pruned if we commit to that statement.

Statement	Prunes Statements
1	3, 4, 5, 6
2	3, 4, 5, 6
3	1, 2, 5, 6
4	1, 2, 5, 6
5	1, 2, 3, 4, 6
6	1, 2, 3, 4, 5

As we have unfolded the goal $\leftarrow \{\texttt{T(x)}\}_1\&\texttt{U(x)}$ within the scope of the commit we cannot simply copy this node's augmented commit label, 1_3, to the two children of this node. Instead we must add an entirely new augmented commit label at this point. We add a new integer pair $m : i$ to each child, where m is some new integer value and i is used to enumerate the children. Thus the new augmented commit labels for the statements `5` and `6` will be $1_3^{2:1}$ and $1_3^{2:2}$ respectively. Note that we have used here the integer 2 as the new integer value m, as 2 is an integer of greater value than the label of any other commits in the above cSLDNF-tree, the only other commit label being 1 in this case.

To ensure that the augmented commit labels all have the same format we may assume that the augmented labels initially introduced (1_1, 1_2 and 1_3) are syntactic sugar for the more complex augmented commit labels $1_1^{0:0}$, $1_2^{0:0}$ and $1_3^{0:0}$. The rule that we must apply when committing to a statement with these more complex augmented commit labels is that when committing to a statement with augmented commit label $n_i^{j:k}$, we prune precisely those other statements with augmented commit labels $n_l^{m:p}$ such that $(l \neq i) \vee (m = j \wedge p \neq k)$.

Following the above process for augmenting commit labels we see that the augmented commit labels for the above statements `1–6` will be $1_1^{0:0}$, $1_1^{0:0}$, $1_2^{0:0}$, $1_2^{0:0}$, $1_3^{2:1}$ and $1_3^{2:2}$ respectively. Following the above pruning rule we see that we have satisfied the necessary pruning in the above table.

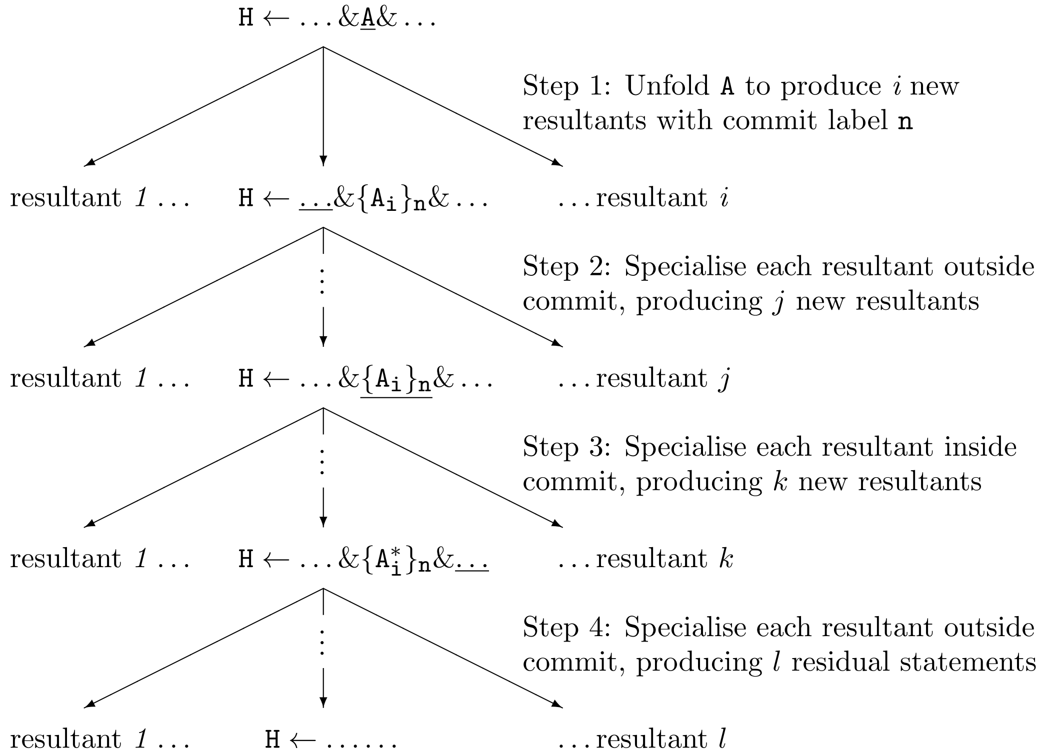


Figure 2.3: A Generalised Irregular cSLDNF-tree

Having performed a partial evaluation *SAGE* is able to analyse the augmented commit labels for the residual resultants and compute a matrix of new Gödel commit labels for each new statement which preserves the soundness of pruning with respect to the original program. This avoids the need to enforce regularity while performing the partial evaluation. For the above example *SAGE* will give the following new definition for the predicate `P`:

```
P(x) <- {{Q(x,y)}_1}_2 & V(y).
P(x) <- {{Q(x,y)}_3}_4 & W(y).
P(A) <- {{True}_1}_3.
P(B) <- {{True}_2}_4.
P(C) <- {{{{True}_1}_2}_3}_4 & U(C).
P(D) <- {{{{True}_1}_2}_3}_4 & U(D).
```

The tree in Figure 2.3 illustrates the generalisation of unfolding with our approach. Note that the only nodes in this tree which have `n`-children are the root node and all nodes which appear in the tree after step 2 but before step 3. All `n`-children of the root node are `n`-children of the second kind and all children of nodes which appear after step 2 but before step 3 are `n`-children of the first kind. We shall make use of this information when we prove the correctness of a pruning step

performed wrt the augmented commit labels.

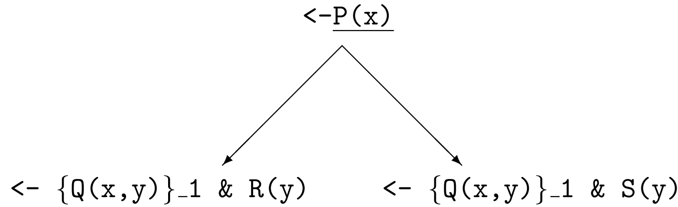
In general, a commit label will be introduced into a partial evaluation in i resultants. For each resultant containing a commit, *SAGE* will unfold outside the scope of this commit to produce j new resultants, for some j. The committed formula in each of these new resultants may be unfolded to produce k residual committed formulas, for some k. A resultant constructed by replacing the original committed formula by the specialised formula may then be unfolded outside the scope of the specialised committed formula to produce l resultants, for some l.

We next show that the augmented commit labels created for the resultants in the tree in Figure 2.3 capture the necessary information concerning the branches used in deriving those resultants. At the first step we unfold an atom A in a resultant, introducing i new resultants containing a committed formula with label $\mathtt{n}$. The commits in these resultants are labelled as $\mathtt{n}_1^{0:0}, \ldots, \mathtt{n}_i^{0:0}$ respectively.

For example, consider the (partial) program:

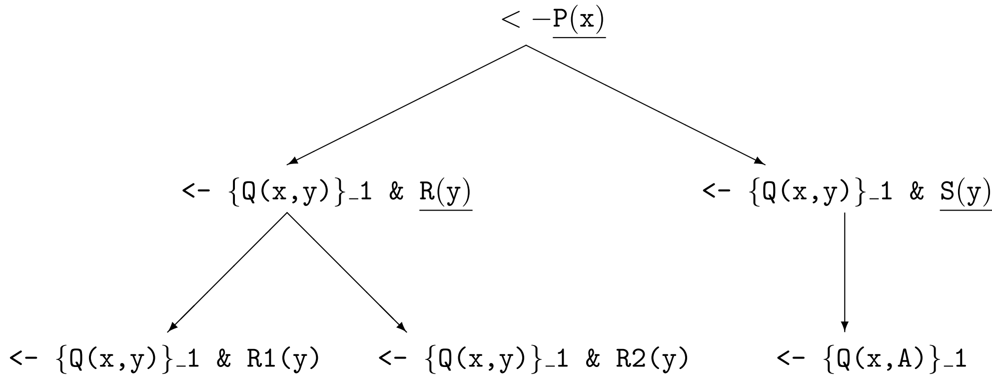
```
P(x) <- {Q(x,y)}_1 & R(y).
P(x) <- {Q(x,y)}_1 & S(x).
Q(x,y) <- Q1(x,y).
Q(x,y) <- Q2(x,y).
R(x) <- R1(x).
R(x) <- R2(x).
S(A).
```

and the following cSLDNF-tree T_1 for this program



we have (step 1) unfolded the atom `P(x)` to produce two 1-children. The augmented labels for these two distinct nodes will be $1_1^{0:0}$ and $1_2^{0:0}$ respectively. Using the notation that n_i is syntactic sugar for $\mathtt{n}_i^{0:0}$ we may see that the augmented pruning rule "n_i prunes n_j whenever $i \neq j$" is correct in this case.

At the second step each of these i new resultants are specialised outside the scope of the commit, producing $j_1, \ldots, j_i$ new resultants respectively. The augmented commit labels for each of these new resultants will be identical to that of the ancestor resultant produced by step 1. If we were to extend the above cSLDNF-tree by step 2 to give the tree T_2

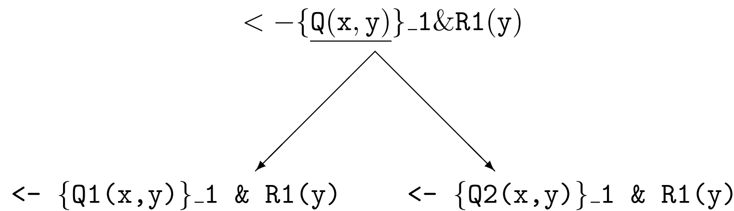


then the augmented labels (in shortened form) for the three leaf nodes of this tree, from left to right, will be 1_1, 1_1 and 1_2. Again we may see that this is correct with respect to our new pruning rule. As we have performed an irregular unfolding step, we do not want the branches which have a common ancestor in the leaf nodes of T_1 (that is, the nodes produced after step 1) to prune each other, but we do want branches which had a distinct ancestor after step 1 to prune each other.

To illustrate the third step we consider just one of the resultants produced from the previous step, with augmented commit label $\mathtt{n}_a^{0:0}$, say. In the third step we specialise the committed formula in this resultant to produce k new resultants. The augmented commit labels for these new resultants will be $\mathtt{n}_a^{m:1}, \ldots, \mathtt{n}_a^{m:k}$ respectively, where m is some new integer value.

After the second step there were j_a resultants with augmented label $\mathtt{n}_a^{0:0}$. For each of these resultants we ensure that a distinct value for m is used in the above augmenting of labels following step 3. For example, following step 3 two resultants would have the augmented commit labels $\mathtt{n}_a^{m_1:b}$ and $\mathtt{n}_a^{m_2:c}$ respectively, for some b and c. If the ancestors of these two resultants (produced after step 2) were distinct then the values of m_1 and m_2 would be different. If these two resultants were produced from the same ancestor by step 3, then the values of m_1 and m_2 would be equal.

For example, consider the leaf node of T_2, `<- {Q(x,y)}`$_1$ `& R1(y)`. We may unfold this node *within* the scope of the commit to produce the subtree



We have performed here a second regular unfolding step. The two leaf nodes of this tree should prune each other in addition to any branches which their parent node would have pruned. Assuming that the integer value 2 was greater than any other value appearing as a commit label in the current

computation, the augmented labels for the leaf nodes of the above subtree would be $n_1^{2:1}$ and $n_1^{2:2}$ respectively. We must add the new integer pairs 2 : 1 and 2 : 2 to the augmented labels for these nodes to distinguish them from any new augmented labels for nodes below the node `<- {Q(x,y)}`$_1$ `& R2(y)` in T_2 which may also be introduced by step 3.

At the fourth and final step each resultant produced by the third step is specialised outside the scope of the commit to produce, say, l new resultants. The augmented commit labels for each of these new resultants will be identical to that of the ancestor resultant which was produced by step 3. Again we may see that this is correct with respect to our new pruning rule, that two distinct resultants with identical augmented labels $n_a^{f:p}$ should not prune one another.

Correctness of *SAGE* 's Unfolding of Commits

Suppose that T is a cSLDNF-tree of the form of Figure 2.3 for a program P and that P' is a partial evaluation of P constructed using the leaf nodes of T. In order to prove the correctness of pruning in P' wrt P we must show that all of the information relevant to the application of a pruning step in T is captured in the augmented labels used to rename the commits in P'.

Consider, for the sake of simplicity, a cSLDNF-tree of the form of Figure 2.3 with a root node $\leftarrow A$, where A is some atom. This tree is a truncated version of the cSLDNF-tree T' used to compute answers for $P \cup \{\leftarrow A\}$. We may restore T to T' by adding the relevant subtrees of T' to the leaf nodes of T. We may refer to this as the *extension* of T. If we take the node $\leftarrow A$ and perform one unfolding step wrt P' we see that we have a tree of depth 2 whose leaf-nodes are equivalent to the leaf nodes of T. We may therefore extend this tree by the (renamed) subtrees used to extend T. In this way we can construct a tree T^* for P' which is equivalent to T'.

When we perform a pruning step in the tree T' we will have a cut node and an explanation for this step. The cut node and the first node in the explanation may both appear in the tree T and will therefore not appear in the tree T^*. Theorem 2.3.2 proves that the information concerning these nodes is captured by our augmented commit labels. Before we present this theorem we must formalise the notion of extension.

Definition Let T be a cSLDNF-tree with leaf nodes $N_1, \ldots, N_n$, $\mathbf{S}$ the set of subtrees S_i, for $i = 1, \ldots, n$, such that S_i is rooted at N_i and T' the tree formed by adding, for $i = 1, \ldots, n$, the subtree S_i to the leaf node N_i of T. We say that T' is the *extension* of T by $\mathbf{S}$.

Definition Let P be a c-program, A an atom, T a cSLDNF-tree for $P \cup \{A\}$, P' the partial evaluation of A, $\mathbf{S}$ a set of subtrees, T' the extension of T by $\mathbf{S}$ and $T(A)$ the cSLDNF-tree for P' of depth two with root node $\leftarrow A$. If **S'** is the set obtained from $\mathbf{S}$ by renaming variables and commits such that they match the corresponding leaf-nodes of $T(A)$ and T^* is the extension of $T(A)$ by **S'** then we say that T^* is the *extension* of P' wrt T and $\mathbf{S}$.

Theorem 2.3.2 *Let P be a normal c-program, G a normal c-goal, $\mathbf{A}$ a finite, independent set of atoms and P' a free partial evaluation of P wrt $\mathbf{A}$ such that $P' \cup \{G\}$ is $\mathbf{A}$-closed and constructed with a (possibly) irregular cSLDNF-tree of the form in Figure 2.3. If S' is a pruned subtree of a cSLDNF-tree T' for $P' \cup \{G\}$ then there is a pruned subtree S of a cSLDNF-tree T for $P \cup \{G\}$ which has the same set of c-computed answers as S'.*

Proof We consider the case where $\mathbf{A}$ consists of the single atom A and our partial evaluation unfolds A to introduce a single new commit with label n. The extension to the more general case is straightforward. We shall also ignore branches arising from the unfolding of A which do not contain a commit as these are trivial cases for which Theorem 1.3.1 is sufficient.

Let T^* be cSLDNF-tree used to construct the partial evaluation of A, T the extension of T^* by some set $\mathbf{S}$, T' the extension of P' wrt T^* and $\mathbf{S}$, B_1 and B_2 leaf nodes of T and B'_1 and B'_2 the equivalent leaf-nodes of T'.

We prove the equivalence of the set of c-computed answers for S and S' by proving that there is a pruning step in T with cut node G_1 and explanation (G_2, B_2) such that B_1 is below G_1 iff there is a pruning step in T' with cut node G'_1 and explanation (G'_2, B'_2) such that B'_1 is below G'_1.

If part:

If we may perform a pruning step in T at a node G_0 with cut node G_1 and explanation (G_2, B_2) such that B_1 is below G_1 then G_1 and G_2 are distinct n-children of G_0.

There is a trivial case where G_0 is a node which is in T but not in T^*. G_0 is therefore a node in some subtree in $\mathbf{S}$. By the definition of extension there will therefore be an equivalent node G'_0 in T' at which we may perform an equivalent pruning step with cut node G'_1 and explanation (G'_2, B'_2) such that B'_1 is below G'_1.

The non-trivial case is when G_0 is in T^*. There are two possibilities, either that G_0 is the root node, $\leftarrow A$, or that G_0 is a node produced after step 2 but before step 3.

If G_0 is the root node $\leftarrow A$ then the augmented commit labels $n_a^{f:p}$ for all sub-nodes of G_1 are such that $a \neq b$ for all augmented commit labels $n_b^{g:q}$ of sub-nodes of G_2. By the rule for pruning with augmented labels there is therefore a pruning step at the root node in T' with cut node G'_1 and explanation (G'_2, B'_2) such that B'_1 is below G'_1.

If G_0 is a node produced after step 2 then the augmented labels for the nodes G_1 and G_2 are $n_a^{f:p}$ and $n_a^{f:q}$ respectively. As G_1 and G_2 are distinct, $p \neq q$. By the rule for pruning with augmented labels there is therefore a pruning step at the root node in T' with cut node G'_1 and explanation (G'_2, B'_2) such that B'_1 is below G'_1.

Only-if part:

If we may perform a pruning step in T' at a node G'_0 with cut node G'_1 and explanation (G'_2, B'_2) such that B'_1 is below G'_1 then G'_1 and G'_2 then either G'_0 is the root node, $\leftarrow A$, or G'_0 is a node below $\leftarrow A$. This latter case is equivalent to the trivial case for the if-part above and there is

therefore an equivalent node G_0 in a subtree in **S** and we may perform the necessary pruning step at this node.

In the first case there are distinct leaf nodes in T^*, equivalent to G'_1 and G'_2, with augmented commit labels $n_a^{f:p}$ and $n_b^{g:q}$ such that either $a \neq b$ or $f = g$ and $p \neq q$.

If $a \neq b$ then these leaf nodes in T^* have ancestors G_1 and G_2 which are distinct n-children of the root node $\leftarrow A$. We may therefore perform the necessary pruning step in T at $\leftarrow A$.

If $f = g$ and $p \neq q$ then these leaf nodes in T^* have distinct ancestors G_1 and G_2 which are n-children of some node G_0 produced after step 2 but before step 3. We may therefore perform the necessary pruning step in T at the node G_0. ∎

Finally in this section we briefly discuss the impact of the above theorem. Examination of almost any program containing commits will demonstrate that to restrict partial evaluation so that unfolding of any formula is only permitted within the scope of *all* commits in that formula would limit partial evaluation so severely as to make it of very little value as a program specialisation technique. Thus it was essential that to make partial evaluation a useful technique for Gödel programs it was necessary to remove the regularity condition from the partial evaluation theorem for programs with commits (Theorem 2.3.1).

In Theorem 2.3.2 we have weakened, but not removed, the regularity condition. In this we have sought only to prove the correctness of *SAGE*'s handling of the unfolding of formulas containing commits. We comment that removing the regularity condition completely would be possible by an extension of the techniques described above, leading to a strengthening of the above theorem. This is a result we expect to see verified in the near future.

2.4 Open and Closed Code

Gödel provides a module structure which we discuss in Section 2.6. However, there is one aspect of Gödel modules which we must introduce at this point.

In general, Gödel modules consist of two parts, an export part and a local part. The local part of a module consists of the code that defines the predicates declared in that module. The local part may also contain the declarations of the symbols which are used locally in that module. The export part of a module contains the declarations for those symbols which may be used by any other module which imports this module.

The particular feature of the module structure which we introduce here is concerned with the export part of modules. In general a Gödel meta-program would expect to be able to access the definition of any predicate defined in a module of some object program. The Gödel predicate `DefinitionInProgram` is used for this purpose. However, modules in a program may be of one of two kinds. The first kind of modules are modules 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*. In certain of the system modules the export part defines the module to be *closed*. A closed module is used to hide the details of the implementation of certain of the Gödel system modules, as a meta-program is unable to access the definition of a predicate with a call to `DefinitionInProgram` if that predicate is defined in the local part of a closed module. Any module which is not closed is an *open* 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.

To unfold a call to an atom whose definition is open, *SAGE* uses the Gödel system predicates `DefinitionInProgram` and `ResolveAll` to obtain this definition and use it to perform a single resolution step upon this atom. *SAGE* uses the Gödel system predicate `DeclaredInClosedModule` to detect atoms which are declared in closed modules and calls the Gödel system predicate `ComputeAll` to unfold these calls as far as possible.

2.5 Input/Output

Gödels' input/output facilities are supported by closed system modules and hence consist of closed code in the above sense. We repeat from [28] the motivation for the intended use of input/output in Gödel programs.

Gödel's input/output facilities do not have a declarative semantics, so it is very important that input/output predicates are confined to as small a part of a program as possible. Let us say a module is an *input/output module* if it depends upon the module `IO`. Then the key idea is to have the input/output modules as high as possible, preferably at the top, of the module hierarchy of a program. This means that most of the modules in such a program will not depend on the input/output modules and can be understood declaratively (together possibly with some commits). Fortunately, it *is* usual for the input/output modules to be near the top of the module hierarchy of a program. In fact, a common module structure is to have just the main module of a program importing the system input/output modules. In this case, all the other modules in the program have a declarative semantics. Furthermore, there is a strong incentive for programmers to adopt this kind of module structure, since it is only the modules that are not input/output ones to which the purest forms of program transformation, declarative debugging, and so on, apply. Given that programmers utilise such a module structure, it is expected that *SAGE* will generally be used to specialise the declarative core, that is to say, the non-input/output modules, of any program.

In fact *SAGE* is capable of processing calls to input/output predicates where necessary. As

the system modules that provide input/output predicates are closed modules the definitions of these predicates consist of closed code, in the sense of Section 2.4 above. Consequently, if *SAGE* encounters a call with some input/output predicate which is sufficiently instantiated to be executed, then it will interpret the execution of this call with a call to `ComputeAll`. If a call with some input/output predicate is encountered which is not sufficiently instantiated to be executed then `ComputeAll` will return this to *SAGE* as a delayed call.

2.6 Modules

The usual software engineering advantages of a module system are well known and apply equally well to Gödel. In its most basic form, a module system simply provides a way of writing large programs so that various pieces of the program do not interfere with one another because of name clashes and also provides a way of hiding implementation details. The Gödel module system is based on these standard ideas. When we partially evaluate a program however, it is often almost impossible to retain more than the barest semblance of the original program's module structure.

Consider the module `DB` which imports the module `People`. We could partially evaluate the atom `Human(x)` to produce the three new statements `Human(Bill)`, `Human(Fred)` and `Human(Mary)`. The constant symbols `Bill`, `Fred` and `Mary` are not accessible to the module `DB` and so we are unable to both replace the definition of the predicate `Human` in `DB` with the above three statements and retain the original module declarations for `People`. We could overcome the declaration problem in this case by requiring that the specialised version of the module `People` exported all symbols that it declared. When we force a module to export all of the symbols that it declares in this manner we say that we are causing this module to *promote* all of its symbols. However, allowing a partial evaluator to alter a program so that each module promotes all of its declared symbols is still not sufficient in general, as the next example illustrates.

Consider the module `Both` which imports the module `Member`. We could partially evaluate the atom `InBoth(x,Cons(A,Nil),y)` to produce the new statement defining `InBoth`, `InBoth(A,Cons(A,Nil),y) <- Member(A,y)`, and a new definition for the predicate `Member`, which will consist of the two specialised statements `Member(A,Cons(A,y))` and `Member(A,Cons(u,y)) <- Member(A,y)`. Here we have introduced into the specialised definition of the predicate `Member` a constant symbol, `A`, which is declared in a module that is not imported by the module `Member`. In the previous example we were able to overcome the problem caused by partial evaluation by promoting the symbols in the imported modules. In this case however, the constant symbol `A` is declared in the module `Both` and forcing `Both` to promote `A` will still not make this symbol accessible to the module `Member`. What has occurred in this case is a *demotion* of the constant symbol `A`. That is to say, the constant symbol `A` has now appeared in a module which was

---

```
MODULE       DB.

IMPORT       People.

PREDICATE    Human : Person.

Human(x) <-
             Male(x).
Human(x) <-
             Female(x).
```

```
EXPORT       People.

BASE         Person.

PREDICATE    Male : Person;
             Female : Person.
```

```
LOCAL       People.

CONSTANT     Bill, Fred, Mary : Person.

Male(Bill).
Male(Fred).
Female(Mary).
```

---

originally imported by the module in which `A` was declared.

```
MODULE      Both.

IMPORT      Member.

BASE        Data.

CONSTANT    A, B, C : Data.
PREDICATE   InBoth : a * List(a) * List(a).

InBoth(x,y,z) <-
            Member(x,y) &
            Member(x,z).
```

```
EXPORT      Member.

CONSTRUCTOR List/1.

CONSTANT    Nil : List(a).
FUNCTION    Cons : a * List(a) -> List(a).
PREDICATE   Member : a * List(a).
```

```
LOCAL       Member.

Member(x,Cons(x,y).
Member(x,Cons(u,y) <-
            Member(x,y).
```

What occurs in general in the partial evaluation of a Gödel program is a *flattening* of the module structure of that program. That is to say, the symbols declared in any program module may be either promoted or demoted so that they might appear in any other module of the program. This flattening will generally compress the program in a manner which makes it extremely difficult to

retain any sensible semblance of the module structure of the original program. In the worst case it is quite conceivable that a partial evaluation would flatten a program to such an extent that the entire module structure of the program would be lost and the specialised code for the residual program would need to be declared in a single large module. Consequently *SAGE* does not make any attempt to preserve the module structure of a program after specialisation and all residual programs are constructed as if they were essentially defined by a single residual module.

Gödel provides support for this flattening of Gödel programs via the concept of a *script*. A script is essentially a Gödel program from which all module structure has been removed. The system module `Scripts` supports a binary predicate `ProgramToScript` that converts the representation of a Gödel program to a Gödel script. A script, based upon some program, has a closed part and an open part. The closed part of a script consists of the language declarations, control declarations and code which appeared in the local parts of the closed modules in the original program. The open part of the script contains the complement of the closed part. It can be seen therefore that the code contained in the open and closed parts of the script will be, respectively, closed and open code, in the sense of Section 2.4. The module `Scripts` provides support for modifying the open part of a script only, as it is only the open code of a program which a Gödel meta-program is able to modify.

Meta-programs may access terms representing scripts by the use of the system module `ScriptsIO`, which supports predicates for reading and writing the representation of a script to and from a file. Lastly, Gödel provides facilities for viewing the representation of a script and converting the representation of a script to an executable program.

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 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.

Chapter 3

Specialising Gödel Meta-Programs

In this chapter we turn to what is probably the most important of all Gödel's facilities, the ground representation. The ground representation is certainly the most important facility for a self-applicable partial evaluator such as *SAGE*. Firstly, *SAGE* is a meta-program and therefore requires the ground representation for its implementation. Secondly, in order to be effectively self-applicable *SAGE* must be capable of specialising all of the Gödel facilities that it itself employs. Naturally then it follows that *SAGE* needs to be able to specialise Gödel meta-programs. As the ground representation is almost certainly the major cause of computational expense in any meta-program, it follows therefore that efficiently specialising the ground representation is of paramount importance for *SAGE*.

In this chapter we discuss the major causes for the expense of the ground representation. In terms of these major causes we then discuss the implementation and specialisation of the ground representation and describe a methodology for Gödel meta-programs that allows them to be effectively specialised by *SAGE*.

3.1 The Ground Representation in Gödel

As described in the previous chapter, the main facilities provided by the Gödel language are types, modules, control (in the form of constraint solving, control declarations and a pruning operator), meta-programming and input/output. This means that Gödel, being a rich and expressive language, has a complex syntax. As Gödel's ground representation is intended to be sufficient to represent Gödel programs it must allow for the construction of terms of sufficient complexity to describe arbitrary formulas and Gödel's types, modules, control, meta-programming and input/output facilities. The current implementation of the ground representation [7] requires some 75 constants and function symbols to construct the terms necessary to adequately represent the entire Gödel language. If all of these symbols were visible in Gödel meta-programs, it would be necessary for

```
VarsInTerm(term,vars) <-
  VarsInTerm1(term,[],vars).

VarsInTerm1(term,vars,[term|vars]) <-
  Variable(term).
VarsInTerm1(term,vars,vars) <-
  ConstantTerm(term,name).
VarsInTerm1(term,vars,vars1) <-
  FunctionTerm(term,name,args) &
  VarsInTerm2(args,vars,vars1).

VarsInTerm2([],vars,vars).
VarsInTerm2([term|rest],vars,vars1) <-
  VarsInTerm1(term,vars,vars2) &
  VarsInTerm2(rest,vars2,vars1).
```

Figure 3.1: Gödel code for `VarsInTerm`

the user to be familiar with the entire representation and competent in the manipulation of all of these symbols before he/she would be competent in the writing of meta-programs. To avoid confronting the user with such complexity unnecessarily, in Gödel, the representations of object level expressions and programs are treated as abstract data types. This also has the added advantage that meta-programs are independent of any specific implementation of the ground representation.

Example Figure 3.1 gives the Gödel code for finding the variables in an object level term. The predicates `Variable`, `ConstantTerm` and `FunctionTerm` are provided by Gödel. The first argument to such predicates are, respectively, the representations of object level variables, constants, and terms with a function at the top level.

The ground representation is an extremely powerful tool for meta-programming. However, it has the disadvantage of considerably increasing computation time. For example, consider an interpreter that computes the answer for some object program and query, using SLDNF-resolution. In the current implementation of Gödel, such an interpreter will run at 100–200 times slower than executing the program and query directly.

There are two major contributory factors to the expense of the ground representation in Gödel. The first is a direct result of supporting the ground representation as an abstract data type. The second, and potentially more serious, factor is that when using the ground representation the process of unification must be performed explicitly. However, the expense incurred by both of these factors

has been overcome by specialising meta-programs with respect to particular object programs. We discuss the above two factors, and their solutions, in more detail in the following two sections.

3.2 Specialising the Representation of Gödel

The major disadvantage of supporting the ground representation as an abstract data type is that we pay a price for not making visible those constants and function symbols used by the ground representation. Consider the predicate `VarsInTerm1` in Figure 3.1, which has three statements in its definition. In each statement the first argument (which is the key argument) in the head of the statement is a variable. As such, no implementation of Gödel would be capable of differentiating between the three statements at the time of procedure entry. Thus a choicepoint would need to be created and the execution time of the above code is increased by the time taken to create this choicepoint and also by any necessary backtracking. The use of choicepoints will also inhibit garbage collection. As meta-programs using the ground representation often process some very large terms (for example, the *representation* of *SAGE* is a Gödel term of approximately 1MByte in size), garbage collection is very important. Any impairment to the efficiency of garbage collection will potentially cause a serious increase in the memory-usage of a meta-program. We need therefore to prevent the creation of these superfluous choicepoints.

Ideally we would like to be able to perform some form of indexing upon the first arguments to `VarsInTerm1`. If the constants and function symbols used in Gödel's representation were accessible to the user, rather than hidden by the abstract data type, we would be able to use these symbols in the definition of `VarsInTerms1` and thus could perform first argument indexing upon this predicate. Such indexing would prevent the need for the creation of choicepoints and all the attendant expense. In our experience, meta-programs which are written without access to the symbols in the ground representation currently run up to three times slower than equivalent programs that do have access to the ground representation. Fortunately, through program specialisation, it is possible for a meta-program written without access to the symbols in the ground representation to achieve the efficiency of one that has. The use of partial evaluation to remove the overheads of abstract data types has also been proposed by Komorowksi [40].

In Gödel's representation, variables are represented by a term `Var(v,n)`, where `v` is a string and `n` an integer (this representation for variables is described in more detail below); constant terms are represented by a term `CTerm(name)`, where `name` is a Gödel term representing the name of this constant; function terms are represented by a term `Term(name,args)`, where `name` is the representation of the name of this function term and `args` is the list of representations of its arguments.

We may specialise the Gödel code in Figure 3.1, even without further knowledge of the values

```
VarsInTerm1(Var(v,n),vars,[Var(v,n)|vars]).
VarsInTerm1(CTerm(name),vars,vars).
VarsInTerm1(Term(name,args),vars,vars1) <-
  VarsInTerm2(args,vars,vars1).
```

Figure 3.2: Specialised code for `VarsInTerm1(term,vars,vars1)`

of any arguments. The first atom in the body of each statement in the definition of `VarsInTerm1` may be unfolded. The result of this will be to make visible the relevant function symbols in Gödel's ground representation. Figure 3.2 illustrates the specialised code for `VarsInTerm1`. As the relevant function symbols representing variables, constant and function terms now appear in the first argument of the heads of the statements defining `VarsInTerm1`, the Gödel system may perform first argument indexing to differentiate between the three statements. Consequently, when a call is made to `VarsInTerm1` with the first argument instantiated, no choicepoints are created and no backtracking is necessary at any point in the computation. When such specialisations are performed upon an entire meta-program the resulting gains in efficiency are considerable.

SAGE is capable of performing an automatic specialisation of the code in Figure 3.1. The residual code will leave the definitions of the predicates `VarsInTerm` and `VarsInTerm2` unchanged, and replace the definition of `VarsInTerm1` with the code in Figure 3.2.

3.3 Specialising Resolution in the Ground Representation

The greatest expense incurred by the use of the ground representation occurs in the manipulation of substitutions. When any variable binding is made, this must be explicitly recorded. Thus any unification, and similarly the composition and application of substitutions, must be performed explicitly. This produces significant overheads in the manipulation of the representations of terms and formulas. In this section we discuss how this expense may be greatly reduced, potentially leading to a specialised form of unification that is comparable to the WAM code [2, 76] for the object program. The need to specialise an explicit unification algorithm for efficiency has also been investigated in [16, 41]. Specialising meta-interpreters for propositional logic to produce WAM-like code has been investigated in [54].

In meta-programming the main manipulations of substitutions occur during resolution or unfolding, where we must unify an atom in some goal with a statement in the object program. Figure 3.3 gives the main part of a very simple Gödel meta-interpreter for definite programs. It is in the third statement of this program that we see the Gödel predicate `Resolve` being used to resolve an atom in the current goal with respect to a statement selected from the object program.

```
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) &
  StatementMatchAtom(program, module, 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).
```

Figure 3.3: A Simple Gödel Meta-Interpreter

The implementation of `Resolve` must handle 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 `Resolve` be implemented as efficiently as possible.

By contrast to the use of `Resolve`, as in the interpreter of Figure 3.3, consider the somewhat naive (although still declarative) interpreter of Figure 3.4. The third statement in the interpreter performs the same task as that of the third statement in the interpreter of Figure 3.3. However this naive interpreter is arguably more obtuse than that of Figure 3.3, as the manipulation of formulas and substitutions is here being performed explicitly. There would appear to be two very strong arguments for avoiding this style of meta-programming. The first is that it is more arduous for a programmer, requiring as it does explicit and sophisticated manipulation of formulas and substitutions. The second is not immediately apparent, but it is that the implementation of the interpreter of Figure 3.4 would be noticeably less efficient than that of Figure 3.3. Furthermore,

```
Demo(program, goal, subst, subst) <-
  EmptyFormula(goal).
Demo(program, goal, subst_in, subst_out) <-
  And(left, right, goal) &
  Demo(program, left, v_in, new_v, subst_in, new_subst) &
  Demo(program, right, new_v, v_out, new_subst, subst_out).
Demo(program,goal,subst_in,subst_out) <-
  Atom(goal) &
  StatementMatchAtom(program, module, goal, statement) &
  RenameFormulas([goal], [statement], [statement1]) &
  IsImpliedBy(head, body, statement1) &
  ApplySubstToFormula(goal, subst_in, goal1) &
  UnifyAtoms(goal1, head, mgu) &
  ComposeTermSubsts(subst_in, mgu, new_subst) &
  Demo(program, body, new_subst, subst_out).
```

Figure 3.4: A Naive Gödel Meta-Interpreter

the interpreter of Figure 3.3 may be specialised with respect to an object program in order to remove the majority of the expense of the ground representation, as we shall describe below. With the inherent inefficiencies of the interpreter of Figure 3.4 however, with its repeated explicit manipulation of the representations of the atom, statement and current substitution, it is far from clear that any specialisation could specialise the resolution process to the same extent.

3.3.1 Specialising Resolve

When we specialise a meta-program such as the interpreter in Figure 3.3 to a known object program, the statements in the object program will be known. Therefore we may 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. To see how this is achieved we must look more carefully at the implementation of `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.

```
P(x,y,z,F(x,u)) <-
  Q(x,x1) &
  P(x1,y,z,u).
```

Figure 3.5: A Gödel Statement

Variable Renaming

In a call to `Resolve`, all variables in the statements are renamed as they are encountered. This saves us from having to perform more than one pass over the statements during resolution. Any variable encountered, which could potentially appear in the new goal, is replaced by a variable with a name that does not occur elsewhere in the current computation. Variables in a statement fall into one of three categories, depending on where they are first encountered. These are:

1. The variable appears in an argument position in the head of the statement. This variable will be bound to the term in the atom's matching argument position and thus does not need to be renamed.

2. The variable appears as a subterm of a term in the head of the statement. This variable may need to be renamed, but this cannot be determined until the matching term in the atom is known.

3. The variable appears only in the body of the statement. This variable must be renamed.

For example, in the statement in Figure 3.5 the variables `x`, `y` and `z` are variables of the first type, variable `u` is of the second type and variable `x1` is of the third type. Thus while variable `x1` will certainly require renaming and variable `u` *may* require renaming, the remaining variables need not be renamed. To see how renaming is achieved we must look more closely at how variables are represented in Gödel.

When represented (by the term `Var(name,N)`), Gödel variables have names of the form `name_N`, where `name` is the *root* of the name of the variable (a string) and the non-negative integer `N` is called the *index* of the variable. To specialise renaming at all times we record `Max`, the highest integer index occurring in a variable in the current computation, and a new variable will be given the name `v_Max1`, where `Max1` is the increment of `Max`. In addition, new names are given only to variables that are guaranteed to occur in the resolvent. In this way the creation of new variables is kept to a minimum. A call to `Resolve` takes the increment of the current value of `Max` as its third argument and returns as its fourth argument the increment of the value of `Max` after all renaming has been performed. Thus specialising the renaming of the statement in Figure 3.5 of this statement

would create the terms `Var("v",max+1)` and (assuming that the variable `u` also required renaming) `Var("v",max+2)`, where `max` is the current highest variable index.

Applying the Current Substitution

Before we attempt to unify the atom with the head of the statement we must consider the possibility that certain variables in the atom will have become bound in the current substitution. Such bindings must be taken into consideration and yet to apply the current substitution to all the terms in the atom is an unnecessary expense. To reduce this expense we must consider the terms in the head of the statement, these terms will each be one of:

1. A variable. Unless this is a repeated variable then the unification of this variable with the matching term in the atom will always succeed. Thus we do not need to apply the current substitution to the matching term in the atom.

2. A constant. We must apply the current substitution to the matching term in the atom before attempting to unify it with this constant.

3. A term with a function at the top level. We must test whether the matching term in the atom is bound in the current substitution to either a variable or to a term with a matching function at the top level. If the term in the atom is bound to a term with a matching function at the top level then we will compare this term's arguments with the arguments of the term in the statement.

Note that in the third case, even though we must test whether the matching term in the atom is a term with a function at the top level, we do not necessarily need to apply the current substitution to the arguments of this term. In the statement in Figure 3.5 for example, if the fourth argument of an atom we wished to resolve with this statement were bound to some term F($\mathcal{T}_1$,$\mathcal{T}_2$), we would not need to apply the current substitution to the term $\mathcal{T}_2$ in order to unify it with the matching variable `u` in the term `F(x,u)`.

Head Unification in `Resolve`

The third operation to be performed in the resolution of an atom with a statement is the unification of the atom and the head of the statement. The unification algorithm employed enforces occur-checking for safeness. Although occur-checking is potentially very expensive, this expense may be greatly reduced by enforcing occur-checking for repeated variables in the head of the statement only.

After renaming, all variables in the statement are guaranteed not to appear elsewhere in either the current goal or the current substitution. This means that any bindings for variables in the head

of the statement may be applied to the body of the statement and then discarded. Consequently only that part of the mgu of the atom and the renamed head of the statement that records the bindings of variables in the atom will need to be composed with the current substitution in order to produce the new substitution.

For example, when unifying an atom with the statement in Figure 3.5, the bindings for the variables `x`, `y` and `z` in the statement are recorded separately from any potential bindings for variables in the atom. These bindings may then be applied to the body of the statement, replacing the variables `x`, `y` and `z` by the terms to which they have been bound. There will only be one potential occur-check during the unification of an atom with the head of this statement and that will be if the fourth argument of the atom is a term $\mathtt{F}(\mathcal{T}_1,\mathcal{T}_2)$. In this case the first argument of this function term will be unified with the first argument of the atom and an occur-check will be performed for this unification step alone.

Composition of the Mgu with the Current Substitution

Having performed the unification of an atom with the head of a statement we must in theory combine the mgu of this unification with the current substitution. In reality it is more efficient for any bindings made to variables in the atom to be composed with the current substitution immediately. In order to achieve these compositions we have a set of predicates, each of which performs one specific unification operation. The predicates which unify arguments of the head of the statement with the matching arguments of the atom are as follows:

`UnifyTerms(term1,term2,subst,subst1)` attempts to unify the atom's two terms `term1` and `term2`. `UnifyTerms` is the only one of these specific argument unification operations which enforces occur-checking and is used to unify repeated variables in the head of the statement. In this and the two subsequent atoms, `subst` is the current substitution and `subst1` is this substitution after the relevant unification step.

`GetConstant(term,constant,subst,subst1)` attempts to unify the atom's term `term` with the constant `constant`.

`GetFunction(term,function,mode,subst,subst1)` attempts to unify the atom's term `term` with a term `function` with a function at the top level. If `term` is bound in the current substitution to a variable then `mode` is set to `Write` and `function` will subsequently be instantiated to a renamed version of the term to which this variable is to be bound. If `term` is bound in the current substitution to a term with a matching function at the top level then `mode` is set to `Read`.

If an argument in the head of the statement is a term with a function at the top level, then there are two cases in which a call to `GetFunction` will succeed. In the first case the atom's matching argument is a variable and we must construct a renamed version of the term in the head of the statement and then bind this variable to it. In the second case the atom's matching argument is a term with a matching function at the top level and we must unify the arguments of this term with the corresponding arguments in the statement's term.

For example, the term `F(x,u)` appears in the head of a statement in Figure 3.5. Thus we make a call to `GetFunction` which will succeed with `mode` set to `Write` if the atom's fourth argument is bound to a variable in the current substitution and will succeed with `mode` set to `Read` if the atom's fourth argument is bound in the current substitution to some term F($\mathcal{T}_1$,$\mathcal{T}_2$).

The following predicates perform the unification operations necessary for processing the arguments of function terms in the head of the statement, either renaming variables when in `Write` mode or unifying these arguments with the arguments of the matching function term in the atom when in `Read` mode.

`UnifyVariable(mode,term,var,ind,ind1)` in `Write` mode will instantiate `var` to the new variable `Var("v",ind)` and `ind1 = ind+1`. In `Read` mode, `var` is instantiated to the atom's term `term` and `ind1 = ind`.

`UnifyValue(mode,term,term1,subst,subst1)` in `Write` mode will instantiate `term1` to `term`. In `Read` mode this call will unify (with occur-checking) the atom's two terms `term` and `term1`. In this and the two subsequent atoms, `subst` is the current substitution and `subst1` is this substitution after the relevant unification step.

`UnifyConstant(mode,term,constant,subst,subst1)` in `Write` mode will instantiate `term` to the constant `constant`. In `Read` mode this call attempts to unify the atom's term `term` with the constant `constant`.

`UnifyFunction(mode,term,function,mode1,subst,subst1)` in `Write` mode will instantiate `term` to the term `function` and `mode1` is set to `Write`. In `Read` mode this call attempts to unify the atom's term `term` with a term `function` with a function at the top level. If `term` is bound in the current substitution to a variable then `mode1` is set to `Write` and `function` will subsequently be instantiated to a renamed version of the term to which this variable is to be bound (as for `GetFunction`). If `term` is bound in the current substitution to a term with a matching function at the top level then `mode` is set to `Read`.

Example Figure 3.6 illustrates the result of specialising `Resolve` with respect to the statement in Figure 3.5. In the second argument in the head of this specialised statement, the term `statement`

```
Resolve(
  Atom(P', [arg1, arg2, arg3, arg4]),
  statement,
  v, v1+1 ,
  subst_in, subst_out,
  Atom(Q', [arg1, Var("v", v1)]) &' Atom(P', [Var("v", v1), arg2, arg3, var])
      ) <-
    GetFunction(arg4, F'([sub1, sub2]), mode, subst_in, new_subst) &
    UnifyValue(mode, arg1, sub1, new_subst, subst_out) &
    UnifyVariable(mode, sub2, var, v, v1).
```

Figure 3.6: Specialised code for `Resolve`

denotes the representation of the statement in Figure 3.5, which we have omitted for the sake of brevity. The residual calls in the body of the specialised call to `Resolve` unify the atom's fourth argument with a term with a function named `F` at the top level and two arguments. If the atom's fourth argument is bound to a variable in `subst_in` then `mode` is set to `Write` by the call to `GetFunction`, which also binds this variable, in `new_subst`, to a new term with this function at the top level. The subsequent calls to `UnifyValue` and `UnifyVariable` will then instantiate the arguments of this new function term to the atom's first argument, `arg1`, and a new variable, `var`. They will also set `subst_out = new_subst` and `v1 = v+1`. If the atom's fourth argument is bound in `subst_in` to a term with a matching function symbol at the top level then `mode` is set to `Read` and `new_subst = subst_in`. The call to `UnifyValue` then unifies, with occur-checking, the atom's first argument, `arg1`, with the first argument, `sub1`, of this function term. If successful, this unification will return the new substitution `subst_out`. The call to `UnifyVariable` then instantiates `var` to the second argument, `sub2`, of the atom's function term and sets `v1 = v`.

A more complex example of the specialised code for `Resolve` is given in Figure 3.7. Here, by specialising `Resolve` to the statement `P(x,x,A,F(y,F(x,A))) <- Q(y)` we may see an example of a call to each of the seven predicates described above.

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. 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.

```
Statement:  P(x, x, A, F(y, F(x, A))) <- Q(y).
Specialised call to Resolve:

Resolve(
  Atom(P', [arg1, arg2, arg3, arg4]),
  statement,
  v, v1,
  subst_in, subst_out,
  Atom(Q', [var])
      ) <-
    UnifyTerms(arg1, arg2, subst_in, s1) &
    GetConstant(arg3, A', s1, s2) &
    GetFunction(arg4, F'([sub1, sub2]), mode, s2, s3) &
    UnifyVariable(mode, sub1, var, v, v1) &
    UnifyFunction(mode, sub2, F'([sub21, sub22]), mode1, s3, s4) &
    UnifyValue(mode1, arg1, sub21, s4, s5) &
    UnifyConstant(mode1, sub22, A', s5, subst_out).
```

Figure 3.7: More specialised code for `Resolve`

The local part for the module `Substs` given here illustrates a potential implementation of these WAM-like predicates, other than `UnifyTerms` which we describe in detail in appendix A.1. A call `BindVariable(v,t,s,s1)` is used in this module to add a variable binding, where `v` is the representation of a variable, `t` the representation of a term, `s` the representation of a substitution θ and `s1` the representation of the substitution $\theta \circ \{\texttt{v/t}\}$. It should be noted that these WAM-like predicates support the implementation of the representation of term substitutions as an abstract data type. That is to say, the actual implementation of the representation of a substitution is hidden, even in the specialised code for a Gödel meta-program that is partially evaluated by *SAGE*. The current implementation for the representation of substitutions and the corresponding implementation for the WAM-like predicates is described in more detail in appendices A.2 and A.3 respectively.

As a final note on the optimisation of the composition of the mgu with the current substitution it should be noted that composition is in general a very expensive operation. This expense may be significantly reduced by careful implementation of the representation of term substitutions. As we have stated above, details of this implementation are hidden from the user and are independent of the implementation of `Resolve` described here. A discussion on the expense of composition of substitutions and how we avoid it is included in appendix A.2.

```
LOCAL         Substs.

GetConstant(term, constant, subst, subst1) <-
  IF  term = constant
    THEN subst1 = subst
    ELSE Variable(term) &
         BindVariable(term, constant, subst, subst1).

GetFunction(term, function, mode, subst, subst1) <-
  FunctionTerm(function, functor, args) &
  IF  FunctionTerm(term, functor, _)
    THEN subst1 = subst &
         mode = Read
    ELSE Variable(term) &
         mode = Write
         BindVariable(term, function, subst, subst1).

UnifyVariable(Write, _, variable, var, var+1) <-
  VariableName(variable, "v", var).
UnifyVariable(Read, term, term, var, var).

UnifyValue(Write, term, term, subst, subst).
UnifyValue(Read, term1, term2, subst, subst1) <-
  UnifyTerms(term1, term2, subst, subst1).

UnifyConstant(Write, constant, constant, subst, subst).
UnifyConstant(Read, term, constant, subst, subst1) <-
  GetConstant(term, constant, subst, subst1).

UnifyFunction(Write, function, function, Write, subst, subst).
UnifyFunction(Read, term, function, mode, subst, subst1) <-
  GetFunction(term, function, mode, subst, subst1).
```

3.3.2 Implementing `Resolve`

Following the above explanation of the optimisations underlying the ground representation of resolution in Gödel and the introduction of the WAM-like predicates, we may now present the Gödel code for the implementation of `Resolve`.

The first block of code gives the implementation of the unification of the atom with the head of the statement. Here variables with the prefix `v_max` hold the values of the maximum variable index, variables with the prefix `push` hold the bindings for the variables in the head of the statement and variables with the prefix `subst` hold the representation of the current substitution.

```
Resolve(Atom(p,a1),Atom(p,a) <-' body,v_max,v_max1,subst,subst1,new_body) <-
  UnifyArgs(a,a1,v_max,v_max2,[],push,subst,subst1) &
  ApplySubstToFormula(body,v_max2,v_max1,[],push,_,new_body).

UnifyArgs([],[],v_max,v_max,push,push,subst,subst).
UnifyArgs([st_term|rest1],[atom_term|rest2],v_max,v_max1,push,push1,s_in,s_out) <-
  UnifyTerms(st_term,atom_term,v_max,v_max2,push,push2,s_in,subst1) &
  UnifyArgs(rest1,rest2,v_max2,v_max1,push2,push1,subst1,s_out).

UnifyTerms(Var(s,i),atom_term,v_max,v_max,push,push1,subst_in,subst_out) <-
  IF  SOME  [value]  Member(Var(s,i) ! value,push)
    THEN  UnifyTerms(value,atom_term,subst_in,subst_out) &
          push1 = push
    ELSE  subst_out = subst_in &
          push1 = [Var(s,i) ! atom_term|push].
UnifyTerms(Term(f,args),atom_term,v_max,v_max1,push,push1,subst_in,subst_out) <-
  GetFunction(atom_term,Term(f,atom_args),mode,subst_in,subst1) &
  CheckFunctionTerm(args,atom_args,mode,v_max,v_max1,push,push1
                  ,subst1,subst_out).
UnifyTerms(CTerm(t),atom_term,v_max,v_max,push,push,subst_in,subst_out) <-
  GetConstant(atom_term,CTerm(t),subst_in,subst_out).

CheckFunctionTerm([],[],_,v_max,v_max,push,push,bind,bind).
CheckFunctionTerm([a|rest],[a1|rest1],mode,v_max,v_max1,push,push1,bind,bind1) <-
  CheckFunctionTerm1(a,a1,mode,v_max,v_max2,push,push2,bind,bind2) &
  CheckFunctionTerm(rest,rest1,mode,v_max2,v_max1,push2,push1,bind2,bind1).

CheckFunctionTerm1(Var(s,i),arg1,mode,v_max,v_max1,push,push1,bind,bind1) <-
  IF  SOME  [value]  Member(Var(s,i) ! value,push)
    THEN  UnifyValue(mode,value,arg1,bind,bind1) &
          v_max1 = v_max &
```

```
          push1 = push
    ELSE  UnifyVariable(mode,arg1,v_max,v_max1) &
          push1 = [Var(s,i) ! arg1|push] &
          bind1 = bind.
CheckFunctionTerm1(Term(f,args),arg1,mode,v_max,v_max1,push,push1,bind,bind1) <-
  UnifyFunction(mode,arg1,Term(f,atom_args),mode1,bind,bind2) &
  CheckFunctionTerm(args,atom_args,mode1,v_max,v_max1,push,push1,bind2,bind1).
CheckFunctionTerm1(CTerm(t),arg1,mode,v_max,v_max,push,push,bind,bind1) <-
  UnifyConstant(mode,arg1,CTerm(t),bind,bind1).
```

Having unified the atom with the head of the statement, we turn our attention to the body of the statement. Any variables in the body of the statement which appeared in the head will have been bound to some new term, either a term in the atom or a new variable. These bindings are held as a list of bindings by variables with the prefix `push`, as above. These bindings will also now be augmented by the new variable names which are used to rename those variables which appear only in the body of the statement.

The next block of code gives this operation for an illustrative sample of the potential formulas in the body of the statement. Note that particular care is taken with quantified variables. A locally quantified variable may share the same name with a variable appearing outside the scope of the quantifier. When processing the variables in the body of the statement we ensure that locally quantified variables are renamed so that their new names differ from all other variables in the statement. Consequently, having performed a resolution step we may be sure that the names of locally quantified variables will not appear outside the scope of the quantifier. The new names for these locally quantified variables are held separately from the other bindings by the variables with the prefix `quant`.

```
ApplySubstToFormula(Empty,v_max,v_max,_,push,push,Empty).
ApplySubstToFormula(left &' right,v_max,v_max1,quant,push,push1,l1 &' r1) <-
  ApplySubstToFormula(left,v_max,v_max2,quant,push,push2,l1) &
  ApplySubstToFormula(right,v_max2,v_max1,quant,push2,push1,r1).
ApplySubstToFormula(Atom(p,args),v_max,v_max1,quant,push,push1,Atom(p,args1)) <-
  ApplySubstToArgs(args,v_max,v_max1,quant,push,push1,args1).
ApplySubstToFormula(Some(s,f1),v_max,v_max1,quant,push,push1,Some(s1,f2)) <-
  StandardiseApartQuants(s,s1,quant,quant1,v_max,v_max2) &
  ApplySubstToFormula(f1,v_max2,v_max1,quant1,push,push1,f2).

StandardiseApartQuants([],[],quant,quant,max,max).
StandardiseApartQuants([var|r],[Var("v",mx)|r1],qnt,qnt1,mx,mx1) <-
        StandardiseApartQuants(r,r1,[var ! Var("v",mx)|qnt],qnt1,mx+1,mx1).
```

When processing a particular variable in the body of the statement we will have one of the following cases:

- The variable is locally quantified. In this case we replace this variable with the new name assigned to it.
- The variable is not locally quantified, but has been previously encountered in either the head of the statement or a part of the statement body which has already been processed. In this case a new value will already have been assigned for this variable and we replace the variable with this new value.
- The variable has not yet been encountered. In this case a new name is generated for this variable and this assignment is recorded.

The last block of code gives the implementation of these operations.

```
ApplySubstToArgs([],v_max,v_max,_,push,push,[]).
ApplySubstToArgs([arg|rest],v_max,v_max1,quant,push,push1,[arg1|rest1]) <-
  ApplySubstToTerm(arg,v_max,v_max2,quant,push,push2,arg1) &
  ApplySubstToArgs(rest,v_max2,v_max1,quant,push2,push1,rest1).

ApplySubstToTerm(Var(s,i),max,max1,quant,push,push1,term1) <-
  IF  SOME  [term2]  Member(Var(s,i) ! term2,quant)
    THEN  max1 = max &
          push1 = push &
          term1 = term2
    ELSE  IF  SOME  [term2]  Member(Var(s,i) ! term2,push)
            THEN  max1 = max &
                  push1 = push &
                  term1 = term2
            ELSE  max1 = max+1 &
                  push1 = [Var(s,i) ! term1|push] &
                  term1 = Var("v",max).
ApplySubstToTerm(Term(f,args),max,max1,quant,push,push1,Term(f,args1)) <-
  ApplySubstToArgs(args,max,max1,quant,push,push1,args1).
ApplySubstToTerm(CTerm(term),v_max,v_max,_,push,push,CTerm(term)).
```

3.3.3 Handling Committed Formulas with `ResolveAll`

There is one last aspect of standard Gödel statements that we must consider before our description of the implementation of resolution in the ground representation is complete. This is the issue of the renaming of commits in Gödel statements.

For much the same reason that variables in a statement must be renamed before resolution to avoid clashing with the names of variables in the current goal, so must the labels of commits in

statements be renamed before resolution. However, the labels (which are integer values) of the commits in a statement must be renamed in the context of the entire *definition* of the predicate.

For example, given the following definition of a predicate P:

```
P(x, y) <-
  { Q(x) }_1 & R(y).
P(x, y) <-
  { S(x) }_1 & T(y).
P(x, y) <-
  { U(x) }_1 & V(y).
```

suppose that we wish to rename the commit label in one of these statements. We could achieve this by replacing the integer label 1 by the new integer max_com+1, where max_com is the highest integer value used in the label of a commit in the current computation. However, if we subsequently wished to rename the commit label in a second of these statements in a manner that preserved the scope of this commit then we would need to rename this label with the same integer value that was used to rename the first statement. We would similarly need to use the same new label again to rename the third statement correctly.

To implement this renaming of commit labels correctly it would therefore seem obvious that we should rename all of the commit labels in the entire definition of the predicate at the same time. To assist the user in achieving this Gödel provides a more powerful version of Resolve which is named ResolveAll. ResolveAll performs resolution by resolving an atom with respect to the entire definition of the predicate which this atom matches, returning two lists. The first list contains the bodies of the statements which have successfully been resolved with the atom and the second list returns the relevant new substitutions for these resolutions.

The atom ResolveAll(atom,sts,v,v1,c,c1,s,ss,bs) is called to perform the resolution of the atom atom with the list, sts, of the statements defining the predicate that matches the predicate of atom. The integers v and v1 are used to rename the statements in this list, as in Resolve. In a similar fashion the integers c and c1 are used to rename the commit labels in these statements. The representation of the term substitution s represents the current answer substitution. The last two arguments to ResolveAll, ss and bs, represent respectively the list of new answer substitutions and the corresponding list of the bodies of those statements in sts for which the resolution was successful.

The first block of code for ResolveAll illustrates the renaming of commits in the statements we are resolving the atom with respect to. Here the bodies of the statements are analysed to find the commits in them and new commit labels are generated for those commits which have not already been renamed during the resolution of the other statements.

```
ResolveAll(atom,ss,var,var1,com,com1,subst,substs,ress) <-
  ResolveAll1(ss,atom,var,var1,com,com1,[],subst,substs,ress).

ResolveAll1([],_,var,var,com,com,_,_,[],[]).
ResolveAll1([head <-' body|rest],atom,v,Max(v1,v2),c,c1,new_coms,subst,ss,ress) <-
  StandardiseCommits(body,c,c2,new_coms,new_coms1) &
  (
  IF SOME [v3,s1,body1] Resolve(atom,head,body,v,v3,new_coms1,subst,s1,body1)
     THEN  ress = [body1|ress1] &
           ss = [s1|ss1] &
           v1 = v3
     ELSE  ress = ress1 &
           ss = ss1 &
           v1 = v
  ) &
  ResolveAll1(rest,atom,v,v2,c2,c1,new_coms1,subst,ss1,ress1).

StandardiseCommits(left &' right, c, new_c, label_dict, new_label_dict) <-
  StandardiseCommits(left, c, c1, label_dict, label_dict1) &
  StandardiseCommits(right, c1, new_c, label_dict1, new_label_dict).
StandardiseCommits(Commit(label, formula), c, new_c, label_dict, label_d) <-
  NewCommit(label, c, c1, label_dict, label_dict1) &
  StandardiseCommits(formula, c1, new_c, label_dict1, label_d).
StandardiseCommits(Empty, c, c, label_dict, label_dict).
StandardiseCommits(Atom(_,_), c, c, label_dict, label_dict).

NewCommit(label, c, new_c, label_dict, new_label_dict) <-
  IF SOME [label1] Member(LabelPair(label, label1), label_dict)
    THEN
      new_c = c &
      new_label_dict = label_dict
    ELSE
      new_label_dict = [LabelPair(label, c)|label_dict] &
      new_c = c + 1.
```

It should be noted that the renaming of commits is not performed immediately. Instead we have added an extra argument to `Resolve` which is used to pass on a list of the values used to rename the commits. The definition of `Resolve` is modified so that this list is passed on as an extra argument to `ApplySubstToFormula`. The modified definition for `ApplySubstToFormula` is virtually identical to the definition given in the previous section, the only difference being that we are now able to include the case where the body of the statement contains committed formulas.

```
ApplySubstToFormula(Commit(l,f),v_max,v_max1,quant,p,vs,vs1,coms,Commit(l1,f1)) <-
  MemberCheck(LabelPair(l,l1),coms) &
  ApplySubstToFormula(f,v_max,v_max1,quant,p,vs,vs1,coms,f1).
```

The Gödel code for `ResolveAll` given above is in fact the code that forms the heart of $SAGE's$ unfolding process. As such, it was designed with the intention that it should be both efficient and able to be specialised in order to produce highly optimised residual code. Thus specialising `ResolveAll` illustrates a part of the self-application of *SAGE*. It also highlights our main aim in the definition of `ResolveAll`, which was to produce a declarative implementation of resolution for the ground representation that was both efficient and capable of producing significantly more efficient code upon specialisation. It is from the original implementation of *SAGE* that we have developed Gödel's current implementation of substitutions, unification and resolution, so that the code for `Resolve` and `ResolveAll` may now also be utilised by other meta-programs and specialised by *SAGE* in order to remove the overheads of the ground representation, while retaining the power of meta-programming.

A Word on the Full WAM Engine

Previous work in [54] has shown that it is possible to emulate that part of the WAM that deals with the execution of alternative statements in a predicate definition. That is, that part of the WAM which uses the Try-Me-Else, Retry-Me-Else and Trust-Me instructions to handle choicepoints. This is achieved by the specialisation of an abstract machine (essentially a meta-interpreter) with respect to lists of statements in a propositional logic and is thus very closely related to that which we have achieved above for the predicate logic case.

It would be possible for us to combine our techniques presented here for specialising the resolution of individual statements in a predicate logic with the techniques of [54] in order to arrive at a more sophisticated implementation of `ResolveAll`. This new implementation could conceivably be further extended to also emulate first-argument indexing over alternative statements in the definition of a predicate. This would give us a definition of `ResolveAll` for which the specialised code would emulate the full WAM instructions for the definitions of predicates, incorporating the WAM's handling of choicepoints and first-argument indexing. Unfortunately, to choose this approach would also force the implementation of `ResolveAll` to interpret the process of resolution in the sequential manner inherent to the WAM and thus it would not have the declarative aspect that we wished for. Consequently we have not taken this approach in the above implementation.

3.3.4 Specialising Unification

Gödel provides predicates to handle the general unification of the representations of types, terms and atoms. `UnifyTerms` for example, is called by `Resolve` when two terms in the atom to be resolved need to be unified. The main difference between the intended applications of `Resolve` and, for example, `UnifyTerms` is that while `Resolve` is intended to be used in cases when we may be certain that variables in the statement that we are resolving will not appear in the current answer substitution, we do not have this assurance for `UnifyTerms`. That is to say, when calling `Resolve` to perform a resolution step with respect to a statement we would expect that the variables in the statement would be renamed by new variable names that do not occur elsewhere in the current computation. In general `UnifyTerms` will be called to unify two terms, both of which may contain variables that occur elsewhere in the current computation and may therefore be bound in the current answer substitution.

`UnifyTerms(t1,t2,s,s1)` takes as arguments the representations, `t1,t2`, of the two terms to be unified and the representations, `s,s1`, of two substitutions. These two (representations of) substitutions may be viewed as 'input' and 'output' substitutions, representing the state of some computed answer both before and after the unification is performed. To be precise, when the first two arguments to `UnifyTerms` are the representations of the terms t_1 and t_2, respectively, and the third argument is the representation of the substitution θ, then the fourth argument is the representation of the composition, $\theta \circ \phi$, of the substitutions θ and ϕ, where ϕ is a unique, specific most general unifier for the terms $t_1\theta$ and $t_2\theta$.

As the input substitution must be applied to both of the two terms to be unified, we are unable to produce an implementation of `UnifyTerms` that we may specialise to the extent that we may specialise `Resolve`.

For example, to specialise the atom `UnifyTerms(Var("v",1),t,s,s1)` we would need firstly to determine whether the variable `Var("v",1)` was bound to any other value in the substitution `s`. As `s` is unknown at this point, this is not possible and therefore it seems fruitless to attempt to specialise further.

The implementation for `UnifyTypes` and `UnifyAtoms` is similar to that for `UnifyTerms`. The code for `UnifyTerms` is presented in appendix A.1.

3.4 A Framework for Meta-Programs

Meta-programs are generally large complex programs. We may utilise our knowledge of the similarities in the structure of most meta-programs to guide specialisation. By providing a general skeletal structure for meta-interpreters, we discuss the specialisation of such programs into a form that is as close as possible to our intuitive idea of an ideal residual program. Later we show

how *SAGE* performs an automatic specialisation matching the intuitive one.

Our approach has been to perform a static analysis of the program and query to be specialised. Information gained about the structure of the program, particularly those predicates whose unrestricted unfolding may potentially lead to infinite unfolding, is then used to guide the unfolding process. The information gained from the static analysis, coupled with our knowledge about the general structure of meta-programs, is used to guide the partial evaluation and thus influence the structure of the residual program. This structure for the residual program is such that code explosion is avoided. We use the term code explosion here to cover two forms of inefficiency. Firstly, the introduction or duplication of redundant terms and code and secondly over-eager unfolding leading to an explosion of run-time search. It is as a consequence of our technique for avoiding code explosion that the unfolding process is guaranteed to terminate.

3.4.1 Structure of Meta-Interpreters

To illustrate the structures of meta-programs and their specialised versions we examine interpreters that use Gödel's ground representation. In general such programs, which compute the answer-substitution of some query with respect to some object program, may perform some preprocessing of the program and query to be interpreted. Following this the query is interpreted and some postprocessing may then be performed upon the computed answer. The interpretation of the goal will generally be handled by recursively performing single resolution steps upon the currently computed goal. Figure 3.8 illustrates this structure. In this program, the main predicate `Demo` performs the preprocessing, calls predicate `Select` to select some subgoal (the leftmost atom, for example) from the current goal and calls `Demo1` to compute the answer, which is then postprocessed. The recursive predicate `Demo1` returns the answer substitution if the computation has terminated (base case) and otherwise performs a single resolution step followed by a selection step and then calls `Demo1` again (recursive case).

The code for `SimpleResolve` is described in Figure 3.11 and performs a single resolution step. It should be noted here that `SimpleResolve` is a gross simplification for the predicate `Resolve` and is in fact insufficient for correct meta-programming as the renaming of statements is not considered. We introduce this simpler version of `Resolve` purely for the sake of brevity and hence we do not consider here standardisation apart and the other features of `Resolve`.

3.4.2 An (Intuitive) Ideal Specialisation

The greatest expense incurred by the use of the ground representation occurs in the manipulation of substitutions. When any variable binding is made, this must be explicitly recorded. Thus any unification, and similarly the composition and application of substitutions, must be performed

```
Demo(program,query,answer) <-
  Preprocess(program,query,goal) &
  EmptyTermSubst(empty) &
  Select(goal,left,selected,right) &
  Demo1(program,selected,left,right,empty,subst) &
  Postprocess(program,subst,answer).

Demo1(_,selected,_,_,subst,subst) <-
  EmptyFormula(selected).
Demo1(program,selected,left,right,subst_in,subst_out) <-
  StatementMatchAtom(program,selected,statement) &
  SimpleResolve(selected,statement,body,subst_in,subst1) &
  And(left,body,left1) &
  And(left1,right,goal) &
  Select(goal,l,s,r) &
  Demo1(program,s,l,r,subst1,subst_out).
```

Figure 3.8: Basic Structure for Meta-Interpreters

explicitly. This produces significant overheads in the manipulation of the representations of terms and formulas. In Section 3.3 it was shown how this expense may be greatly reduced by specialising resolution with respect to a specific object program. In the interpreter of Figure 3.8 we may specialise the predicate `SimpleResolve` with respect to the statements in some object program in this manner. Having performed such a specialisation we would therefore wish to replace the recursive statement defining `Demo1` with, for each statement in the object program, a specialised statement for `Demo1` for which the call to `SimpleResolve` has been specialised to the object statement.

Figure 3.9 illustrates the structure that, we argue, is most suitable for a specialised version of the interpreter in Figure 3.8. Here we have specialised the meta-interpreter with respect to a program containing two statements defining, respectively, predicates `P` and `Q`. The recursive statement in the definition of `Demo1` has been specialised to produce one statement for each of these two statements in the object program. Note that while the definitions of `Demo1` and `Select` have been specialised, calls to these predicates in the definitions of `Demo` and `Demo1` have not been unfolded (as `Preprocess` and `Postprocess` have).

There are two notational conventions that we introduce here. In the first we have removed certain superfluous terms in the specialised code. For example, in Figure 3.9 we have removed the terms representing the object program which we have specialised the interpreter with respect to. Thus we have replaced an atom such as `Demo(program,goal,answer)` by the atom `Demo(goal,answer)`.

```
Demo(query,answer) <-
  Preprocess*(query,goal) &
  Select(goal,left,selected,right) &
  Demo1(selected,left,right,EmptySubst,subst) &
  Postprocess*(subst,answer).

Demo1(EmptyFormula,_,_,subst,subst).
Demo1(P(args),left,right,s_in,s_out) <-
  SimpleResolve*(args,body,s_in,s1) &     % Specialised code for atom P(...)
  Select(left &' body &' right,l,s,r) &
  Demo1(s,l,r,s1,s_out).
Demo1(Q(args),left,right,s_in,s_out) <-
  SimpleResolve*(args,body,s_in,s1) &     % Specialised code for atom Q(...)
  Select(left &' body &' right,l,s,r) &
  Demo1(s,l,r,s1,s_out).
```

Figure 3.9: Specialised Structure of Meta-Interpreters

We shall describe a more general application of this technique later. Secondly, we use the convention that in specialised programs we append an asterix to the name of a predicate in order to denote the replacing of an atom in the original program by the specialisation of this call. For example, in Figure 3.9 we use the atom `Preprocess*(query,answer)` to indicate the replacement of the call `Preprocess(program,query,answer)` in the original program by the (possibly empty) conjunction of formulas forming the specialisation of that call.

3.4.3 Avoiding Code Explosion: Structure of Residual Programs

In Figure 3.9 we have not sought to unfold calls to `Select` in the definitions of `Demo` and `Demo1` as, in general, we have insufficient knowledge of the arguments of such a call to be able to fully specialise it. To unfold the call to `Select` in these statements would be counter-productive as it would increase the amount of code and lead to less efficient residual code. Instead we have left these calls to `Select` in the residual code for `Demo` and `Demo1` and then specialised the definition of `Select` separately, for the general case. Similarly in `Demo`, while we would quite probably wish to at least partially specialise the calls to `Preprocess` and `Postprocess`, unfolding `Demo1` would again lead to an increase in the amount of (less efficient) residual code.

Figure 3.10 illustrates the result of unfolding the call to `Demo1` in the definition of `Demo` in Figure 3.9. Here, through over-specialisation, we have both increased the size of the residual code and made it less efficient, as `Preprocess*` and `Select` are potentially called twice (once in each

```
Demo(query,answer) <-
  Preprocess*(query,goal) &
  Select(goal,left,P(args),right) &
  SimpleResolve*(args,body,EmptySubst,subst1) &    % Resolve for P
  Select(left &' body &' right,l,s,r) &
  Demo1(s,l,r,subst1,subst_out) &
  Postprocess*(subst,answer).
Demo(query,answer) <-
  Preprocess*(query,goal) &
  Select(goal,left,Q(args),right) &
  SimpleResolve*(args,body,EmptySubst,subst1) &    % Resolve for Q
  Select(left &' body &' right,l,s,r) &
  Demo1(s,l,r,subst1,subst_out) &
  Postprocess*(subst,answer).
```

Figure 3.10: Unfolding of `Demo1` in `Demo`

statement) although, with the query instantiated, only one version will succeed. In adding an extra statement to the definition of `Demo` we have also introduced a superfluous choicepoint. It would have been preferable to leave the call to `Demo1` as a residual call in `Demo`, and have the specialised definition of `Demo1` separate.

Our justification for unfolding the calls to `Preprocess` and `Postprocess` in `Demo` as far as possible is as follows. Generally `Postprocess` will not be sufficiently instantiated to perform more than a minimal specialisation, but a closer examination should allow this without risking code explosion. `Preprocess`, in our experience, will either be sufficiently instantiated to be fully unfolded (or very nearly so, leaving at most a few residual calls) or, like `Postprocess`, only suitable for minimal specialisation. In either case `Preprocess` may be unfolded without a risk of code explosion in the new definition of `Demo`. We illustrate this issue more clearly when we consider the specialisation (by self-application) of the partial evaluator *SAGE* in Section 4.1.

Bearing the above issues in mind we see that ideally we would wish for the specialised meta-interpreter to display a structure very similar to that of the original program, where calls to `Preprocess`, `Postprocess` and `SimpleResolve` had been unfolded in the statements defining `Demo` and `Demo1`, but not calls to `Select` and `Demo1`. The definitions of `Select` and `Demo1` would then be replaced by specialised versions of more general calls with these predicates. In this way we avoid code-explosion, as we are effectively adding the object program to the meta-program, thus preventing an excessive increase in the amount of residual code.

Our reasoning for singling out `Demo1` and `Select` to be specialised separately, while permitting

```
SimpleResolve(atom, statement, body, subst_in, subst_out) <-
  IsImpliedBy(head, body, statement) &
  PredicateAtom(head, pred, args1) &
  PredicateAtom(atom, pred, args2) &
  UnifyArgs(args1, args2, subst_in, subst_out).

UnifyArgs([], [], subst, subst).

UnifyArgs([arg1|rest1], [arg2|rest2], subst_in, subst_out) <-
  UnifyTerms(arg1, arg2, subst_in, subst1) &
  UnifyArgs(rest1, rest2, subst1, subst_out).
```

Figure 3.11: Code for `SimpleResolve`

`SimpleResolve`, `Preprocess` and `Postprocess` to be unfolded within `Demo1` and `Demo`, is that `Demo1` and `Select` are both recursive *and* unable to have that recursion removed. Therefore it is better to retain them as independent (recursive) code, rather than embedding this recursion by unfolding calls to these atoms in the bodies of `Demo` and `Demo1` (as in the previous example). Embedding of recursion in this way would lead to code explosion. Thus we want to detect those predicates which may potentially be recursive in the residual code. Non-recursive predicates obviously will not be, nor will any recursive predicates which may be *selectively* unfolded until they terminate. Our strategy is to first detect all recursive predicates and then identify which of these can be guaranteed to terminate. All other recursive predicates are highlighted as being potentially recursive in the residual code.

Let us assume for the moment that a call to `UnifyTerms` may be specialised with a known first argument to return a single, determinate, residual formula (which will be a conjunction of atoms) in much the same manner as for `Resolve` in Section 3.3. Using this knowledge it can easily be seen that, for a known object statement, the code for `SimpleResolve` in Figure 3.11 may be specialised to produce a single, determinate, specialised formula. The recursive predicate `UnifyArgs` may be unfolded without fear of non-termination, as its first argument will always be ground.

Our technique has been to fully analyse the program and query being partially evaluated before commencing the partial evaluation. In this way we seek to find a set of predicates in the program that are potentially 'unsafe' to the partial evaluation, that is, those predicates whose indiscriminate selection might possibly lead to infinite unfolding. For example, when specialising the interpreter in Figure 3.8 with respect to a call to `Demo`, the predicates `Select` and `Demo1` would be unsafe.

In the partial evaluation, literals containing unsafe predicate symbols are not selected for

unfolding. Generalised calls to these predicates are then unfolded by the same strategy. That is, the predicates are unfolded without unfolding unsafe predicates. Our strategy is thus somewhat conservative. Following this method, the results of the partial evaluation will be a set of specialised calls to atoms in the query and of specialised definitions of all the unsafe predicates upon which the specialised calls in the query depend. The static analysis performed by *SAGE* to detect unsafe predicates is discussed in Chapter 4.

An alternative method to avoid code explosion is to enclose a non-determinate test in a conditional. For example, if C is some non-determinate test which causes a computation to compute T if C succeeds and E if C fails, then this may be expressed as `IF` C `THEN` T `ELSE` E. As described in the previous chapter, the specialisation of a formula of this form will be a single residual conditional formula in the case where C is not sufficiently instantiated to be fully unfolded.

3.4.4 Implementing a Selection Strategy

When specialising meta-programs we find that the program code falls into two more or less distinct categories. In the first of these categories we place code which is sufficiently instantiated to allow its unfolding and in the second category we place all code which must be left as residual code after specialisation. For the partial evaluation of meta-programs this first category will generally cover that code which manipulates the representation of the object program (which is known at the time of partial evaluation) and the second category will cover that code which manipulates the representations of goals to this object program (which will be unknown). This distinction has also been highlighted in [43] where the code defining interpreters was partitioned into what was referred to as *parsing* and *execution* instructions.

One important aspect of meta-programs which concerns the manipulations of object level goals is the selection strategy that a meta-program implements. It is generally not possible to perform more than a small amount of partial evaluation of such code. Normally we would not expect to be able to do more than remove the overheads imposed by Gödel's abstract data types for such code. This then is a very strong argument for insisting that when writing a meta-program particular care is taken to ensure that the implementation of the selection strategy should be as efficient as possible.

Consider the implementations of a simple 'leftmost literal' selection function in figures 3.3 and 3.12. Here we have assumed that the left conjunct of a conjunctive formula is never empty and that we will therefore always be able to select a literal from this formula. The simple meta-interpreter of Figure 3.3 in fact only supports the selection of positive and not negative literals, but this could easily be extended.

The major difference between the implementation of selection in figures 3.3 and 3.12 is that in the first the selection function is *implicit* in the code for the interpreter whereas in the second it is

```
Select(formula, left, selected, right) <-
  And(l, r, formula) &
  Select(l, left, selected, r1) &
  AndWithEmpty(r1, r, right).
Select(formula, empty, formula, empty) <-
  Literal(formula) &
  EmptyFormula(empty).
```

Figure 3.12: A Simple Leftmost Literal Selection Function

```
Select(formula, left, selected, right) <-
  And(l, r, formula) &
  IF SOME [l1,r1,s]  Select(l, l1, s, r1)
    THEN
      left = l1 &
      selected = s &
      AndWithEmpty(r1, r, right)
    ELSE
      Select(r, l1, selected, right) &
      AndWithEmpty(l, l1, left).
Select(formula, empty, formula, empty) <-
  Atom(formula) &
  EmptyFormula(empty).
```

Figure 3.13: A Sophisticated Leftmost Literal Selection Function

made *explicit* by an explicit definition. Comparing the execution times of the specialised version of the interpreter of Figure 3.3 with those for an interpreter which used the explicit selection function of Figure 3.12 we will generally find that the former is around three to four times faster. This gives us some indication of the expense inherent in an explicit selection function in the residual code for a meta-program.

The selection strategies employed by figures 3.3 and 3.12 are naturally very simple and in general we would require a more sophisticated implementation which allows for the possibility that we are unable to select a literal from the left conjunct of a conjunction. Figures 3.13 and 3.14 illustrate two potential implementations of a 'leftmost-literal' selection strategy which allow for this possibility.

The first of these implementations relies upon an implementation strategy which we refer to as *failure-driven*. It utilises an `IF-THEN-ELSE` conditional to implement an attempt to first select

```
Select(formula, formula, None, formula) <-
  EmptyFormula(formula).
Select(formula, left, selected, right) <-
  And(l, r, formula) &
  Select(l, l1, s, r1) &
  IF  s = None
    THEN  Select(r, left, selected, right)
    ELSE  left = l1 &
          selected = s &
          AndWithEmpty(r1, r, right).
Select(formula, empty, Positive(formula), empty) <-
  Atom(formula) &
  EmptyFormula(empty).
Select(formula, empty, Negative(atom), empty) <-
  Not(atom, formula) &
  EmptyFormula(empty).
```

Figure 3.14: An Efficient Leftmost Literal Selection Function

a literal from the left conjunct of a conjunction and, failing that, from the right conjunct. This implementation handles the failure to select a literal by failing to produce an answer.

By contrast to the failure-driven implementation of Figure 3.13, the selection strategy of Figure 3.14 relies upon a *success* or *forward-driven* strategy for its implementation. Rather than failing to compute an answer if a literal cannot be selected from a formula, this implementation computes an answer that *reports* the failure to select a literal by returning the constant `None` as the selected formula. This implementation requires that a new type be introduced for the terms that report success or failure of the selection strategy. This may be used to further effect in the return of a successfully selected literal, as the atom in a literal will be returned as `Positive(atom)` if it is a positive literal and `Negative(atom)` if it is a negative literal. Thus we may determine whether a selected literal is positive or negative without needing to examine the representation of that literal.

We assert here that the forward-driven style of programming is more efficient and, arguably, clearer than the failure-driven style. This is reflected in the comparison of execution times for the residual code of a specialised forward-driven implementation of the selection strategy as opposed to the specialised failure-driven implementation of the same strategy.

To summarise, we have identified two points to be considered when implementing Gödel meta-programs. The first, very general, point is that the forward-driven style of programming is cleaner and more efficient in both unspecialised and specialised programs than is the failure-driven style.

The second point, specific to meta-programs, is that the selection strategy is a major computational expense in specialised meta-programs and therefore the strategy should be as simple as possible and its implementation should be as efficient as possible.

3.4.5 Removing Redundant Terms

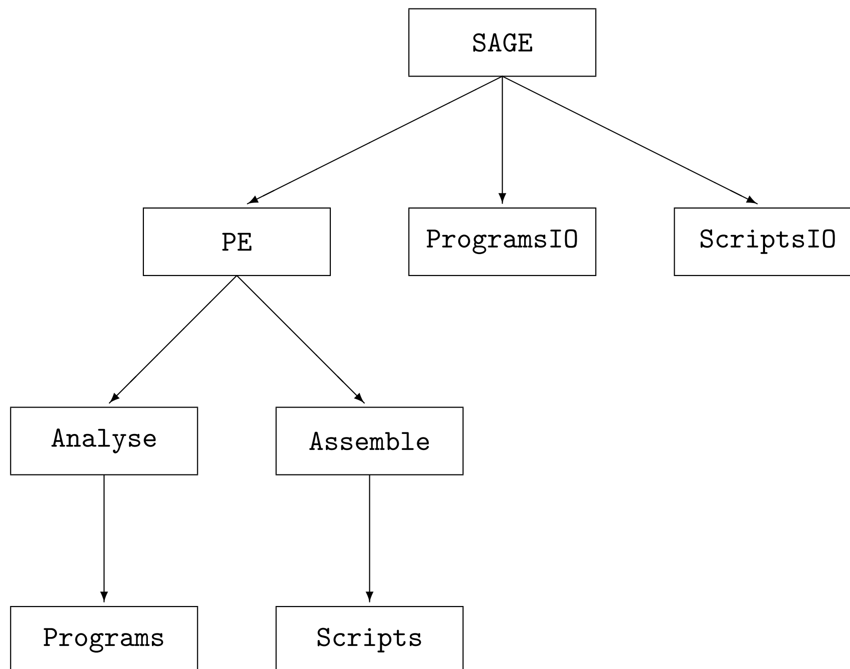
The introduction of redundant terms is often a major cause of inefficiency in any residual code. This problem can be particularly acute in specialising meta-programs which use a ground representation as such programs often process some very large terms (such as the representation of an object program). However, the majority of these terms can be removed through some fairly straightforward postprocessing optimisations. For example, in Figure 3.9 a call to `Demo(program,query,answer)` has been specialised with respect to some known program. Following specialisation the redundant term representing the object program was removed, leaving the new binary predicate `Demo`. Any subsequent call to `Demo(program,query,answer)` is replaced by a call to `Demo(query,answer)`. The implementation of these optimisations is described in more detail in Section 4.4. Similar optimisations have been described in [22].

Chapter 4

The Anatomy of SAGE

In this chapter we examine how the partial evaluation techniques described in the previous chapter are utilised by *SAGE* for the partial evaluation of Gödel programs.

SAGE is comprised of four modules, `SAGE`, `PE`, `Analyse` and `Assemble`. The figure below shows the relationships between these four modules and the Gödel system modules that they each import. An arrow from one module to another means the first module refers to the second.



The module `SAGE` is the smallest of these modules and defines the user-interface for *SAGE*. `SAGE` is the only module in *SAGE* which uses Gödel's input/output facilities. The module `PE` defines the code which computes partial evaluations. `PE` imports the two subsidiary modules `Analyse`, which defines the code that computes *SAGE*'s static analyses, and `Assemble`, which defines the code that

performs the post-processing optimisations and assembles the residual script.

Our description of *SAGE* breaks neatly into three parts which correspond to the three modules `Analyse`, `PE` and `Assemble`. In Section 4.2 we describe in more detail the static analysis which guides *SAGE*'s selection strategy. In Section 4.3 we provide details of the implementation of the unfolding strategy employed by *SAGE*. In Section 4.4 we describe the postprocessing optimisations performed by *SAGE* and how these results are incorporated into the assembly of the residual Gödel script. Finally in Section 4.5 we outline the proofs of the termination and correctness of *SAGE*.

Before we examine *SAGE* in detail we first give an overview of the issues in constructing an effectively self-applicable partial evaluator, thus providing a motivation for the anatomy of *SAGE*.

4.1 Constructing a Self-Applicable Partial Evaluator

The *SAGE* system introduced in this thesis is a partial evaluator for *full* Gödel. That is to say, it has been designed to partially evaluate *any* program in the Gödel language, although particular emphasis has been placed upon the specialisation of meta-programs. *SAGE* is therefore self-applicable. That is, it is capable of specialising itself to produce *efficient* residual code. Specialising full Prolog and the construction of an effective self-applicable Prolog partial evaluator are tasks that are made most difficult by the so called 'impure' features of Prolog. This is illustrated by the sophistication needed to specialise these features [59, 72]. By an 'effective' self-applicable partial evaluator, we mean one capable of producing efficient residual code upon self-application.

In order for a partial evaluator to be self-applicable it needs to be a relatively sophisticated, and fairly complex, program. In order to produce efficient code upon self-application the partial evaluator needs to be able to produce relatively simple residual code. Consequently, the greatest difficulty in achieving an effective, self-applicable partial evaluator, is that the more complex that the partial evaluator becomes, the harder it is to specialise, and thus the less efficient the residual code is likely to be. What is needed is for the partial evaluator to be as simple and straightforward as possible and to employ no, or at least very few, excessively complex operations. It is for this reason that previous attempts at self-applicability in Prolog partial evaluators have generally considered only the unfolding of a pure subset of the language [18, 19, 51].

There are two major reasons for the complexity of most current full-language partial evaluators, these being the ability to specialise the more complex facilities of the full language and the selection strategy employed in computing a partial evaluation. We have overcome the first of these obstacles by implementing *SAGE* in Gödel, which has few non-logical (and therefore hard to specialise) features. Next we show how *SAGE* may be described in terms of the framework of Section 3.4 and how the specialisation of this framework, coupled with *SAGE*'s use of static analysis to guide the selection strategy, enables *SAGE* to produce efficient code upon self-application. We claim

Prolog's failure to provide adequate meta-programming facilities, particularly a suitable ground meta-representation, coupled with its impure features, seem likely to prevent an effective self-applicable partial evaluator for full Prolog from ever being built.

4.1.1 Specialising a Self-Applicable Partial Evaluator

As has been stated, there are two criteria which an effectively self-applicable partial evaluator must meet. The first is that it must be able to specialise all of the facilities provided by the language in which it is implemented. The second is that it must produce specialised code which executes in a significantly reduced time. In chapters 2 and 3 we have presented the necessary techniques for specialising the various facilities of the Gödel language. In this section we shall describe the factors that allow a Gödel partial evaluator to produce efficient residual code.

The framework for meta-interpreters, described in Section 3.4, can be extended to encompass a partial evaluator and likewise the specialisation of such interpreters corresponds to the self-application of such a partial evaluator. We have already identified the two major areas for expense in such meta-programs as being their use of the ground representation and the potential expense of a sophisticated selection strategy.

The self-application of a partial evaluator requires the specialisation of that partial evaluator to a known object program. This object program will be a meta-interpreter in the case of the second Futamura projection and the partial evaluator itself in the case of the third Futamura projection. We have already seen in Section 3.3 how the expense of resolution, the most expensive aspect of the ground representation, may be largely eliminated for these cases.

The majority of selection strategies proposed for partial evaluation, such as those in [3, 5, 10, 57, 74], involve a certain amount of dynamic ancestor-checking. The strategies range from setting an *ad hoc* depth bound upon the amount of unfolding which is permitted to some extremely sophisticated dynamic analysis based upon some well-founded ordering relations for atoms in the partial evaluation. The most sophisticated selection strategies may often lead to better results, as the results of a partial evaluation are largely determined by the selection strategy employed.

The drawback of employing a most sophisticated strategy is the expense, as operations such as ancestor-checking can become immensely expensive for large partial evaluations. This expense becomes an even more serious issue when we consider the self-application of a partial evaluator. As we have already shown in Section 3.4.4, the dynamic aspect of the selection function employed by any meta-program cannot be specialised to a large degree and can therefore be a significant expense in the specialised code. As we wish to specialise the partial evaluator itself we should therefore seek to reduce the dynamic aspect of its selection strategy as far as possible.

Next we describe a general framework in which we may describe the selection strategy for any given partial evaluator. In terms of this framework we shall then discuss which kinds of selection

strategies are most amenable to specialisation.

4.1.2 A Framework for Selection in Partial Evaluation

We propose a general framework for the selection strategy of a partial evaluator which is a program conforming to the framework of Figure 3.8. This framework for implementing selection allows for any combination of static and dynamic analysis in the selection strategy. We shall specify conditions under which this framework will guarantee the termination of partial evaluation and in Section 4.2 we describe *SAGE*'s selection strategy in terms of this framework and prove that it meets the necessary conditions for termination.

Our framework is intended to be used to define the selection strategy for any partial evaluator. It is based on the assumption that we may divide the selection strategy into a static and dynamic part. The static part is performed before the partial evaluation takes place and is an analysis of the program and query to be partially evaluated which deduces information that may be used to guide the dynamic aspect. The dynamic aspect of the selection strategy is implemented as a selection function of the kind described in Section 3.4.4.

First we define the basic components of a general selection strategy. Using these we present algorithms for the static and dynamic parts of a generic selection strategy for partial evaluation. Partial evaluations may be computed with respect to a set of predicate symbols which it is permitted to unfold. This set of predicates will often be specified by the user. As in [3], we refer to this as the set of *selectable* predicates.

Definition Let L be a set of predicate symbols. We say a literal is *L-selectable* if its predicate symbol is in L. We say an SLDNF-tree is *L-compatible* if the predicate symbol of each selected literal in the tree (including subsidiary refutations and trees) is in L.

In order to guarantee the correctness of a partial evaluation wrt a set of selectable predicates L we require the following weak condition.

Definition Let P be a normal program and L a set of predicate symbols. We say L is *well-structured* wrt P if, whenever $p \in L$ and there exists a path in the dependency graph for P from a predicate symbol q to p, then $q \in P$.

We assume that any selection strategy must have an underlying rule which specifies the order in which literals in a resultant may be tested for finite unfolding.

Definition A *fundamental selection strategy* is a determinate rule which uniquely selects a literal from some resultant.

A fundamental selection strategy may be a simple 'leftmost-literal' rule, or may use sophisticated determinancy or variable dependency analysis, for example. We use the concept of a fundamental

selection strategy to define the selection strategy used when termination is not an issue.

Definition Let P be a program, G a goal for this program and T a finite SLDNF-tree for G wrt P. A predicate symbol p is *safe* wrt T iff no literals with predicate symbol p appear in any leaf-node of T. We say a predicate which is not safe is an *unsafe* predicate.

Intuitively, we say a predicate symbol p is *safe* in a partial evaluation if every occurrence of a literal with predicate symbol p may be unfolded in this partial evaluation without risk of an infinite unfolding. For example, non-recursive predicates are trivially safe.

Definition Let P be a program, $\mathbf{A}$ a set of atoms, T the SLDNF-tree used in the partial evaluation of $\mathbf{A}$ wrt P, p a predicate symbol in P and $\mathbf{C}$ an independent set of atoms with predicate symbol p. We say that $\mathbf{C}$ is a *covering atom* for p wrt the partial evaluation of $\mathbf{A}$ iff every atom in a node of T with predicate symbol p is an instance of an atom in $\mathbf{C}$.

Finally we define informally an *information-set* for a predicate symbol. An information-set for a predicate symbol p is any data which is used by a dynamic selection strategy in determining whether it is safe to unfold an atom with predicate symbol p. For example, an ancestor list or the finite prefoundings of [10] would constitute an information-set for a predicate symbol.

With the above definitions, we present the static part of our generic selection strategy in Figure 4.1. The static part of a selection strategy computes an abstraction of the partial evaluation to be performed. Based on this abstraction the static part determines a subset of the selectable predicates which will be safe in the subsequent partial evaluation. In addition the static analysis returns a set of information-sets and covering atoms for the unsafe selectable atoms.

Having performed the static part of the selection strategy, a partial evaluator may proceed to compute a partial evaluation, using the dynamic part of the selection strategy. The algorithm for the dynamic part of our generic selection strategy is given by Figure 4.2. The dynamic part of the selection strategy does not restrict the selection of atoms with safe predicates (unless the fundamental selection strategy imposes a restriction) and uses the information-sets computed for the unsafe predicates to determine whether atoms with these predicates may be selected.

In order to guarantee the termination of such a selection strategy we must ensure that it meets the following conditions:

1. The static analysis must terminate.
2. The static analysis is correct wrt the subsequent partial evaluation. That is to say:
   - Unfolding all instances of atoms with predicates in the safe set will not lead to infinite unfolding in this partial evaluation.
   - The covering atoms computed for the unsafe predicates are sufficient to cover all instances of atoms with these predicates in this partial evaluation.

Input:

a program P

a goal G

a set $Preds$ of selectable predicate symbols

a fundamental selection strategy $Select$

Output:

a safe subset of $Preds$, $Safe$

a set of information-sets I for predicates in $Preds - Safe$

a set of covering atoms for atoms with predicates in $Preds - Safe$

Initialisation

$I := \emptyset$

$Safe := Preds$

$A := \emptyset$

$Fix := Preds$

While $Fix \neq Preds - Safe$ **do**

$Fix := Preds - Safe$

compute abstract derivation tree T for $\leftarrow G$, based on strategy $Select$, information-set I and set of safe predicate symbols $Safe$

$Unsafe := \{\ p : p$ deemed unsafe in $T\ \}$

$Safe := Safe - Unsafe$

update I on basis of information-set gained from T

$I := I \cup I_{Unsafe}$, where $I_{Unsafe} = \{$ information-set gained in T for $p : p \in Unsafe\ \}$

$A := \{$ covering atoms for p in $T : p \in Unsafe\ \}$

EndWhile

Figure 4.1: Static Selection Algorithm

Input:

a set $Preds$ of selectable predicate symbols

a subset of $Preds$, $Safe$, of safe predicate symbols

a set of information-sets $\{ I_p : p \in Preds - Safe \}$

a fundamental selection strategy $Select$

a resultant R

Output:

a selected literal A in R

I_{New}, an update of the input set of information-sets

Initialisation

$I^* :=$ input set of information-sets

$Success := No$

While $Success = No$ and there exist unexamined literals in the body of R **do**

according to strategy $Select$, select a $Preds$-selectable literal, A^*, from the body of R

update I^* on basis of information-set gained during selection of A^*

If (the predicate symbol of $A^* \in Safe$ **or** A^* is selectable according to I^*)

Then

$Success := Yes$

$A := A^*$

$I_{New} := I^*$

EndWhile

Figure 4.2: Dynamic Selection Algorithm

3. The dynamic analysis must control the unfolding of atoms with an unsafe predicate name in a manner which guarantees that the unfolding of such atoms will terminate.

4.1.3 Specialising the Selection Strategy

Clearly the above framework is sufficiently general to describe a broad spectrum of selection strategies. Towards one end of this spectrum we have a mostly dynamic strategy, such as that of [10] where the static analysis at most identifies non-recursive predicates as being safe and performs a complex dynamic analysis during partial evaluation that is based upon a well-founded ordering of the recursive atoms in the partial evaluation. Towards the opposite end of the spectrum we have a selection strategy which performs a sophisticated static analysis of the program and query to be specialised and uses this information to restrict the complexity of the dynamic analysis as far as possible.

For self-applicability a selection strategy of the former kind is not satisfactory. In specialising a partial evaluator we would wish to specialise the selection strategy as far as possible. As we have already described in Section 3.4.4, the dynamic part of any selection strategy cannot be specialised to any great degree as it relies upon a dynamic analysis of the resultant from which it is attempting to select a literal and the computation in which it appears. This information is not available at the time of partial evaluation. However the static part of a selection strategy relies mostly on a static analysis of the program and query to be partially evaluated. At partial evaluation time we will certainly know the program we are specialising the partial evaluator with respect to and we expect to have at least some partial information about the query. Thus the static part of a selection strategy can be specialised at least to a significant degree, if not fully.

To construct a selection strategy for a partial evaluator which is amenable to specialisation we must therefore seek to produce an implementation of the selection strategy which performs the greater part of its analysis as a static analysis. Specialising such a selection strategy for a known program we will remove the majority of this static analysis. This will leave us with a residual dynamic selection strategy which has been specialised with respect to the known program and is therefore no more sophisticated than is necessary. This in turn implies that the specialised selection strategy will be no more computationally expensive than is necessary.

4.2 The Selection Strategy for *SAGE*

The selection strategy used by *SAGE* employs a relatively sophisticated static analysis of the program and query to determine as large a subset as possible of the selectable predicates which is safe in the subsequent partial evaluation. With such an analysis we are able to avoid entirely any need for dynamic analysis during partial evaluation. The major advantage of this technique is

```
SAGE <-
        InputProgram(program) &
        InputGoal(goal) &
        InputSelectablePredicates(select) &
        GetSelectableAtomsInGoal(goal, select, atoms) &
        PartialEvaluation(program, atoms, select, script) &
        OutputScript(script).

Partial Evaluation(program, atoms, select, script) <-
 % Preprocessing.
        MakeAtomsIndependent(atoms, atoms1) &
        StaticAnalysis(program, atoms1, select, atoms2, danger) &
 % Interpretation (PE).
        ComputePEs(program, atoms2, select, danger, residuals) &
 % Postprocessing.
        AssembleScript(program, residuals, script).
```

Figure 4.3: Pseudo-code for top level of *SAGE*

evident in the self-application of *SAGE*, where we are able to specialise the static analysis part to a large extent. The expense of the dynamic analysis, which cannot be significantly specialised, we have effectively reduced to nothing.

Figure 4.3 gives a pseudo-code outline for the top-level structure of *SAGE*, in the framework of Figure 3.8. A more complete description of this algorithm is given in Figure 4.9. In this framework `Preprocess` corresponds to the static analysis performed by *SAGE*, `Postprocess` to the postprocessing optimisations of Section 4.4 and the interpretation performed by `Demo1` corresponds to the partial evaluation of atoms in the query to be specialised. This partial evaluation is itself closely related to an interpretation and is therefore a subprogram in *SAGE* which can also be described in terms of the framework of Figure 3.8. The process of computing partial evaluations in *SAGE* may therefore be specialised by *SAGE* in a similar fashion to the specialisation of the program in Figure 3.8.

Here we see the advantage for self-applicability of using static analysis as a manner of providing information that may be used to reduce the dynamic aspect of a partial evaluators selection strategy for a particular program. As `Preprocess` may be specialised in Figure 3.8, so may the static analysis performed by a partial evaluator be specialised to a very large degree when specialising that partial evaluator with respect to a particular program.

In the following sections we shall firstly present an informal overview of *SAGE*'s static analy-

sis. We follow this by a more formal presentation which includes a proof of the correctness and termination of *SAGE*'s selection strategy and finally we discuss the justification for this particular strategy.

4.2.1 Overview of Static Selection Strategy for *SAGE*

In partial evaluation we are increasingly seeing the use of abstract interpretation [1, 14] to provide flow information to guide a partial evaluation [24, 23, 74] either through a static or dynamic analysis of the program and query to be specialised. The static analysis performed by *SAGE* uses techniques similar to those of abstract interpretation to compute as large as possible a set of predicates which is safe in the partial evaluation to be performed. An abstraction of the tree used in computing the partial evaluation is computed and analysed to determine a set of predicates which may be unfolded while still guaranteeing that the partial evaluation will terminate. We refer to such an abstract tree as an *abstract partial evaluation*. Our description of this static analysis closely reflects its implementation by *SAGE*. At present this implementation cannot be presented as an abstract interpretation in the usual sense. Consequently we provide below a self-contained description of this analysis.

When constructing an abstract partial evaluation tree we must ensure that the selection strategy employed is equivalent to the strategy that will be employed by the subsequent partial evaluation. For the abstract partial evaluation to be a correct abstraction of the subsequent concrete partial evaluation we must ensure that the set of safe predicates is equivalent for both. The static analysis will repeatedly compute abstract partial evaluations with respect to the most recently computed set of safe predicates until an abstract partial evaluation is computed during which the set of safe predicates remains constant. This ensures that the set of safe predicates remains constant throughout the abstract partial evaluation which abstract the subsequent concrete one. Figure 4.4 illustrates this process and is the top-level algorithm for *SAGE*'s static analysis.

Our abstract domain consists of abstractions for the set of all ground terms and the set of all terms. Using this abstract domain we construct an abstract partial evaluation in which by not unfolding certain recursive predicates we guarantee the termination of all partial evaluations that are concretisations of this abstract one.

For example, consider the following predicate definition:

```
Squares([],[]).
Squares([x|l1],[x*x|l2]) <-
  Squares(l1,l2).
```

The predicate `Squares` is recursive and thus if every atom with this predicate were unfolded in a partial evaluation then the partial evaluation might not terminate. However, if we consider the

Input:

a program P

a set $Preds$ of selectable predicate symbols

an abstract goal G

Output:

a set of unsafe predicate symbols $Unsafe$ ($Unsafe \subseteq Preds$)

a set of covering atoms $Covers$

Initialisation

$\mathbf{A} := \{ A : A$ is a $Preds-$selectable atom in $G \}$

$Recurs := \{ p : p \in Preds$ and p is recursive in $\mathbf{A}$ wrt $P \}$

$Unsafe := \emptyset$

$Covers := \emptyset$

$Patterns := \emptyset$

$Fix := \emptyset$

compute abstract partial evaluation tree T for atoms in $\mathbf{A}$

returning updated values for $Unsafe$, $Patterns$ and $Covers$

While $Fix \neq Unsafe$ **do**

$Fix := Unsafe$

compute abstract partial evaluation tree T for atoms in $\mathbf{A}$

returning updated values for $Unsafe$, $Patterns$ and $Covers$

EndWhile

Figure 4.4: Static Analysis Procedure

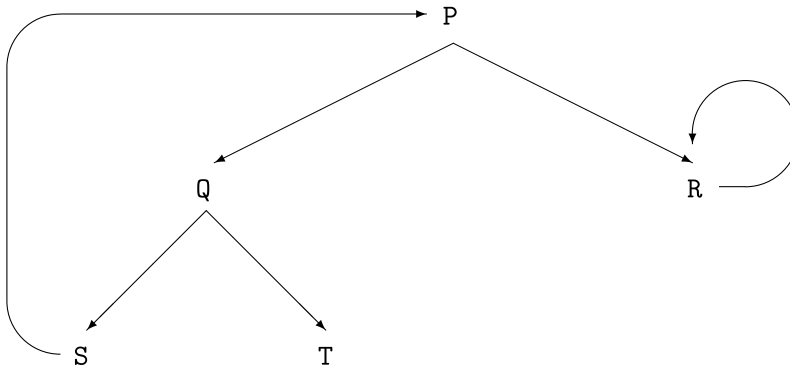


Figure 4.5: Predicate Dependency Graph

abstract partial evaluation of the atom `Squares`($\mathcal{T}_1, \mathcal{T}_2$), where $\mathcal{T}_1$ is some ground term, then it is obvious that the repeated unfolding of atoms with this predicate will eventually terminate. In the case where $\mathcal{T}_1$ is an abstraction for the set of all ground terms in the above program then this analysis tells us that the partial evaluation of any atom with this predicate will terminate whenever the first argument is a ground term.

To identify recursive predicates in a program we compute a predicate-dependency graph for the selectable predicates in that program. Recursive predicates may be recursive because they call themselves directly or because they occur in a cycle in the predicate dependency graph. For a cycle in a predicate dependency graph we need only to denote only one of the predicates as being recursive when considering termination, as the example below illustrates. For such a set of predicates we denote an arbitrary one of them as recursive and the rest as non-recursive.

Example The predicate dependencies for $\mathcal{P} \cup \{\leftarrow \texttt{P}\}$, where $\mathcal{P}$ is:

```
P <- Q & R.
Q <- S & T.
R <- R.
S <- P.
T.
```

are illustrated by the graph in Figure 4.5. In this graph we have the cycle [P,Q,S] and the recursive predicate R. We may denote P as being the recursive predicate in the cycle, giving us {P,R} as the set of recursive predicates for $\mathcal{P} \cup \{\leftarrow \texttt{P}\}$. If we denoted both of these predicates as not being selectable, then it is obvious that the partial evaluation of any literal in the above program will terminate.

4.2.2 Abstract Partial Evaluations

The abstract domain consists of the three elements {⊥,`AG`,`ANG`} which are abstractions of the sets of undefined terms, all ground terms and all terms respectively. We refer to the elements of this abstract domain as *pattern terms*. The pattern terms are ordered as $\perp \leq$ `AG` $\leq$ `ANG`.

In the implementation of the static analysis we supplement this abstract domain by a set of data structures which denote a countable set of variables represented by the terms `V(0)`,`V(1)`,..., constant and function terms represented as `F(name,args)` where `name` is the name of the function or constant and `args` is the list of its abstract arguments, a countable set of abstract ground terms represented as `GT(0)`, `GT(1)`,... and a well-founded ordering on the set of abstract ground terms. We refer to the representation of a term constructed from these data structures as an *abstract term* and to the representation of an atom with arguments taken from the set of abstract terms as an *abstract atom*.

Definition A *variable-free* abstract term is an abstract term of the form `GT(g)` or a term of the form `F(f,a)` where the terms in the list `a` contained no variables.

Abstract substitutions consist of a set of variable to abstract term bindings. Unification of abstract terms is according to the following rules:

- Variable to term (or similarly, term to variable): variable is bound to term provided that the variable does not occur in this term, in which case the unification fails.

- Function term `F(n1,a1)` to function term `F(n2,a2)`: if `n1`=`n2` then unify the two argument lists `a1` and `a2`, else the unification fails.

- Function term `F(n,a)` to ground term `GT(g)`: let V be the set of variables which occur in terms in the list `a`. For each variable in V add to the current substitution a binding of this variable to some new abstract ground term and record in the ordering on abstract ground terms that this new ground term is strictly less than `GT(g)`.

- Ground term `GT(g1)` to ground term `GT(g2)`: record `GT(g1)`=`GT(g2)` in the ordering on abstract ground terms provided that this does not violate the anti-symmetry of the ordering, in which case the unification fails.

In the abstract resolution procedure we resolve an atom with abstract arguments with respect to a statement in the program which has a matching predicate symbol and some currently computed abstract substitution. This resolution process is implemented in a similar manner to that for concrete resolution, described in the previous chapter. However, for the sake of conciseness we may consider the abstract resolution process to be as follows:

1. Transform statement to abstract representation. This involves replacing all variables by renamed variables of the form `V(n)` and replacing all constant and function terms with name `name` and arguments `a` by abstract terms of the form `F(name,a1)` where `a1` is the transformed list of terms `a`.

2. Perform the abstract unification of the arguments of the atom with the arguments of the head of the transformed statement and compose the resulting substitution with the current abstract substitution.

3. Replace the abstract atom in the current goal by the body of the transformed statement.

Definition Let $H \leftarrow A_1\& < \ldots \& A_i \& \ldots \& A_n$ be a resultant, $B \leftarrow B_1 \& \ldots \& B_m$ a statement and θ the mgu of A_i and B then for each $B_r\theta$ in $(H \leftarrow A_1\& < \ldots \& B_1 \& \ldots \& B_m \& \ldots \& A_n)$ we refer to A_i as the *immediate parent* of $B_r\theta$. If two atoms A and B are related by the transitive closure of the relation immediate parent then we say that A is a *direct ancestor* of B.

To ensure that the computation of the abstract partial evaluation will itself terminate we adopt the following strategy for unfolding abstract atoms which have a recursive predicate. We do not unfold abstract atoms which have a direct ancestor with the same predicate. Instead we compare the atom with its ancestor in order to determine whether the abstract partial evaluation rooted at the ancestor atom could be guaranteed to terminate if all atoms with this predicate were unfolded. If this abstract partial evaluation can be guaranteed to terminate under these conditions then mark this predicate as being safe, otherwise it is marked as being unsafe. Note that if at any point a predicate is marked as being unsafe then it cannot subsequently be marked as safe.

Abstract Partial Evaluations: Definitions

Definition Each abstract partial evaluation is rooted at a single atom which we refer to as the *head atom*.

Definition Let A be an atom with recursive predicate p and A' an atom with predicate p that appears in the non-root node of an abstract partial evaluation rooted at A. We say that A' is a *recursive occurrence* of A.

We include here also an informal definition of the ordering relation between terms in the abstract domain, the formal definition being introduced later. Let t_1 and t_2 be abstract terms then we say that t_1 is *ordered* wrt t_2, or t_1 is *less than* t_2 if, for every concretisation t_1' and t_2' of t_1 and t_2, t_1' and t_2' are ground terms and the number of function and constant symbols in t_1' is strictly less than the number in t_2'.

The fundamental selection strategy used by *SAGE* is a depth-first strategy where we select the leftmost selectable literal. This fundamental selection strategy is therefore also used when constructing abstract partial evaluations.

Definition Let A be an abstract atom, T the abstract abstract partial evaluation rooted at A and S the set of computed abstract substitutions for the leaf nodes on successful branches of T. We call the set S the *complete unfolding* of A.

One example of a complete unfolding which will be of interest later is the case in which A is the leftmost selectable literal in a formula F, R is the abstract formula to the right of A in F and $S = \{\theta_1, \ldots, \theta_n\}$ is the complete unfolding of A. Then the complete unfolding of A in F will be the set of abstract formulas $\{R\theta_1, \ldots, R\theta_n\}$.

With the above definitions we may now present the algorithm for *SAGE*'s static analysis. We first present a simplified algorithm for the computation of abstract partial evaluations. To motivate the algorithm for the static analysis we shall then highlight the constraints that must be added to ensure the correctness of the static analysis and also describe some optimisations in its implementation. This allows us to present the full algorithm for computing abstract partial evaluations and to describe how this is incorporated into the static analysis.

Abstract Partial Evaluations: Algorithm

An abstract partial evaluation is the abstract partial derivation tree computed from an initial abstract atom A, a set of selectable predicate symbols L and a subset of L which we refer to as the safe set S. Note that in the course of computing the abstract partial evaluation some predicates may be removed from the set S if they are deemed to be unsafe. An ordering on the set of abstract ground terms is recorded for each node in the tree. At the root node this ordering is empty.

The tree is constructed downwards from the root by the selection and unfolding of atoms in resultants in the nodes of the tree. At a leaf node of the partly constructed tree we select the leftmost selectable literal and construct the next part of the abstract tree according to the following conditions upon the selected literal:

- A literal with a safe recursive predicate which does not match the predicate in the head atom. Compute an abstract partial evaluation rooted at the atom in this selected literal. Delete this literal from the resultant in this node.

- A literal with a safe recursive predicate which matches the predicate in the head atom. Examine this literal with respect to the head atom to determine whether it is safe. Delete this literal from the resultant in this node.

- An unsafe literal.
  Delete this literal from the resultant in this node.

- A literal with a non-recursive predicate.
  Unfold this literal in the current resultant.

To determine whether an atom in a selected literal which is a recursive occurrence of the head atom is safe we examine the arguments of the recursive occurrence with respect to the head atom. We seek to determine whether, for some argument position, the corresponding terms in the head atom and recursive occurrence are both ground and that the term in the recursive occurrence is ordered with respect to the corresponding term in the head atom. If this is not the case then we denote this predicate as being unsafe.

Assume that in the above examination we determine that a predicate is safe (or, to be more precise, is not yet determined to be unsafe). The literal with this predicate is next deleted from the goal from which it has been selected. This will leave the formula, F, to the right of this literal to be further unfolded. This indicates that we are next considering the case where we have performed the complete unfolding of the selected literal. However, it is possible that during this complete unfolding we may have instantiated some of the variables in F to ground terms and we would wish to ensure that such instantiations are not lost. Consequently when we determine that a literal with a recursive predicate is safe we also seek to determine which, if any, of the variables in this literal will have been bound to ground terms following its complete unfolding.

Example We analyse the unfolding of a call to `Squares(`$\mathcal{T}_1, \mathcal{T}_2$`)`, where $\mathcal{T}_1$ is a ground term, with respect to the program:

```
Squares([],[]).
Squares([x|l1],[x*x|l2]) <-
  Squares(l1,l2).
```

In the first (base) case, the second argument is instantiated to a ground term. In the second (recursive) case, the second argument is instantiated to a term which contains only variables which also occur in the recursive call (`l2`, in this case). By induction we deduce that the second argument in the recursive call will be instantiated to a ground term and so therefore will the second argument in the original call.

Correctness of Algorithm

The above description tells us how abstract partial evaluations may be computed. Before we describe in more detail how the above algorithm may be implemented we first describe three important issues which must be considered for the static analysis to be correct.

Firstly note in the above description that when a literal with a recursive predicate is selected the way in which it is treated if the predicate is unsafe will differ from the way it is treated if the predicate is safe. This reflects a similar difference in the treatment of literals with recursive predicates in the computation of the subsequent concrete partial evaluation. However, when the abstract partial evaluation is computed it is possible for a predicate which was marked as safe at the start of the computation to have been marked as unsafe during the computation. To ensure that the computed abstract partial evaluation is a correct abstraction of the subsequent concrete partial evaluation we must ensure that the set of safe predicates is the same throughout the computation of the abstract partial evaluation. Thus whenever the set of safe predicates is reduced while computing the abstract partial evaluation we will compute a new abstract partial evaluation with respect to this new set of safe predicates. As the set of recursive atoms in a program is finite and predicates marked as unsafe cannot subsequently be marked as safe the eventual termination of this process is guaranteed.

A second issue which must be considered is what will be done with the predicates which are determined to be unsafe. In the subsequent concrete partial evaluation literals with unsafe predicates will not be selected for unfolding as they may lead to non-termination. However, during the computation of the abstract partial evaluation we identify a generalisation of all instances of atoms with unsafe predicate symbols. We refer to such an atom as a *covering atom* for an unsafe predicate and a specialisation of each covering atom is computed during the concrete partial evaluation. Thus the static analysis must ensure that a correct abstract partial evaluation of each covering atom is computed.

The final issue that must be considered is under what conditions our ordering relation between an atom and its recursive occurrence can be considered correct. That is to say, what is the formal definition which we must give to this ordering relation that will ensure that we are correct to assume that it implies the termination of the repeated unfolding of 'ordered' literals. We introduce these definitions below and then present the complete algorithm for *SAGE*'s static analysis.

4.2.3 Computing Abstract Call Patterns

We next introduce the concept of a *pattern* for a recursive predicate. A pattern is a pair consisting of an *input pattern* and an *output pattern*. The input and output patterns are each abstract atoms whose arguments are taken from the abstract domain of pattern terms. Patterns are used as an abstract representation in which the input pattern records the state (either ground or non-ground) of the arguments of some atom before its unfolding in the subsequent concrete partial evaluation and the output pattern records the state of those arguments after its complete unfolding. For conciseness of representation, for a given predicate we may represent the input and output patterns as the list of pattern terms corresponding to the arguments of the relevant abstract atoms.

For example , let A be a binary abstract atom with output pattern `[AG,ANG]`, S be the set of abstract substitutions forming the complete unfolding of A and V be the set of variables occurring in the first argument of A. Then the fact that the first argument of the output pattern for A is the pattern term `AG` indicates that for any substitution in the set S all variables in V will be bound to variable-free terms.

Just as the domain of our abstract partial evaluation theoretically has abstraction and concretisation functions which map terms in the concrete domain to terms in the abstract domain and *vice versa*, we define similar functions for mapping between the domain of abstract partial evaluations and the domain of patterns. We refer to the abstraction and concretisation functions between these two abstract domains as *matching* and *applying* respectively.

Definition Let A be an abstract atom with recursive predicate p, $\mathcal{P}$ a pattern for p. We say that A *matches* $\mathcal{P}$ iff for each argument of the input pattern for $\mathcal{P}$ that is marked `AG`, the corresponding argument of A is a variable-free abstract term.

Definition Let A be an abstract atom with recursive predicate p, $\mathcal{P}$ a pattern for p. Then A', the result of *applying* the pattern $\mathcal{P}$ to A, is the atom obtained by unifying each non-ground argument of A for which the corresponding argument in the output pattern for $\mathcal{P}$ is `AG`, with some new abstract term of the form `GT(g)`.

For example, let `P` be a binary recursive predicate with input pattern `[AG,ANG]` and output pattern `[AG,AG]`. An abstract atom `P(`$\mathcal{T}_1$`,`$\mathcal{T}_2$`)` would match this pattern iff $\mathcal{T}_1$ were a variable-free abstract term. The result of applying the output pattern for this predicate to the abstract atom `P(GT(0),F(f',[x,y]))` is the atom `P(GT(0),(F(f',[GT(g),GT(g+1)]))`, where `GT(g)` and `GT(g+1)` are new (abstract) ground terms.

The ordering relation which we use to determine the safeness of some predicate is defined in terms of the input pattern for that predicate. First we define what it means for one abstract term to be a strict subterm of another.

Definition Let t_1 and t_2 be variable-free abstract terms then we say that t_1 is a *strict subterm* of t_2 iff

- **either** t_2 =`F(f,a)` and either t_1 is a member of list `a` or is a strict subterm of a member of list `a`
- **or** t_2 =`GT(g2)`, t_1 =`GT(g1)` and `GT(g1)` is strictly less than `GT(g2)` in the ordering on abstract ground terms.

Definition Let t be a term and a a pattern term then we say that a term t is *ordered* with respect to a term t' and the pattern term a iff $a =$ `AG` and t is a strict subterm of t'.

Definition Let A be an atom with recursive predicate p, $\mathcal{P}$ a pattern for p, $I_{\mathcal{P}}$ the input pattern for $\mathcal{P}$ and A' a recursive occurrence of A. We say that A' is ordered wrt A and $\mathcal{P}$ iff for some argument t' of A', t' is ordered wrt the corresponding arguments of A and $I_{\mathcal{P}}$.

Patterns allow us to make a further abstraction in the computation of our abstract partial evaluations. When computing an abstract partial evaluation which is rooted in some atom with a recursive predicate we will compute a further abstraction of the arguments of this atom into terms from the domain {`AG`,`ANG`} and a new "most general" abstract atom which matches this pattern. We then compute the abstract partial evaluation of this most abstract atom and determine which, if any, of its arguments are guaranteed to have become instantiated to ground terms. This allows us to construct the output pattern for this most general atom. The advantage of this technique is that when we subsequently encounter an atom with this predicate while computing the abstract partial evaluation, we may test to see if this subsequent atom is an instance of the previously computed pattern. If it is then we do not need to compute the abstract partial evaluation rooted at this new atom but can simply 'apply' the output pattern to the arguments of this atom.

4.2.4 Static Analysis Algorithm

In the static analysis an abstraction of the partial evaluation is computed for each atom to be partially evaluated. During the static analysis an initial pattern will be constructed for a recursive predicate on the first time that an atom with this predicate is encountered. This pattern is then updated whenever we encounter a recursive occurrence of the atom being partially evaluated.

At each node in an abstract partial evaluation we will have a set of patterns for those recursive predicates marked as safe. The other recursive predicates which have been encountered previously will be marked as unsafe and each such predicate will have a covering atom. At all points the atom in the leftmost selectable literal will be selected for unfolding. The following list describes the unfolding rule for atoms selected in this abstract partial evaluation procedure.

1. First occurrence of a recursive atom.

   We construct the input pattern for this atom and compute a new abstract partial evaluation rooted at this atom.

2. Recursive atom currently marked as safe, with a predicate symbol different from that of the head atom.

   If atom matches some pattern for this predicate

   Then apply that pattern to this atom

   Else Construct the new input pattern for this atom and compute its abstract partial evaluation.

3. Recursive occurrence of the head atom.

   If atom is ordered wrt current input pattern and head atom

   Then compute the new input and output patterns

   Else

   delete all patterns for this predicate, mark the predicate as unsafe and

   If the atom is an instance of the head atom

   Then record the head atom as the covering atom for this predicate

   Else compute the msg of the head atom and selected atom, record this msg as the covering atom for this predicate and compute the abstract partial evaluation of this msg.

4. Unsafe atom.

   If atom is not an instance of the current covering atom

   Then compute the msg of the head atom and selected atom, record this msg as the covering atom for this predicate and compute the abstract partial evaluation of this msg.

5. Non-recursive atom.

   Unfold this atom.

The following text describes the motivation behind each item on the above list in more detail.

In computing these abstract partial evaluations, those selectable atoms with recursive predicates are given special attention. The first time such an atom is encountered, an abstraction of its arguments (its input pattern) is computed and a separate abstract partial evaluation is computed just for this atom. This separate abstract partial evaluation seeks to determine whether the atom is safe or unsafe. An atom is determined to be safe if we can establish an ordering between at least one argument of this atom and the corresponding argument in the ancestor atom. If the atom is safe, then an abstraction of its arguments upon termination of the partial evaluation (its output pattern) will also be computed.

If a recursive atom is encountered which has previously been marked as safe, with a computed input and output pattern, then this atom is compared against the previous input pattern. If the atom matches the input pattern then the output pattern is applied, otherwise a new analysis of this atom is computed.

An atom which has previously been marked unsafe is not unfolded as this atom will not be unfolded when *SAGE* performs the actual partial evaluation. However, a generalisation of this atom and its previous unsafe occurrence is computed and an abstract partial evaluation of this generalised atom is computed, as this new generalised atom will be partially evaluated during the second phase of *SAGE*'s partial evaluation process. This last step may be avoided if the unsafe atom encountered is an *instance* of the atom for which we have already computed the abstract partial evaluation.

When a recursive occurrence of some atom is encountered, its arguments are compared to the matching ancestors arguments. If it can be established that at least one of the arguments in the recursive atom is of lesser complexity than the matching argument in the head atom then the atom will be marked as being safe. If no ordering can be established then the atom is marked as being unsafe. The input pattern is used in ensuring that there must be at least one argument position in which the relevant terms are strictly decreasing in complexity in all recursive cases.

In the abstract partial evaluation an atom with a predicate symbol which is selectable but not recursive is simply unfolded with respect to the definition of this predicate.

Example We construct the abstract partial evaluation of `<- P(GT(0))` wrt the program:

```
P(x) <- Q(x,y) & R(y).
Q(F(x),y) <- P(x) & S(y).
R(G(x)) <- R(x).
```

where the predicates `P`, `R` and `S` are recursive, `S` has been marked as unsafe with covering atom `S(x)` and `R` has the input pattern `[AG]`. We first unfold unfold `P(GT(0))` to produce the new goal `<- Q(GT(0),y) & R(y)`. As `Q` is not recursive unfolding rule 5 says we unfold `Q(GT(0),y)` to produce the goal `<- P(GT(1)) & S(y) & R(y)`, where `GT(1)` is less than `GT(0)` in the ordering on ground terms. By unfolding rule 3 we see that, as the atom `P(GT(1))` is ordered wrt the atom `P(GT(0))` then this atom is discarded. By unfolding rule 4 we also discard the atom `S(y)` leaving the goal `<- R(y)`. As `R(y)` does not match the pattern for `R` then, by unfolding rule 2, we construct the new input pattern `[ANG]` for `R` and compute a new abstract partial evaluation from the goal `<- R(y)`. Unfolding this goal we see that the next unfolding rule to be applied will be rule 3 and that, as the selected atom will not be ordered wrt the head atom of this new abstract partial evaluation, `R` will be marked as unsafe and the pattern for this predicate will be deleted.

4.2.5 Updating Abstract Call Patterns

When computing the abstract partial evaluation of a recursive atom A, the arguments of the input pattern for A are initially set to `AG` when the corresponding argument of A is a variable-free term and `ANG` otherwise. The arguments of the output pattern for A are initially set to $\perp$. The pattern for A will be modified in one of two cases, either when a branch in the abstract partial evaluation of A leads to a terminating (or base) case or when a branch leads to a recursive occurrence of A. When a base case is encountered the output pattern for A is updated using the algorithm of Figure 4.6. When a recursive occurrence of A is encountered the entire pattern for A is updated using the algorithm of Figure 4.7.

Input:

an atom A

a substitution θ (the answer substitution for this terminating branch of the abstract partial evaluation of A)

a pattern $\mathcal{P}$ for A

Output:

$\mathcal{P}'$, an updated pattern for A

Initialisation

$A' := A\theta$

$O_{\mathcal{P}} :=$ the output pattern of $\mathcal{P}$

For each argument t of A' **do**

If t is an abstract ground term and the corresponding argument of $O_{\mathcal{P}}$ is $\perp$ or `AG`

Then

the corresponding argument in $O'_{\mathcal{P}} :=$ `AG`

Else

the corresponding argument in $O'_{\mathcal{P}} :=$ `ANG`

EndFor

$\mathcal{P}' := \mathcal{P}$ with $O'_{\mathcal{P}}$ replacing $O_{\mathcal{P}}$

Figure 4.6: Updating Pattern for Base Case

Input:

an atom A with predicate symbol p

A', a recursive occurrence of A

a set of patterns $\mathcal{S}$ for p

a particular pattern $\mathcal{P} \in \mathcal{S}$ for A

a set of (unsafe) predicates U

Output:

$\mathcal{S}'$, the updated set of patterns for p

U', the updated set of unsafe predicates

Initialisation

$I_{\mathcal{P}}$:= the input pattern of $\mathcal{P}$

$O_{\mathcal{P}}$:= the output pattern of $\mathcal{P}$

If A' is ordered wrt A and $\mathcal{P}$

Then

For each argument a of A' **do**

If a is ordered wrt the corresponding arguments in A and $I_{\mathcal{P}}$

Then

the corresponding argument in $I'_{\mathcal{P}}$:= `AG`

Else

the corresponding argument in $I'_{\mathcal{P}}$:= `ANG`

If the following are true:

- the corresponding argument of $O_{\mathcal{P}}$ is $\perp$ or `AG`
- a contains no variables which do not occur in the corresponding argument of A

Then

the corresponding argument in $O'_{\mathcal{P}}$:= `AG`

Else

the corresponding argument in $O'_{\mathcal{P}}$:= `ANG`

EndFor

$\mathcal{P}' := \mathcal{P}$ with $I'_{\mathcal{P}}$ replacing $I_{\mathcal{P}}$ and $O'_{\mathcal{P}}$ replacing $O_{\mathcal{P}}$

$\mathcal{S}' := \{\mathcal{P}'\} \cup (S - \{\mathcal{P}\})$

$U' := U$

Else

$\mathcal{S}' := \emptyset$

$U' := U \cup \{p\}$

Figure 4.7: Updating Pattern for Recursive Case

Example We illustrate the above process by the abstract partial evaluation of the program:

```
P([x|y],z) <- Q(x,w) & R(y,w,z).
Q(A,B).
Q(B,C).
R([],x,[x]).
R(x,y,[y|z]) <- P(x,z).
```

with respect to the abstract goal ←`P(`$\mathcal{T}$`,v)`, where $\mathcal{T}$ is some abstract ground term. The predicate `P` is a recursive predicate and so we start our investigation by computing an initial pattern for it. The initial input pattern for `P` is `[AG,ANG]` and we set its output pattern to be `[⊥,⊥]`. We unfold the atom `P(`$\mathcal{T}$`,v)` in the initial (abstract) resultant `P(`$\mathcal{T}$`,v)`←`P(`$\mathcal{T}$`,v)` to produce the resultant `P(`$\mathcal{T}$`,v)`←`Q(`$\mathcal{T}_1$`,v1) & R(`$\mathcal{T}_2$`,v1,v)`, where $\mathcal{T}_1$ and $\mathcal{T}_2$ are both ground subterms of $\mathcal{T}$. We select the atom `Q(`$\mathcal{T}_1$`,v1)` in this resultant and unfold it to produce the two new resultants `P(`$\mathcal{T}$`,v)`←`R(`$\mathcal{T}_2$`,B,v)` and `P(`$\mathcal{T}$`,v)`←`R(`$\mathcal{T}_2$`,C,v)`. Although *SAGE* will analyse each of these resultants in turn, we shall only look in detail at the analysis of the first as the resultants are analogous. We unfold the atom `R(`$\mathcal{T}_2$`,B,v)` in this resultant to produce the two resultants `P(`$\mathcal{T}$`,[B])`← and `P(`$\mathcal{T}$`,[B|v2])`←`P(`$\mathcal{T}_2$`,v2)`. For the first resultant we combine the current output pattern for `P`, `[⊥,⊥]`, with the terms `[`$\mathcal{T}$`,[B]]` to produce the new output pattern `[AG,AG]`. In the second resultant we select the atom `P(`$\mathcal{T}_2$`,v2)`. This is a recursive occurrence of the head atom and so we seek to establish an ordering between these two atoms. As the current input pattern for `P` is `[AG,ANG]`, we see that we may only check that the arguments in the first argument position are ordered. As $\mathcal{T}_2$ is a subterm of $\mathcal{T}$ we may deduce that the unfolding of this resultant will terminate. We next combine the current output pattern for `P`, `[AG,AG]`, with the terms `[`$\mathcal{T}_2$`,v2]` to produce the new output pattern. We see that the first argument position is obviously set to `AG`. The second argument position we also set to `AG`, as the head-term, `[B|v2]`, contains no variables that do not occur in the matching atom term, `v2`, and the previous output pattern for this argument is `AG`. As the analysis of the remaining resultant in this investigation is analogous, we see that we complete the analysis with `P` marked as a safe predicate, with input pattern `[AG,ANG]` and output pattern `[AG,AG]`.

4.2.6 The Dynamic Selection Strategy for *SAGE*

Although until now we have continually referred to a meta-program as generally selecting *literals* for unfolding, we must generalise this somewhat for standard Gödel programs. Statements and goals in standard Gödel may include arbitrary formulas, committed formulas and conditionals. *SAGE* does not restrict selection of a formula for unfolding to literals only. The selected formula may be an atom, a negated formula, a conditional or a committed formula. The dynamic selection strategy for *SAGE* implements a fundamentally leftmost selection strategy with respect to a set of selectable

predicates L and the following constraints:

Atoms

Selected iff the predicate for this atom is safe and selectable.

Negations

Selected.

Conditionals

Selected.

Committed Formulas

Selected subject to the restriction described below.

As has been described in Section 2.1 implications and equivalences are transformed to a normal form. This transformation is handled by *SAGE*'s dynamic selection strategy.

The restriction that *SAGE* places on the selection of committed formulas is that a committed formula may be specialised only once in a partial evaluation.

The above restrictions show that *SAGE* has a very conservative selection strategy for unfolding atoms with unsafe predicates, although it is precisely the strategy that we need to produce residual programs of the form presented in Section 3.4. Calls to unsafe atoms will not be unfolded during a partial evaluation and the specialised definitions of the unsafe predicates are produced by computing the partial evaluations of the formulas resulting from a single unfolding step performed upon the covering atom for each unsafe predicate.

This strategy trivially satisfies the final condition for the guaranteed termination of *SAGE*'s selection strategy, that the dynamic selection strategy ensure that recursive occurrences of atoms with unsafe predicates are unfolded only a finite number of times. We next prove that *SAGE*'s selection strategy satisfies the first two conditions for termination, that the static part terminates and is sound.

Definition Let A be an atom. We say that the result of replacing all ground terms in A by an abstract ground term is the *abstraction* for A.

Note that if A is some abstract ground atom then A is the abstraction for the set of atoms $\{A' : A' \text{ matches } A\}$.

Lemma 4.2.1 *Let P be a Gödel program, A an atom with a recursive predicate p in P, A^* the abstraction of A and $\mathcal{P}$ a pattern for p computed by the static analysis algorithm of Figure 4.4 such that A^* matches $\mathcal{P}$. Then the following conditions hold:*

(a) The partial evaluation of A wrt P, using the above dynamic selection strategy, will only unfold

finitely many recursive occurrences of A.

(b) For all answers θ computed by the partial evaluation of A, the result A' of applying $\mathcal{P}$ to A^* is an abstraction of $A\theta$.

Proof Part (a): Note that, as the static analysis has computed a pattern for p, p is a safe predicate and therefore the partial evaluation of A will unfold *every* recursive occurrence of A.

Examining case 3 of the unfolding of atoms in the static analysis it is trivial to see that every recursive occurrence of A^* in the abstract partial evaluation of A^* will be ordered wrt A^* and $\mathcal{P}$. Part (a) follows directly.

Part (b): Consider a branch in the tree used to construct the partial evaluation of A, with computed answer ϕ. In this branch we will unfold n recursive occurrences of A. The definition of A may depend upon m other safe recursive predicates, each of which will have its own pattern. We prove part (b) by a double induction upon m and n.

Case $m = 0$:

Base case: assume that $n = 0$, then the static analysis contains a branch in which no recursive occurrences of A^* appear. By the algorithm of Figure 4.6 only those arguments of A^* which are instantiated to ground terms at this point will be marked as `AG` in the output pattern for $\mathcal{P}$. Arguments will be ground for one of the following reasons:

1. This argument was ground in A^*
2. This argument was bound to a ground term during the computation
3. This argument was instantiated to a ground term by the application of some pattern.

As $m = 0$ and $n = 0$ then case 3 does not hold in this instance. It is trivial to see that A' is an abstraction for $A\phi$ whenever case 1 or case 2 holds for arguments bound to ground terms in A^*.

Inductive case: assume that every recursive occurrence of A in the partial evaluation has a corresponding abstraction which is a recursive occurrence of A^* in the static analysis. The above three reasons for determining how a term in A^* may be bound to a ground term also apply in this recursive case. It is again trivial to see that part (b) is correct whenever cases 1 and 2 hold. As $m = 0$, any pattern applied in case 3 will be a recursive occurrence of A^*. By the algorithm of Figure 4.7 only those arguments of A^* which contain no variables that do not occur in the corresponding arguments of the recursive occurrences of A^* will be marked as `AG` in the output pattern for $\mathcal{P}$. By the inductive hypothesis, these variables will be ground if the relevant argument in the output pattern for $\mathcal{P}$ is marked `AG` and therefore the corresponding argument in $A\phi$ will be ground. A' is therefore an abstraction of $A\phi$.

Case $m = i + 1$:

The inductive case is a trivial extension of the above inductive proof for the base case. ∎

Theorem 4.2.2 *Let P be a Gödel program, G a goal for P and L a set of selectable predicates. The static analysis algorithm of Figure 4.4 of G wrt P and L terminates, returning a set of predicates S and a set of atoms M for which:*
(a) S is safe wrt the partial evaluation of G wrt P, using the above dynamic selection strategy.
(b) $M = \{p(\tilde{t}) : p \in L - S\}$ is a set of covering atoms for the unsafe atoms in the above partial evaluation.

Proof Let **A** be the set of L-selectable atoms in G. We give the proof for the case when **A** consists of a single atom F. The extension to the general case is straightforward. First we prove the termination of the static analysis algorithm.

It is trivial that computing the predicate dependency graph for F will terminate.

The computation of an abstract partial evaluation of F terminates if recursive occurrences of any atom with a recursive definition are only unfolded a finite number of times. To prove this we must first define an ordering on atoms in an abstract partial evaluation with the same predicate symbol. We define a termination function μ from atoms into the well-founded set of non-negative integers under $<$.

We define μ in terms of the mapping μ^* as:

$\mu^*(\mathtt{P}(\mathtt{t_1}, \ldots, \mathtt{t_n})) = \mu'(\mathtt{t_1}) + \ldots + \mu'(\mathtt{t_n})$, where $\mathtt{P}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ is an atom
with the inductive definition for μ':
$\mu'(t) = 1$, where t is a variable or a constant
$\mu'(\mathtt{F}(\mathtt{t_1}, \ldots, \mathtt{t_n})) = \mu'(\mathtt{t_1}) + \ldots + \mu'(\mathtt{t_n}) + 1$, where $\mathtt{F}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ is a function term.

Let $v(A)$ be the number of distinct variables in an atom A, then we define $\mu(A) = \mu^*(A) - v(A)$

We note that given two atoms, A_1 and A_2, with the same predicate symbol such that A_2 is not an instance of A_1, where $\mu(A_1) = m$ and $\mu(A_2) = n$, then it is trivial that $k < m$ and $k \leq n$, where $k = \mu(A^*)$ and A^* is the msg of A_1 and A_2.

Let A be an atom, selected at some point in the abstract partial evaluation, T, with head atom H. We say that an atom B is a *descendant* of A in T if B appears in the subtree of T rooted at A. To prove that the abstract partial evaluation T is finite we must prove that, following the selection of A, we only unfold atoms with the same predicate symbol as A a finite number of times. For each of the cases for unfolding A we prove that if we select an atom A', where A' is a descendant of A in T with a matching predicate symbol, then $\mu(A') < \mu(A)$.

1. A is the first occurrence of some recursive atom.
   An abstract partial evaluation of A is computed. Cases 3-5 below prove the finiteness of the unfolding of atoms with some predicate symbol p when the head atom also has predicate symbol p.

2. A is a recursive atom currently marked as safe with a predicate symbol other than H's.

   If A matches some pattern for this predicate

   Then A is not unfolded

   Else An abstract partial evaluation of A is computed. Cases 3-5 below prove the finiteness of the unfolding of atoms with some predicate symbol p when the head atom also has predicate symbol p.

3. A is a recursive occurrence of H.

   If A is ordered wrt H and current input pattern

   Then A is not unfolded

   Else % A is marked as *unsafe*

   If A is an instance of H

   Then A is not unfolded

   Else $A^* :=$ the msg of A and H. An abstract partial evaluation of A^* is computed. It is trivial to see that $\mu(A^*) < \mu(A)$

4. A is already marked unsafe.

   If A is not an instance of the current covering atom

   Then $A^* :=$ the msg of A and the current covering atom. An abstract partial evaluation of A^* is computed. It is trivial that $\mu(A^*) < \mu(A)$

5. A is not recursive.

   A cannot be repeatedly unfolded.

The recursive loop of the algorithm computes abstract partial evaluations, adding predicates to the list of unsafe predicates. The termination condition for this loop is that the number of unsafe predicates so computed is the same as for the previous abstract partial evaluation. As there are only finitely many recursive predicates in P and no predicate may be marked as safe once it is marked as unsafe, the algorithm will terminate.

Next we prove the soundness of the algorithm. First we prove that, having terminated, a final abstract partial evaluation, T, has been computed, for which the following conditions have been met:

Condition 1: For each recursive predicate in the set of safe predicates S such that an atom with this predicate appears in T there are a set of patterns, $\mathcal{P}$, for this predicate such that every atom in T with this predicate matches some pattern in $\mathcal{P}$.

Condition 2: For each unsafe predicate the set of covering atoms contains an atom, C, with this predicate. Every atom in T with this predicate (including recursive calls of atoms with this predicate in the abstract partial evaluation of C) is an instance of C.

To meet the terminating condition for the static analysis, no predicates can have been marked as unsafe during the computation of T. Consider an atom, A, that has been selected for unfolding at some point in T. For each of the cases for unfolding A we prove that the above two conditions have been met.

1. A is the first occurrence of some recursive atom.
   A pattern, $\mathcal{P}$, is computed for A and an abstract partial evaluation of A is computed. A will not be marked as unsafe and therefore $\mathcal{P}$ will not be deleted. It is trivial that A matches $\mathcal{P}$ and therefore condition 1 is satisfied. As A is safe, condition 2 is trivially satisfied.

2. A is a recursive atom currently marked as safe with a predicate symbol other than that of the head atom.
   If A matches some pattern for this predicate
   Then condition 1 is satisfied. As A is safe, condition 2 is trivially satisfied.
   Else A new pattern, $\mathcal{P}$, is computed for A and an abstract partial evaluation of A is computed. A will not be marked as unsafe and therefore $\mathcal{P}$ will not be deleted. It is trivial that A matches $\mathcal{P}$ and therefore condition 1 is satisfied. As A is safe, condition 2 is trivially satisfied.

3. A is a recursive occurrence of the head atom.
   A cannot be marked as being unsafe, therefore A must be ordered wrt H and the current input pattern. Therefore A matches the current input pattern and condition 1 is satisfied. As A is safe, condition 2 is trivially satisfied.

4. A is already marked unsafe.
   If A is not an instance of the current covering atom
   Then the new covering atom, A^*, is the msg of A and the current covering atom. All previously encountered atoms with the same predicate symbol as A are instances of the current covering atom and are therefore instances of A^*. A is also an instance of A^* and therefore condition 2 is satisfied. As A is unsafe, condition 1 is trivially satisfied.

5. A is not recursive.
   The two conditions are trivially satisfied.

Part (a) follows directly from condition 1 above and lemma 4.2.1.
Part (b) follows directly from condition 2 above. ∎

4.2.7 Justifying *SAGE*'s Selection Strategy

A general outline of the justification for *SAGE*'s selection strategy is as follows. When partially evaluating meta-programs we generally find that key arguments to the meta-programs, mostly

the ground representations of programs or theories, are ground. Many recursive predicates in meta-programs, particularly those we would seek to unfold, are recursive by the nature of the fact that they process complex terms. These predicates are recursive because they are called recursively on subterms of the term or terms they process. The fact that these key arguments are ground whenever these predicates are called allows us to deduce that these calls will terminate, as ground terms are of necessarily finite complexity. The static analysis also seeks to determine which previously uninstantiated terms will have been instantiated to ground terms at the termination of these calls. This analysis shares similarities with termination analyses such as those of [44, 48, 75] and mode-inferencing analyses such as those of [17, 38, 53].

As an example, consider the Gödel meta-interpreter of Figure 4.8, which conforms to the basic framework of Figure 3.8. This meta-interpreter has the two recursive predicates `Demo1` and `Select`. The definitions for the Gödel system predicates `FormulaMaxVarIndex` and `ApplySubstToFormula` are recursive and the definition for `Resolve`, presented in the previous chapter, also relies upon recursive predicates.

When specialising the above interpreter with respect to a known program and an unknown goal we see that the above recursive predicates fall naturally into two categories. In the first we have the predicates `Demo1`, `Select`, `FormulaMaxVarIndex` and `ApplySubstToFormula` which are not sufficiently instantiated to be fully unfolded. These predicates would be unsafe in the partial evaluation of this interpreter. We would therefore not unfold calls with these predicates but would seek to specialise the definitions of these predicates in a manner which retained their recursive nature. By contrast, a call to `StatementMatchAtom` for a known program will return the ground representation of a statement in this program as its fourth argument. This argument is passed to the call to `Resolve`. Examining the definition for `Resolve` in the previous chapter we see that if the WAM-like predicates are declared to be non-selectable, then we may unfold all other atoms in the unfolding of a call to `Resolve` with respect to a known statement. This unfolding will terminate returning a single conjunction of WAM-like atoms as the residual code for this call.

When a meta-interpreter interprets the execution of some goal wrt a program we generally find that we can seperate the various calls made by the interpreter into those that perform some analysis of the program and those that perform some analysis of the goal. When we partially evaluate a meta-interpreter wrt an object program the goal that the meta-interpreter is to interpret is unknown. Naturally therefore it seems obvious that a partial evaluation of a meta-interpreter should seek to unfold those calls which perform an analysis of the object program and leave as residual those calls which perform an analysis of the goal. In the static analysis performed by *SAGE* we have simply taken advantage of the fact that Gödel uses a ground representation. Thus when a recursive call in a meta-program is analysing some part of the object program, some significant argument will be ground in this call. When a recursive call is analysing a part of the goal for this program,

```
Demo(program, goal, answer) <-
  FormulaMaxVarIndex(goal, var) &
  EmptyTermSubst(subst) &
  Select(goal, left, selected, right) &
  Demo1(selected, program, left, right, var, subst, computed_answer) &
  ApplySubstToFormula(goal, computed_answer, answer).

Demo1(None, program, _, _, _, subst, subst).
Demo1(Positive(atom), program, left, right, var, subst_in, subst_out) <-
  Atom(goal) &
  StatementMatchAtom(program, _, atom, statement) &
  Resolve(atom, statement, var, new_v, subst_in, new_subst, body) &
  And(left, body, left1) &
  And(left1, right, new_goal) &
  Select(new_goal, new_left, selected, new_right) &
  Demo1(selected, program, new_left, new_right, new_v, new_subst, subst_out).

Select(formula, formula, None, formula) <-
  EmptyFormula(formula).
Select(formula, left, selected, right) <-
  And(l, r, formula) &
  Select(l, l1, s, r1) &
  IF  s = None
    THEN  Select(r, left, selected, right)
    ELSE  left = l1 &
          selected = s &
          AndWithEmpty(r1, r, right).
Select(formula, empty, Positive(formula), empty) <-
  Atom(formula) &
  EmptyFormula(empty).
```

Figure 4.8: A Basic Gödel Meta-Interpreter

no such argument will be ground. Similar arguments have been made by Sestoft for functional programs [62].

4.3 The Unfolding Strategy for *SAGE*

There are four varieties of Gödel formulas which *SAGE* treats as atomic for the purpose of unfolding. That is, there are four varieties of formulas which *SAGE* will not unfold by simply transforming them to some normal form during selection. These are safe selectable atoms, negated formulas, conditional formulas and committed formulas. We shall describe *SAGE*'s treatment of these four varieties in turn. Firstly however, we briefly present *SAGE*'s internal representation of the nodes in the partial evaluation being computed.

4.3.1 A Helpful Representation for Resultants

SAGE represents the nodes in a partial evaluation as resultants whose head is the formula being specialised and whose body is the current specialised code for this formula. These resultants are represented by *SAGE* as terms of the form `Res(head,body,s,v,c)`, where `head` is the head of the resultant and `body` its body. The current substitution for this resultant is stored by `s`, the current maximum variable index by `v` and `c` stores a list of the augmented labels for the commits appearing in the body of this resultant. This representation makes explicit all pertinent information for any node in a *SAGE* partial evaluation.

4.3.2 Unfolding Atoms

An atom will be either an open or closed atom in a Gödel program, depending on whether its predicate (or proposition) symbol is declared in an open or a closed module. Having selected an atom for unfolding, these two mutually exclusive cases are determined by whether a call to `DefinitionInProgram`, in the case of an open atom, or `DeclaredInClosedModule`, in the case of a closed atom, succeeds.

Open Atoms

A call to `DefinitionInProgram` will return the definition of the predicate or proposition in an open atom. *SAGE* uses a call to `ResolveAll` to perform an unfolding step for this atom. `ResolveAll` returns two lists corresponding to the answer substitutions and resolvents for the successful unfolding steps. These lists are converted to new resultants in *SAGE* by a call to `ConstructResultants(b,s,c,h,l,r,v,[],res,res1)`, where `b` is the list of resolvents, `s` the matching list of substitutions, `c` the list of commits in the current resultant, `h` the head of the current

resultant, `l` and `r` the formulas to the right and left respectively of the selected atom in the current resultant, `v` the maximum variable index returned by the call to `ResolveAll` and `res` and `res1` are respectively the list of resultants forming the nodes in the current partial evaluation before and after this unfolding step.

When constructing the new resultants formed by unfolding an atom in a resultant, *SAGE* analyses the resolvents of the unfolding step to detect any new commit labels which may have been introduced. These commit labels will have been renamed by the call to `ResolveAll`. `ConstructResultants` calls `CommitsInFormula` to determine all new commits introduced and appends the relevant augmented commit labels to the current list of commit labels for each resultant.

```
ConstructResultants([], [], _, _, _, _, _, _, res, res).
ConstructResultants([body|rest], [subst|ss], coms, head, left, right, var, new
                    , r, [Res(head, body1, subst, var, coms1)|r1]) <-
  {CommitsInFormula(body, new, new1, coms, coms1)} &
  AndWithEmpty(left, body, body2) &
  AndWithEmpty(body2, right, body1) &
  ConstructResultants(rest, ss, coms, head, left, right, var, new1, r, r1).

CommitsInFormula(formula, new, new1, coms, coms1) <-
  And(left, right, formula) |
  CommitsInFormula(left, new, new2, coms, coms2) &
  CommitsInFormula(right, new2, new1, coms2, coms1).
CommitsInFormula(formula, new, [C(labl, occ)|new1], coms, [C(labl, occ)|coms1]) <-
  Commit(label, formula1, formula) |
  IF SOME [occ1, new2] DeleteFirst(C(labl, occ1), new, new2)
    THEN  occ = occ1+1 &
          CommitsInFormula(formula1, new2, new1, coms, coms1)
    ELSE  occ = 1 &
          CommitsInFormula(formula1, new, new1, coms, coms1).
CommitsInFormula(formula, new, new, coms, coms) <-
  Atom(formula) \/
  Some(_, _, formula) \/
  IfSomeThenElse(_, _, _, _, formula) \/
  EmptyFormula(formula) \/
  IfThenElse(_, _, _, formula) \/
  IfSomeThen(_, _, _, formula) \/
  IfThen(_, _, formula) \/
  Or(_, _, formula) \/
  All(_, _, formula) \/
  Implies(_, _, formula) \/
  IsImpliedBy(_, _, formula) \/
```

```
  Equivalent(_, _, formula) \/
  Not(_, formula) |.
```

For example, let A be an atom selected in a *SAGE* resultant. Let H be the head of the resultant, L the formula to the left of A, R the formula to the right of A, θ the current substitution and C the list of (augmented) labels for the commits appearing in this resultant.

We unfold A to return three resolvents, A_1, A_2 and A_3, with respective new substitutions θ_1, θ_2 and θ_3. If each resolvent contains a committed formula with a commit that has been assigned the new integer label `n`, then `ConstructResultants` will construct the three new *SAGE* resultants representing $(H \leftarrow L\&A_1\&R)\theta_1$, $(H \leftarrow L\&A_2\&R)\theta_2$ and $(H \leftarrow L\&A_3\&R)\theta_3$. It will record the list of augmented commit labels for these resultants as being `[C(n,1)|`C`]`, `[C(n,2)|`C`]` and `[C(n,3)|`C`]` respectively.

Closed Atoms

Closed atoms are unfolded by *SAGE* with a call to `ComputeAll`, which returns the new substitutions and new maximum variable index following the interpretation of this atom in much the same manner that `ResolveAll` returns these values after an unfolding step. The major difference between `ComputeAll` and `ResolveAll` is that `ComputeAll` interprets the computation of the atom in its second argument up to the step where the computation either succeeds, fails or flounders, where `ResolveAll` interprets only a single unfolding step.

The goals returned by `ComputeAll` will be either empty formulas, in the case of successful computations, or delayed goals, in the case of floundered computations. As these delayed goals may later be further instantiated, *SAGE* will determine the free variables in such goals. A delayed closed atom may be selected for further unfolding (using `ComputeAll`) should any of these free variables become instantiated to a non-variable term later in the partial evaluation.

4.3.3 Unfolding Negated Formulas

As described by the algorithm in Figure 2.1, to unfold a negated formula, ~F, we first compute a partial evaluation of F. For each residual formula F_i in the partial evaluation of F we must determine which of the free variables in F have been instantiated. This is accomplished by a call to `GetFormulaBindings(f,subst,bindings)` where `f` is the representation of the formula F, `subst` is the answer substitution computed for the residual formula F_i and `bindings` is the list of variable-term pairs for the free variables in F and their bindings in F_i.

```
GetFormulaBindings(formula, subst, bindings) <-
  FormulaVariables(formula, vars) &
  BoundVariables(vars, subst, bindings1) &
  FilterBindings(bindings1, bindings1, vars, bindings).
```

```
BoundVariables([], _, []).
BoundVariables([var|rest], subst, bind) <-
  (
  IF  SOME  [term]  BindingInTermSubst(subst, var, term)
      THEN  bind = [var @ term|bind1]
      ELSE  bind = bind1
  ) &
  BoundVariables(rest, subst, bind1).

FilterBindings([], _, _, []).
FilterBindings([var @ term|rest], bindings, vars, bindings1) <-
  FilterBindings(rest, bindings, vars, bindings2) &
  IF  Variable(term)
    THEN
      (
      DeleteFirst(var @ term, bindings, other_bindings) &
      IF  ( Member(term, vars) \/ Member(_ @ term, other_bindings) )
        THEN  bindings1 = [var @ term|bindings2]
        ELSE  bindings1 = bindings2
      )
    ELSE
      bindings1 = [var @ term|bindings2].
```

`GetFormulaBindings` performs a filtering operation which ensures that only essential bindings for the free variables of F are recorded. We refer to a binding which instantiates a free variable in F to another variable which is not bound elsewhere in F and F_i as an *unessential binding*. No binding will be recorded for a free variable in F which is either not instantiated or only appears in a single unessential binding.

If the partial evaluation for F contains no residual code then F has failed finitely and ~F has succeeded (without binding any variables). If the partial evaluation for F contains a residual F_i which is the empty formula and for which there are no essential bindings for the free variables of F, then ~F has failed safely. In this case the resultant in which ~F appeared will be deleted.

If neither of the above two cases hold then we must construct the residual code for ~F, as described in Figure 2.1. Given a list of variable-term pairs, a call to `MakeBindings` constructs the formula (a conjunction of equality atoms) which records the relevant bindings.

```
MakeBindings([], bind) <-
  NewProgram("Empty", p) &
  ProgramPropositionName(p, "", "True", true) &
  PropositionAtom(bind, true).
```

```
MakeBindings([b|rest], bind) <-
  NewProgram("Empty", p) &
  ProgramPredicateName(p, "", "=", 2, equals) &
  MakeBindings1(rest, b, equals, bind).

MakeBindings1([], var @ term, equals, bind) <-
  PredicateAtom(bind, equals, [var, term]).
MakeBindings1([b|rest], var @ term, equals, bind) <-
  PredicateAtom(atom, equals, [var, term]) &
  And(atom, bind1, bind) &
  MakeBindings1(rest, b, equals, bind1).
```

The conjunction of equality atoms produced by `MakeBindings` is conjoined with the residual formula F_i and negated to produce the residual code for this branch of the computation tree for F. The residual negated formulas produced from each F_i are conjoined to produce the specialised code for ~F.

For example, if a program contained the following definition for a predicate `P`:

```
P(x) <-
  Member(x, [A, B, C]) &
  Q(x).
```

and we specialised the formula `~P(x)` with respect to this program, where `Q` was a non-selectable predicate, *SAGE* would produce the specialised formula:

```
~(x = A & Q(x)) & ~(x = B & Q(x)) & ~(x = C & Q(x))
```

4.3.4 Unfolding Conditional Formulas

In Section 2.1 we stated that a conditional formula of the form `IF` C `THEN` T can be considered as being the formula `IF` C `THEN` T `ELSE True`. For the purposes of partial evaluation we may also consider conditionals of the form `IF` C `THEN` T `ELSE` E to be the formula `IF SOME []` C `THEN` T `ELSE` E. Therefore *SAGE* needs only consider conditionals of the form `IF SOME` V C `THEN` T `ELSE` E.

As described by the algorithm in Figure 2.2, to unfold a conditional `IF SOME` V C `THEN` T `ELSE` E we firstly compute the partial evaluation of the condition C. If this partial evaluation produces no residuals then C has failed finitely and we proceed to specialise E. If the partial evaluation produces residuals for C then we construct the specialised condition with a call to `MakeCondition(v,c,r,new)`, where `v` is the list of quantified variables V, `c` is the condition C, `r` is the list of residuals of the partial evaluation of C and `new` is the specialised condition.

```
MakeCondition(some, condition, residuals, new_condition) <-
  Some(some, condition, condition1) &
  FormulaVariables(condition1, vars) &
  MakeCondition1(residuals, vars, new_condition).

MakeCondition1([], _, condition) <-
  EmptyFormula(condition).
MakeCondition1([Res(_, test, subst, _, _)|rest], vars, condition) <-
  BoundVariables(vars,  subst,  bindings1) &
  FilterBindings(bindings1,  bindings1,  vars,  bindings) &
  MakeBindings(bindings, bind) &
  ApplySubstToFormula(test, subst, test1) &
  AndWithEmpty(bind, test1, new_test) &
  MakeCondition1(rest, vars, condition1) &
  Or(new_test, condition1, condition).
```

If one of the residual formulas for C is the empty formula and this residual has performed no essential bindings of free variables in C then the corresponding disjunct of the new condition will be the formula `True`. The condition has therefore succeeded safely in at least one case and we proceed to specialise the formula C^* `&` T, where C^* is the specialised condition.

If neither of the above two cases hold then we specialise T and E. We construct the new then-part T^* and else-part E^* by calling `MakeCondition` for the residual code for T and E respectively. We then construct the residual conditional formula by a call to `NewConditional(v,o,c,t,e,f)`, where `v` is the list of variables V, `o` the original condition C, `c` the new condition C^*, `t` the new then-part T^*, `e` the new else-part E^* and `f` the new conditional formula.

```
NewConditional(some, old_test, new_test, new_then, new_else, new_conditional) <-
  Some(some, old_test, old_test1) &
  FormulaVariables(old_test1, old_free_vars) &
  FormulaVariables(new_test, new_vars) &
  RemoveFreeVars(old_free_vars, new_vars, new_some) &
  IF  new_some = []
    THEN  IfThenElse(new_test, new_then, new_else, new_conditional)
    ELSE  IfSomeThenElse(new_some, new_test, new_then, new_else, new_conditional).

RemoveFreeVars([], s, s).
RemoveFreeVars([var|rest], some, some1) <-
  IF  SOME  [some2]  DeleteFirst(var, some, some2)
    THEN  RemoveFreeVars(rest, some2, some1)
    ELSE  RemoveFreeVars(rest, some, some1).
```

The above code ensures that any new variables local to the new condition appear in the new list

of quantified variables for this condition.

For example, given the program:

```
P(F(x)).


Q(F(x)) <-
  Member(x, [A, B, C]).


R(D).
R(x) <-
  S(x)
```

if we specialised the conditional `IF P(x) THEN Q(x) ELSE R(x)` with respect to this program, where `S` was a non-selectable predicate, *SAGE* would produce the specialised conditional:

```
IF SOME [y] x = F(y)
  THEN  ( y = A \/ y = B \/ y = C )
  ELSE  ( x = D \/ S(x) )
```

Note that in this example, `y` is a new variable that has been introduced which is local to the new condition and new then-part. Therefore `y` has had to be locally quantified in the residual conditional. By contrast, the variable `x`, which was a free variable in the condition of the original conditional, remains free in the condition of the residual conditional.

4.3.5 Unfolding Committed Formulas

In Section 2.3 we presented an outline of *SAGE*'s strategy for handling the unfolding of formulas containing committed formulas. Two issues were dealt with, freeness and regularity. In order to guarantee the correctness of unfolding without enforcing regularity we noted that an augmented representation of the labels of commits needed to be implemented.

Handling Regularity

SAGE records the augmented labels of the commits appearing in the body of a resultant as a list of terms `C(n,i,m,j)` and `C(n,i)`, where `n`, `i`, `m` and `j` are integers. A term `C(n,i,m,j)` (where `m > 0` and `j > 0`) represents the augmented commit label $n_i^{m:j}$ and a term `C(n,i)` represents the augmented commit label $n_i^{0:0}$. The augmented labels for commits introduced by an unfolding step are calculated during a call to `ConstructResultants`, as described above.

Having selected a committed formula, $\{F\}_n$ (with augmented commit label $n_i^{0:0}$), from a resultant R, *SAGE* computes the partial evaluation of the formula F. The new resultants

produced by the unfolding of this committed formula are then constructed by a call to `ConstructCommits(r,n,i,m,1,hd,lf,rt,rs,rs1)`, where `r` is the list of residual resultants for the partial evaluation of F, `n` and `i` are the integers n and i respectively, `m` is an integer one greater then any appearing as a commit label in the current partial evaluation, `hd` is the head of R, `lf` and `rt` respectively the formulas to the left and right of $\{F\}_n$ in the body of R and `rs` and `rs1` are respectively the list of resultants forming the nodes in the current partial evaluation before and after this unfolding step.

```
ConstructCommits([], _, _, _, _, _, _, _, res, res).
ConstructCommits([Res(_, body, s, v, cs)|rest], n, i, m, j, hd, lf, rt, res
                 ,[Res(hd, new_body, s, v, [C(n, i, m, j)|cs])|res1]) <-
  Commit(n, body, body1) &
  AndWithEmpty(lf, body1, lf1) &
  AndWithEmpty(l1, rt, new_body) &
  ConstructCommits(rest, n, i, m, j+1, hd, lf, rt, res, res1).
```

Note that *SAGE* only permits the unfolding of a committed formula with an augmented commit label of the form $n_i^{0:0}$. This ensures that *SAGE* will only attempt to unfold a committed formula once in a partial evaluation, as described in Section 2.3.

For example, let the three formulas A_1, A_2 and A_3 be the three residuals for the partial evaluation of a formula $\{F\}_n$, with augmented commit label `C(n,i)`. If `m` were the integer one greater then any appearing as a commit label in the current partial evaluation then the new augmented commit labels for the three new resultants constructed from A_1, A_2 and A_3 by a call to `ConstructCommits` would be `C(n,i,m,1)`, `C(n,i,m,2)` and `C(n,i,m,3)` respectively.

Having performed the partial evaluation of some atom, A, *SAGE* must next construct a matrix of labels with which to rename the commits in the residual code. This is to ensure that the pruning in the specialised code for A is correct with respect to the original code for A. *SAGE* constructs the matrix of new commit labels used to rename the residual commits with a call to `RenameCommits`.

```
RenameCommits(res, matrix) <-
  GetAllCommits(res, [], commits, [], sins) &
  SumCommits(commits, sums) &
  SumCommitsS(sins, sums1, sums) &
  RegulateCommits(sums1, 0, [], matrix).

GetAllCommits([], list, list, sin, sin).
GetAllCommits([Res(_, _, _, _, _, coms)|rest], list, list1, sin, sin1) <-
  GetCommits(coms, list, list2, sin, sin2) &
  GetAllCommits(rest, list2, list1, sin2, sin1).

GetCommits([], list, list, sin, sin).
```

```
GetCommits([C(label, o)|r], list, list1, sin, sin1) <-
  IF SOME [found, l2] DeleteFirst(F(label, found), list, l2)
    THEN  (
          IF SOME [n, f1] DeleteFirst(C(o, n), found, f1)
            THEN
              GetCommits(r, [F(label, [C(o, n+1)|f1])|l2], list1, sin, sin1)
            ELSE
              GetCommits(r, [F(label, [C(o, 1)|found])|l2], list1, sin, sin1)
          )
    ELSE  GetCommits(r, [F(label, [C(occ, 1)])|list], list1, sin, sin1).
GetCommits([C(label, occ, foo, bar)|r], list, list1, sin, sin1) <-
  (
  IF SOME [found1, s2] DeleteFirst(F(label, occ, foo, found1), sin, s2)
    THEN  (
          IF SOME [s, f11] DeleteFirst(C(bar, s), found1, f11)
            THEN
              sin2 = [F(label, occ, foo, [C(bar, s+1)|f11])|s2]
            ELSE
              sin2 = [F(label, occ, foo, [C(bar, 1)|found1])|s2]
          )
    ELSE  sin2 = [F(label, occ, foo, [C(bar, 1)])|sin]
  ) &
  IF SOME [found, l2] DeleteFirst(F(label, found), list, l2)
    THEN  (
          IF SOME [n, f1] DeleteFirst(C(occ, n), found, f1)
            THEN
              GetCommits(r, [F(label, [C(occ, n+1)|f1])|l2], list1, sin2, sin1)
            ELSE
              GetCommits(r, [F(label, [C(occ, 1)|found])|l2], list1, sin2, sin1)
          )
    ELSE  GetCommits(r, [F(label, [C(occ, 1)])|list], list1, sin2, sin1).
```

`RenameCommits` calls `GetAllCommits` to collect the augmented commit labels in the residual code into sets which are partitioned by distinct augmented labels. These sets are of two kinds. In the first we group all labels of the form $n_a^{f:p}$ into partitions distinguished by the distinct n_i. In the second set we group all labels of the form $n_a^{f:p}$ into partitions distinguished by the distinct $n_i^{f:p}$. Calls to `SumCommits` and `SumCommitsS` then compute the sizes of these partitioned sets.

```
SumCommits([], []).
SumCommits([F(label, found)|rest], [S(label, found, sum, mods)|sums]) <-
  SumCommits1(found, [], mods, 1, sum) &
  SumCommits(rest, sums).
```

```
SumCommitsS([], s, s).
SumCommitsS([F(l, o, f, found)|rest], [S(l, o, f, found, s, m)|sums], sums1) <-
  SumCommits1(found, [], m, 1, s) &
  SumCommitsS(rest, sums, sums1).

SumCommits1([], mods, mods, sum, sum).
SumCommits1([C(_, n)|rest], mods, mods1, sum, n*sum1) <-
  IF n = 1
    THEN  SumCommits1(rest, mods, mods1, sum, sum1)
    ELSE  IF SOME [m, mods2] DeleteFirst(Mod(n, m), mods, mods2)
            THEN  SumCommits1(rest, [Mod(n, m+1)|mods2], mods1, sum, sum1)
            ELSE  SumCommits1(rest, [Mod(n, 1)|mods], mods1, sum, sum1).
```

Having grouped the commit labels into sets of common labels and computed the sizes of these sets the matrix of new integer labels for each augmented commit are then generated with a call to `RegulateCommits`. For each commit label `RegulateCommits` calls `Regulate` which uses a modulo-arithmetic method to generate labels which perform the relevant pruning on the groups of common labels.

```
RegulateCommits([], _, matrix, matrix).
RegulateCommits([S(l, fd, sum, mods)|r], z, mx, [M(l, reg)|mx1]) <-
  Regulate(fd, z, next, sum, mods, reg) &
  RegulateCommits(r, next, mx, mx1).
RegulateCommits([S(l, o, f, fd, sum, mods)|r], z, mx, [M(l, o, f, reg)|mx1]) <-
  Regulate(fd, z, next, sum, mods, reg) &
  RegulateCommits(r, next, mx, mx1).

Regulate([], _, 0, _, _, []).
Regulate([C(o, n)|rest], z, Max(next, next1), sum, mods, [Labels(o, ls)|mx]) <-
  (
  IF n = 1
    THEN  mods1 = mods &
          GenerateIntegers(z+1, z+sum, labels1, []) &
          ls = [labels1] &
          next1 = z+sum
    ELSE  DeleteFirst(Mod(n, m), mods, mods2) &
          mods1 = [Mod(n, m-1)|mods2] &
          GenerateLabels(n, (n^(m-1)*(n-1)), z, 1, sum, n^(m-1), ls) &
          next1 = z+n*m
  ) &
  Regulate(rest, z, next, sum, mods1, mx).
```

```
GenerateLabels(0, _, _, _, _, _, []) <-
  |.
GenerateLabels(n, mod, zero, occ, sum, m, [labels|rest]) <-
  n > 0 |
  GenerateIntegers(zero+occ, zero+occ+m-1, labels, labels1) &
  GenerateRest(occ+m+mod, zero, sum, m, mod, labels1, []) &
  GenerateLabels(n-1, mod, zero, occ+m, sum, m, rest).

GenerateRest(from, zero, sum, m, mod, labels, labels1) <-
  from =< sum |
  GenerateIntegers(zero+from, zero+from+m-1, labels, labels2) &
  GenerateRest(from+m+mod, zero, sum, m, mod, labels2, labels1).
GenerateRest(from, _, sum, _, _, labels, labels) <-
  from > sum |.

GenerateIntegers(n, n, [n|rest], rest) <-
  |.
GenerateIntegers(n, m, [n|rest], rest1) <-
  n < m |
  GenerateIntegers(n+1, m, rest, rest1).
```

Having determined the necessary renamings for the augmented labels in the specialisation of A, *SAGE* constructs the residual code for the partial evaluation of A with a call to `ReconstructResultants(r,m,[],s)`, where `r` are the residual resultants for A, `m` the matrix of new commit labels for these resultants and `s` the specialised code for A.

```
ReconstructResultants([], _, []).
ReconstructResultants([Res(hd, body, s, _, _, cs)|rest], matrix, [f|rest1]) <-
  ApplySubstToFormula(hd, s, head) &
  GetComs(cs, matrix, matrix1, labels) &
  ReconstructBody(body, s, labels, body1) &
  IsImpliedBy(head, body1, f) &
  ReconstructResultants(rest, matrix1, rest1).

GetComs([], matrix, matrix, []).
GetComs([C(label, occ)|rest], matrix, matrx1, [New(label, labels)|rest1]) <-
  DeleteFirst(M(label, labels1), matrix, m2) &
  GetLabels(labels1, labels2, occ, labels) &
  GetComs(rest, [M(label, labels2)|m2], matrx1, rest1).
GetComs([C(label, o, f, b)|rest], matrix, matrx1, [New(label, labels)|rest1]) <-
  DeleteFirst(M(label, labels1), matrix, m2) &
```

```
  GetLabels(labels1, labels2, o, labelsU) &
  DeleteFirst(M(label, o, f, labelsS1), [M(label, labels2)|m2], m3) &
  GetLabels(labelsS1, labelsS2, b, labelsS) &
  Append(labelsU, labelsS, labels) &
  GetComs(rest, [M(label, o, f, labelsS2)|m3], matrx1, rest1).
```

We give below sufficient of the definition of `ReconstructBody` to give an illustration of its implementation.

```
ReconstructBody(formula, subst, coms, formula1) <-
  And(l, r, formula) |
  ReconstructBody(l, subst, coms, l1) &
  ReconstructBody(r, subst, coms, r1) &
  AndWithEmpty(l1, r1, formula1).
ReconstructBody(formula, _, _, formula) <-
  EmptyFormula(formula) |.
ReconstructBody(atom, subst, _, atom1) <-
  Atom(atom) |
  ApplySubstToFormula(atom, subst, atom1).
ReconstructBody1(formula, subst, coms, formula1) <-
  Commit(label, f1, formula) |
  MemberCheck(New(label, labels), coms) &
  ReconstructBody(f1, subst, coms, formula2) &
  MakeCommits(labels, formula2, formula1).
  ReconstructBody(f1, subst, coms, formula1).

MakeCommits([], formula, formula).
MakeCommits([label|rest], formula, formula1) <-
  MakeCommits(rest, formula, formula2) &
  Commit(label, formula2, formula1).
```

Handling Freeness

As discussed in Section 2.3, the freeness condition essentially states that a partial evaluator should not be permitted to perform any pruning during a partial evaluation. We have described certain circumstances under which the freeness condition may be disabled. This is represented in *SAGE* by a flag which is set to the constant `Free` if freeness is enabled and `Prune` if pruning is permitted. This flag is set to `Free` by default.

Having unfolded a committed formula in a resultant R, *SAGE* will construct the new resultants with a call to `MakeCommit(r,n,m,m1,f,hd,lf,rt,rs,rs1)`, where `r` is the list of residual resultants for the partial evaluation of the committed formula, `n` is the augmented commit label for this

formula, `m` is an integer one greater than any appearing as a commit label in the current partial evaluation, `m1` the new value for `m` after this unfolding step, `f` is the free/prune flag, `hd` is the head of R, `lf` and `rt` respectively the formulas to the left and right of $\{F\}_n$ in the body of R and `rs` and `rs1` are respectively the list of resultants forming the nodes in the current partial evaluation before and after this unfolding step.

`MakeCommit` initially calls `SingleCommit`. If pruning is permitted then we may perform a pruning step if any of the residuals for the committed formula is the empty formula and no essential variable bindings have been made in that residual. In this case `SingleCommit` will return an integer pointer to one of these residuals. Otherwise this pointer is set to a null value, `-999`. If a pruning step cannot be performed then `MakeCommit` will call `ConstructCommit` to construct the new resultants, otherwise a call to `ExecuteCommit` will perform the relevant pruning step.

```
MakeCommit(residuals, C(n, i), m, m1, free, hd, lf, rt, res, res1) <-
  SingleCommit(free, residuals, catch) &
  IF  catch = -999
    THEN  ConstructCommits(residuals, n, i, m, 1, hd, lf, rt, res, res1) &
          m1 = m+1
    ELSE  ExecuteCommit(catch, residuals, n, i, hd, lf, rt, res, res1) &
          m1 = m.

SingleCommit(Free, _, -999).
SingleCommit(Prune, residuals, catch) <-
  SingleCommit1(residuals, 1, catch).

SingleCommit1([], _, -999).
SingleCommit1([Res(head, body, subst, _, _)|rest], n, catch) <-
  GetFormulaBindings(head, subst, bind) &
  IF EmptyFormula(body) & bind = []
    THEN  catch = n
    ELSE  SingleCommit1(rest, n+1, catch).

ExecuteCommit(1, [Res(_, _, subst, v, cs)|_], n, i, hd, lf, rt, res
             , [Res(h, body, subst, v, cs)|res1]) <-
  | AndWithEmpty(lf, rt, body) &
  Prune(res, n, i, res1).
ExecuteCommit(st, [_|rest], n, i, hd, lf, rt, res, res1) <-
  st > 1 |
  ExecuteCommit(st-1, rest, n, i, hd, lf, rt, res, res1).

Prune([], _, _, []).
Prune([Res(head, body, subst, var, coms)|rest], n, i, res) <-
```

```
  IF  Cut(coms, n, i)
    THEN  Prune(rest, n, i, res)
    ELSE  res = [Res(head, body, subst, var, coms)|res1] &
          Prune(rest, n, i, res1).

Cut([C(n, j)|_], n, i) <-
  i ~= j.
Cut([C(n, j, _, _)|_], n, i) <-
  i ~= j.
Cut([_|rest], n, i) <-
  Cut(rest, n, i).
```

4.4 Assembling The Residual Script

In Section 2.6 we discussed the reasons which required that the specialised version of a Gödel program be represented as a script. In this section we describe the postprocessing optimisations performed by *SAGE* and explain in more detail how these results are incorporated into the construction of the residual script.

Having performed the partial evaluation of a program, *SAGE* is left with a set of atoms and their partial evaluations. The following steps are performed in order to assemble the residual program from the original program and this set:

1. Convert original program to a script
2. Optimise residual code
3. Update `DELAY` declarations
4. Update predicate and proposition declarations
5. Replace original code with partially evaluated code

4.4.1 Converting Programs to Scripts

Converting the original program to a script, where `<program>` is the term representing the object program, is performed by the call `ProgramToScript(<program>,script)`. The predicate `ProgramToScript` is defined in the Gödel system module `Scripts`. Having created a script based upon the original program, *SAGE* is then able to optimise the residual code and perform any necessary modifications to declarations in the script before replacing the original code with the optimised, specialised code.

4.4.2 Optimising Residual Code

In general we expect to have constructed the partial evaluation of a set of atoms that define the specialisation of predicates in the original program to particular calls. Generally, certain arguments of these atoms are instantiated to non-variable terms before they are specialised. For example in a meta-interpreter with top level predicate `Demo` we may have specialised this interpreter to a particular object program by partially evaluating the atom `Demo(<program>,query,answer)`, where `<program>` is the term representing the object program. The representation of an object program will generally be a very large term and is likely to be redundant (not utilised) in the residual code. We may therefore delete this term. To delete such a term we would replace the ternary predicate `Demo` with a new binary predicate, `Demo_1` say, where any computed answer for `Demo_1(query,answer)` would be equivalent to that computed for `Demo(<program>,query,answer)`.

In addition to the above, non-ground non-variable terms in the partially evaluated atoms may also be removed from the residual code. If the meta-interpreter in the above example say, were specialised to a particular object program and a particular class of queries for that program, for example by specialising the atom `Demo(<program>,Atom(P',[x,y,z]),answer)`, where `Atom(P',[x,y,z])` is the representation of the atom `P(arg1,arg2,arg3)` with meta-variables in the place of its arguments, then instances of this `Demo` atom could be replaced by instances of the atom `Demo_1(x,y,z,answer)` in the specialised meta-interpreter.

We compute a pattern for the optimisation of a partially evaluated atom with a call to `OptimiseArgs(a,t,p,[],type)`, where `a` is the list of arguments of the atom in question, `t` the *variable typing* for these arguments (that is, the set of pairs of variables in this atom and the type inferred for that variable), `p` the computed optimising pattern for this atom and `type` the list of types of the new arguments for the optimised atom.

```
OptimiseArgs([], _, [], types, types).
OptimiseArgs([arg|rest], typing, [p|pattern], types, types1) <-
  IF  GroundTerm(arg)
    THEN  p = Del(arg) &
          OptimiseArgs(rest, typing, pattern, types, types1)
    ELSE  OptimiseTerm(arg, typing, p, types, types2) &
          OptimiseArgs(rest, typing, pattern, types2, types1).

OptimiseTerm(term, typing, Arg, [type|rest], rest) <-
  Variable(term) |
  BindingInVarTyping(typing, term, type).
OptimiseTerm(term, typing, F(functor, args), types, types1) <-
  FunctionTerm(term, functor, args1) |
  OptimiseArgs(args1, typing, args, types, types1).
```

The pattern computed for a list of arguments consists of a list of pattern terms. A pattern term is one of either `Del(term)`, indicating a ground term `term`, `Arg`, indicating a variable or `F(name,args)`, indicating a function term with name `name` and arguments the list of pattern terms `args`. For example, the computed pattern for the term `Atom(P',[x,y,z])` would be `F(Atom',[Del(P'),F(.',[Arg,F(.',[Arg,F(.',[Arg,Nil'])])])])`, where `.'` and `Nil'` were the names of the representations of the list constructor and the empty list respectively.

We optimise the residual code by replacing every occurrence of an instance of a partially evaluated atom (that is, any atom which matches the computed pattern) by an equivalent instance of the optimised atom. We perform this optimisation by first optimising the heads of the statements in the partial evaluation of the atom to be optimised. Next we optimise all instances of this atom in the bodies of the statements forming the residual code for the partial evaluation. We perform this optimisation with a call to `ChangeBody(b,c,b1)`, where `b` is the body of a residual statement, `c` the list of patterns for all the optimised atoms and `b1` the body of the optimised statement. We give an illustrative section of the code for `ChangeBody` here.

```
ChangeBody(body, changes, new_body) <-
  And(left, right, body) |
  ChangeBody(left, changes, left1) &
  ChangeBody(right, changes, right1) &
  And(left1, right1, new_body).
ChangeBody(body, changes, new_body) <-
  PredicateAtom(body, pred, args) |
  IF  SOME  [pattern, new_pred, new_args]
          ( Member(Change(pred, new_pred, pattern, _, _, _), changes) &
            ChangeArgs(args, pattern, new_args, []) )
          THEN  NewAtom(new_args, new_pred, new_body)
          ELSE  new_body = body.
ChangeBody(body, _, body) <-
  EmptyFormula(body) |.

ChangeArgs([], [], args, args).
ChangeArgs([arg|rest], [pattern|rest1], new_args, new_args1) <-
  ChangeTerm(pattern, arg, new_args, new_args2) &
  ChangeArgs(rest, rest1, new_args2, new_args1).

ChangeTerm(Del(term), term, new_args, new_args).
ChangeTerm(Arg, term, [term|new_args], new_args).
ChangeTerm(F(functor, pattern), term, new_args, new_args1) <-
  FunctionTerm(term, functor, args) &
  ChangeArgs(args, pattern, new_args, new_args1).
```

```
NewAtom([], prop, atom) <-
  PropositionAtom(atom, prop).
NewAtom([arg|rest], pred, atom) <-
  PredicateAtom(atom, pred, [arg|rest]).
```

The list of patterns for the optimised args is a list of terms of the form `Change(n,n1,p,_,_,_)`, where `n` is the name of a predicate, `n1` the new name for the optimised predicate atoms and `p` the pattern for the atoms to be optimised. The remaining arguments of these `Change` terms record information pertaining to the type and control declarations for the optimised atoms.

For example, the atom `Demo(<program>,Atom(P',[A',B',C']),answer)` in the above example, where `A'`, `B'` and `C'` are the representations of constants, would be replaced by `Demo_1(A',B',C',answer)`. As the set of partially evaluated atoms is independent we are assured that no atom in either the original program or the residual code will be an instance of more than one atom in the set of optimising patterns.

The advantages of performing such optimisations, which are relatively straightforward and simple to implement, are that removing redundant terms will significantly improve the performance of the specialised program by cutting down on unification and the use of heap-space and it will also permit argument indexing to a greater depth than in the original program. This latter advantage is illustrated by the previous example, where indexing may be performed upon the *arguments* in the representation of the query, as opposed to being performed upon the representation of the query.

4.4.3 Updating Declarations

Having performed the above optimisations *SAGE* must also update the predicate declaration and any matching `DELAY` declarations. These optimisations are performed by a call to `UpdateDelays(d,c,n,p,d1,c1)`, where `d` and `c` are respectively the (possibly empty) list of delays and the corresponding list of conditions for the atom to be optimised, `n` the new name of the optimised predicate atom, `p` the pattern for this optimised atom and `d1` and `c1` are respectively the new list of delays and the corresponding new list of conditions for the new atom.

```
UpdateDelays([], [], _, _, [], []).
UpdateDelays([atom|rest], [cond|rest1], pred, pattern, delays, conds) <-
  PredicateAtom(atom, _, args) &
  {
  IF SOME [args1, cond1] UpdateArgs(args, pattern, 0, _, args1, [], cond, cond1)
    THEN  NewAtom(args1, pred, atom1) &
          delays = [atom1|delays1] &
          conds = [cond1|conds1]
    ELSE  delays = delays1 &
          conds = conds1
```

```
  } &
  UpdateDelays(rest, rest1, pred, pattern, delays1, conds1).

UpdateArgs([], [], var, var, args, args, cond, cond).
UpdateArgs([arg|rest1], [pattern|rest2], var, var1, args, args1, cond, cond1) <-
  UpdateTerm(pattern, arg, var, var2, args, args2, cond, cond2) &
  UpdateArgs(rest1, rest2, var2, var1, args2, args1, cond2, cond1).

UpdateTerm(Del(term), term1, var, var, args, args, cond, cond1) <-
  EmptyTermSubst(empty) &
  UnifyTerms(term, term1, empty, _) &
  TermVariables(term1, vars) &
  DelVarsInCond(vars, cond, cond1).
UpdateTerm(Arg, term, var, var, [term|args], args, cond, cond).
UpdateTerm(F(functor, pattern), term, var, var1, args, args1, cond, cond1) <-
  IF  Variable(term)
    THEN  CreateArgs(pattern, var, var1, args, args1, vars, []) &
          UpdateCondition(cond, term, vars, cond1)
    ELSE  FunctionTerm(term, functor, args2) &
          UpdateArgs(args2, pattern, var, var1, args, args1, cond, cond1).
```

Where a ground argument has been removed from an atom, any variables in the corresponding argument in the delay may be removed from the condition. This is performed by a call to `DelVarsInCond(v,c,c1)`, where `v` is the list of variables to be deleted, `c` a condition and `c1` this condition with all occurrences of these variables removed.

```
DelVarsInCond([], cond, cond).
DelVarsInCond([var|rest], cond, cond1) <-
  UpdateCondition(cond, var, [], cond2) &
  DelVarsInCond(rest, cond2, cond1).

UpdateCondition(cond, var, vars, cond1) <-
  AndCondition(c1, c2, cond) |
  UpdateCondition(c1, var, vars, c3) &
  UpdateCondition(c2, var, vars, c4) &
  AndCondition(c3, c4, cond1).
UpdateCondition(cond, var, vars, cond1) <-
  OrCondition(c1, c2, cond) |
  UpdateCondition(c1, var, vars, c3) &
  UpdateCondition(c2, var, vars, c4) &
  OrCondition(c3, c4, cond1).
UpdateCondition(cond, var, vars, cond1) <-
```

```
  GroundCondition(var1, cond) |
  IF  var1 = var
    THEN  GrindVars(vars, cond1)
    ELSE  cond1 = cond.
UpdateCondition(cond, var, _, cond1) <-
  NonVarCondition(var1, cond) |
  IF  var1 = var
    THEN  TrueCondition(cond1)
    ELSE  cond1 = cond.

GrindVars([], cond) <-
  TrueCondition(cond).
GrindVars([var|rest], cond) <-
  GrindVars1(rest, var, cond).

GrindVars1([], var, cond) <-
  GroundCondition(var, cond).
GrindVars1([var|rest], var1, cond) <-
  GrindVars1(rest, var, cond2) &
  GroundCondition(var1, cond1) &
  AndCondition(cond1, cond2, cond).
```

If the pattern contains a function term then a call to `CreateArgs` recreates the function term from which this pattern is derived. This function term may then be compared against the corresponding term in the delay.

```
CreateArgs([], var, var, args, args, vars, vars).
CreateArgs([arg|rest], var, var1, args, args1, vars, vars1) <-
  CreateTerm(arg, var, var2, args, args2, vars, vars2) &
  CreateArgs(rest, var2, var1, args2, args1, vars2, vars1).

CreateTerm(Del(_), var, var, args, args, vars, vars).
CreateTerm(Arg, var, var+1, [term|args], args, [term|vars], vars) <-
  VariableName(term, "v", var).
CreateTerm(F(_, pattern), var, var1, args, args1, vars, vars1) <-
  CreateArgs(pattern, var, var1, args, args1, vars, vars1).
```

For example, if the atom `Demo(<program>,Atom(P',[x,y,z]),answer)` with declaration:

```
PREDICATE  Demo : Program * Formula * TermSubst.
DELAY      Demo(p,q,a)  UNTIL GROUND(p) & GROUND(q).
```

were optimised to produce the new atom `Demo_1(x,y,z,answer)` then *SAGE* will replace the above declaration with:

```
PREDICATE  Demo_1 : Term * Term * Term * TermSubst.
DELAY      Demo_1(x,y,z,a)  UNTIL GROUND(x) & GROUND(y) & GROUND(z).
```

In Section 2.2 we also described how polymorphic type declarations for predicates may also need to be specialised. This specialisation is performed by *SAGE* at this point, on both optimised and non-optimised predicates. The new type declaration for a partially evaluated atom may be inferred simply by computing the variable typing for the optimised version of the partially evaluated atom.

4.4.4 Assembling the Script

Having performed the above optimisations, *SAGE* next deletes the definitions of all predicates for which a partial evaluation of an atom with this predicate has been produced. The partially evaluated code and the relevant declarations, updated as above, are then inserted and finally the representation of this new script is output to the text file specified by the user.

4.5 Termination and Correctness of *SAGE*

In this final section we outline the proofs for the guaranteed termination of *SAGE* and the correctness of the partial evaluations performed by *SAGE*.

4.5.1 Termination of *SAGE*

By Theorem 4.2.2 a static analysis computed by *SAGE* will terminate. Part (a) of Theorem 4.2.2 guarantees that the subsequent partial evaluation computed by *SAGE* will be finite and the computation of this partial evaluation will therefore terminate. The post-processing optimisations described in the previous section are based upon a straightforward syntactic analysis of the residual code produced by the partial evaluation and a trivial examination of this process demonstrates its termination. These components are the sum of *SAGE* and therefore the execution of *SAGE* will terminate.

4.5.2 Correctness of *SAGE*

In Chapter 2, Theorem 2.3.2 extended Theorem 3.2 of [29] to cover the non-regular unfoldings of c-programs performed by *SAGE*. Theorem 3.2 is the extension of Theorem 4.3 of [47] from normal programs to c-programs. We now prove that the partial evaluations computed by *SAGE* meet the preconditions of Theorem 2.3.2.

The preconditions of Theorem 2.3.2 state that for *SAGE*'s partial evaluation of a goal G wrt a program P to be correct, *SAGE* must compute a partial evaluation, P', of a finite, independent set of atoms $\mathbf{A}$ such that $P' \cup G$ is $\mathbf{A}$-covered.

Input

A c-program P

A goal G

A set of selectable predicate symbols L which is well-structured wrt P

Output

P', the partial evaluation of P wrt the L-selectable atoms in G

Initialisation

Find the set $\mathbf{A}'$ of L-selectable atoms in G.

% These atoms may appear in either a positive or negative literal in G.

While there exist a pair of atoms A_1, A_2 in $\mathbf{A}'$ such that

A_1 and A_2 have a common instance **do**

$\mathbf{A}' := (\mathbf{A}' - \{A_1, A_2\}) \cup \{A\}$, where A is the most specific generalisation of A_1 and A_2

EndWhile

Compute the static analysis of $\mathbf{A}'$ wrt P and L returning S, a safe subset of L,

and $\mathbf{A}''$, the set of covering patterns for the unsafe predicates

$\mathbf{A} := (\mathbf{A}' - \{A : A \in \mathbf{A}'$ and A has an unsafe predicate symbol$\}) \cup \mathbf{A}''$

Compute the partial evaluations of $\mathbf{A}$ wrt P, L and S

$P^* := P - \{C : C$ is the definition of predicate p and $p \in L\}$

Assemble P' as the residual script based on P^* and the above partial evaluations.

Figure 4.9: Algorithm for *SAGE*

Figure 4.9 outlines the top-level algorithm of *SAGE* for the specialisation of a goal G wrt a c-program P and set of selectable predicates L which is well-structured wrt P.

The goal G is finite and any pairs in $\mathbf{A}'$ which have a common instance are replaced by their msg. By Theorem 4.2.2 the static analysis will terminate returning a finite set of atoms, each with a distinct predicate symbol. It is trivial to see therefore that $\mathbf{A}$ will be a finite independent set of atoms. We now show that the partial evaluation P' computed by *SAGE* is such that $P' \cup G$ is $\mathbf{A}$-covered.

Let P^* be the subprogram of P' consisting of the definitions of predicate symbols in P' upon which G depends. $P' \cup G$ will be $\mathbf{A}$-covered iff $P^* \cup G$ is $\mathbf{A}$-closed. If an atom in $\mathbf{A}$ has a safe predicate symbol then its only instances in $P^* \cup G$ will be in G. It is trivial to see that all atoms in G with this predicate symbol will be instances of A. If an atom A in $\mathbf{A}$ has an unsafe predicate symbol then by part (b) of Theorem 4.2.2 A is a covering pattern for this predicate and therefore every atom in $P^* \cup G$ with this predicate is an instance of A.

The correctness of *SAGE* now follows directly from Theorem 2.3.2.

Chapter 5

Results and Conclusions

In this final chapter we present the results obtained by specialising a range of meta-programs with *SAGE*, including *SAGE* itself. We then discuss the contributions that this thesis makes to logic programming and its relation to other work in the field. Finally we suggest profitable areas for future research.

5.1 Results

In this section we present the results of specialising Gödel meta-programs with *SAGE* by comparing the runtimes of unspecialised and specialised programs. We deal with each of the Futamura projections in turn.

5.1.1 The First Futamura Projection

For the first Futamura projection we have used *SAGE* to specialise a theorem-prover, a simple SLD-interpreter and a coroutining interpreter to a range of object programs.

The `ModelElimination` Theorem Prover

The program `ModelElimination` is a model elimination theorem prover simplified for the definite case. This program was based upon a similar theorem prover developed by de Waal in [15].

As the Gödel system module `Theories` was not fully implemented at the time of writing this program, we have represented a theory via the representation of a Gödel program. For example, a sentence S in a theory T may be represented as the statement `Hook <-`S in the program representing T. We may represent arbitrary first-order sentences S in this way as Gödel allows arbitrary formulas as the bodies of statements. In `ModelElimination` the three predicates `Prop`, `Clause` and `NegGoal` were used in place of the proposition `Hook`.

```
MODULE       ModelElimination.

IMPORT       Model, ProgramsIO.

PREDICATE  Go : String * String * Integer * String * String.

Go(prog_string, goal_string, depth_bound, answer_string, proof_string) <-
    FindInput(prog_string++".prm", In(stream)) &
    GetProgram(stream, theory) &
    MainModuleInProgram(theory, module) &
    StringToProgramFormula(theory, module, goal_string, [goal]) &
    StringToProgramFormula(theory, module, "NegGoal(a)", [neg_goal]) &
    StringToProgramFormula(theory, module, "Clause(a)", [clause]) &
    StringToProgramFormula(theory, module, "Prop(a)", [prop]) &
    EmptyTermSubst(sub_in) &
    EmptyFormula(pr) &
    StandardiseFormula(goal, 0, var, goal1) &
    Demo(theory, module, neg_goal, clause, prop, goal1, depth_bound,
         sub_in, sub_out, var, _, pr, proof) |
    ApplySubstToFormula(goal1, sub_out, answer) &
    ProgramFormulaToString(theory, module, answer, answer_string) &
    ProgramFormulaToString(theory, module, proof, proof_string).
```

```
EXPORT       Model.

PREDICATE  Demo : Program * String * Formula * Formula * Formula * Formula
                 * Integer * TermSubst * TermSubst * Integer * Integer
                 * Formula * Formula.
```

```
LOCAL        Model.

IMPORT       Programs.

Demo(_, _, _, _, _, r, _, s, s, i, i, pr, pr) <-
    EmptyFormula(r)|.
Demo(ot, m, ng, c, p, r, db, s_in, s_out, i_in, i_out, pr_in, pr_out) <-
    And(left, right, r) |
    Demo(ot, m, ng, c, p, left, db, s_in, s1, i_in, i1, pr_in, pr1) &
    Demo(ot, m, ng, c, p, right, db, s1, s_out, i1, i_out, pr1, pr_out).
Demo(ot, m, ng, c, p, r, db, s_in, s_out, i_in, i_out, pr_in, pr_out) <-
    db > 1 &
    Atom(r) |
    Theory(ot, m, ng, c, p, r, new_r, db-1, s_in, s1, i_in, i1, pr_in, pr1) &
    IF  EmptyFormula(new_r)
      THEN
        s_out = s1 &
        i_out = i1 &
        pr_out = pr1
      ELSE
        Demo(ot, m, ng, c, p, new_r, db-1, s1, s_out, i1, i_out, pr1, pr_out).

PREDICATE  Theory : Program * String * Formula * Formula * Formula * Formula
                    * Formula * Integer * TermSubst * TermSubst * Integer
                    * Integer * Formula * Formula.

Theory(ot, m, _, _, p, r, new_r, db, s_in, s_out, i_in, i_out, l, l_1) <-
    db > 1 &
    StatementMatchAtom(ot, m, p, hook) &
    IsImpliedBy(proof, axiom, hook) &
    SplitHeadBody(axiom, head, new_r1) &
    IsImpliedBy(head, new_r1, st) &
    Resolve1(st, r, i_in, i_out, s_in, s_out, new_r) &
    AndWithEmpty(l, proof, l_1).
Theory(ot, m, _, c, _, r, new_r, db, s_in, s_out, i_in, i_out, l, l_1) <-
    db > 2 &
    StatementMatchAtom(ot, m, c, hook) &
    IsImpliedBy(proof, axiom, hook) &
    SplitHeadBody(axiom, head, new_r1) &
```

```
    IsImpliedBy(head, new_r1, st) &
    Resolve1(st, r, i_in, i_out, s_in, s_out, new_r) &
    AndWithEmpty(l, proof, l_1).
Theory(ot, m, ng, _, _, r, new_r, db, s_in, s_out, i_in, i_out, l, l_1) <-
    db > 2 &
    StatementMatchAtom(ot, m, ng, hook) &
    IsImpliedBy(proof, axiom, hook) &
    SplitHeadBody(axiom, head, new_r1) &
    IsImpliedBy(head, new_r1, st) &
    Resolve1(st, r, i_in, i_out, s_in, s_out, new_r) &
    AndWithEmpty(l, proof, l_1).

PREDICATE  SplitHeadBody : Formula * Formula * Formula.

SplitHeadBody(axiom, head, body) <-
    IF  Literal(axiom)
    THEN
        EmptyFormula(body) &
        head = axiom
    ELSE
      And(h, b, axiom) &
      SplitHeadBody(h, head, body1) &
      And(body1, b, body).

PREDICATE  Resolve1 : Formula * Formula * Integer * Integer
                      * TermSubst * TermSubst * Formula.

Resolve1(st, r, i_in, i_out, s_in, s_out, new_r) <-
  Atom(r) |
  Resolve(r, st, i_in, i_out, s_in, s_out, new_r).
Resolve1(st, r, i_in, i_out, s_in, s_out, new_r) <-
  Not(not_r, r) |
  IsImpliedBy(head, body, st) &
  Not(not_head, head) &
  IsImpliedBy(not_head, body, not_st) &
  Resolve(not_r, not_st, i_in, i_out, s_in, s_out, new_r).
```

When this program was specialised with respect to a given object theory all predicates other

than StandardiseFormula, ApplySubstToFormula and the WAM-like predicates were marked as being selectable in the partial evaluation. The predicate Demo was marked as unsafe by *SAGE*'s static analysis.

The SLD Interpreter

```
MODULE      SLD.

IMPORT      Solve, ProgramsIO.

PREDICATE   Demo : String * String * String.

Demo(program_string, body_string, answer_string) <-
           FindInput(program_string++".prm", In(is)) &
           GetProgram(is, program) &
           MainModuleInProgram(program, module) &
           StringToProgramFormula(program, module, body_string, [body]) &
           StandardiseFormula(body, 0, var, body1) &
           EmptyTermSubst(empty_substitution) &
           Solve(body, program, var, _, empty_substitution, answer) &
           ApplySubstToFormula(body1, answer, answer_body) &
           ProgramFormulaToString(program, module, answer_body, answer_string).
```

The program SLD is a simple SLD-interpreter for definite Gödel programs which do not import any system modules. It imports the module Solve which contains the code presented in Figure 3.3.

When SLD was specialised with respect to a given object program all predicates other than StandardiseFormula, ApplySubstToFormula and the WAM-like predicates were marked as being selectable in the partial evaluation. The predicate Solve was marked as unsafe by *SAGE*'s static analysis.

The Coroutine Interpreter

The program Coroutine is a more sophisticated SLD-interpreter which emulates a coroutining behaviour by interpreting the DELAY declarations for the object program that it interprets. The predicate CanRunAtom determines whether an atom matches its corresponding delay declarations. CanRunAtom corresponds closely to the system predicate RunnableAtom, the major difference being that CanRunAtom may be specialised by *SAGE* with respect to the delay declarations for a given

```
MODULE      Coroutine.

IMPORT      Conc, ProgramsIO.

PREDICATE   Go : String * String * String.

Go(program_string, body_string, answer_string) <-
           FindInput(program_string++".prm", In(is)) &
           GetProgram(is, program) &
           MainModuleInProgram(program, module) &
           StringToProgramFormula(program, module, body_string, [body]) &
           StandardiseFormula(body, 0, var, body1) &
           EmptyTermSubst(empty) &
           EmptyFormula(e) &
           MySucceed(body1, program, var, _, e, e, empty, answer) &
           ApplySubstToFormula(body1, answer, answer_body) &
           ProgramFormulaToString(program, module, answer_body, answer_string).
```

object program. This specialisation is similar to the specialisation of `Resolve` wrt to the statements in an object program.

```
EXPORT      Conc.

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

```
LOCAL       Conc.

IMPORT      Programs.

BASE Bind.

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

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

MySucceed(formula, _, var, var, _, _, answer, answer) <-
           EmptyFormula(formula) |.
MySucceed(formula, program, var, var1, l, r, answer_so_far, answer) <-
           Atom(formula) |
           StatementMatchAtom(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  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) <-
  IF  SOME [var] GroundCondition(var, cond)
    THEN  Member(var ! term, subst) &
          GroundTerm(term)
    ELSE
      IF SOME [cond1,cond2] OrCondition(cond1, cond2, cond)
        THEN OrConditionSatisfied(cond1,cond2, subst)
        ELSE
          IF SOME [cond1,cond2] AndCondition(cond1, cond2, cond)
            THEN  ConditionSatisfied(cond1, subst) &
                  ConditionSatisfied(cond2, subst)
            ELSE
              IF  SOME [var] NonVarCondition(var, cond)
                THEN  Member(var ! term, subst) &
                      NonVariable(term)
                ELSE  IF  TrueCondition(cond)
                        THEN  True
                        ELSE  False.
```

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

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

When `Coroutine` was specialised with respect to a given object program all predicates other than `StandardiseFormula`, `ApplySubstToFormula`, `ApplySubstToTerm`, `GroundTerm`, `NonVariable` and the WAM-like predicates were marked as being selectable in the partial evaluation. The predicates `MySucceed` and `Select` were marked as unsafe by *SAGE*'s static analysis.

Results

The table in Figure 5.1 indicates the speedups computed by comparing the execution times of the unspecialised meta-programs and their specialised versions. That is to say, the first column of figures gives the execution time of the query $I(P, Q)$, for a meta-program I, object program (or theory) P and query Q. The second column of figures gives the execution time for the query $I_P(Q)$, where I_P is the specialised program *SAGE* produces when specialising I to P. The last column gives the speedup computed by comparing the execution times of $I(P, Q)$ and $I_P(Q)$.

In this table we present the results of specialising the theorem prover `ModelElimination` with respect to two small definite theories, specialising the interpreter `SLD` with respect to a program for matrix transposition and a program which computes Fibonacci numbers and specialising the interpreter `Coroutine` with respect to a sorting program that uses the British Museum (or permutation) sort and a program for computing solutions to the eight queens problem. As an attempt to illustrate the potential speedups for larger object programs, such as meta-programs, which have a significant proportion of variable arguments in the heads of statements we added five redundant arguments to the statements in the Fibonacci program.

To give an indication of the relative speedups achieved by specialising the ground representation in the manner described in Chapter 3, Table 5.1 indicates the speedups gained for those parts of the relevant meta-program which perform interpretation. That is, the initial calls to `Demo`, `Solve` and `MySucceed` for `ModelElimination`, `SLD` and `Coroutine` respectively.

For the two meta-interpreters `SLD` and `Coroutine` we have been able to estimate that the interpreted programs (`Transpose`, `Fibonacci`, `BM-Sort` and `EightQueens`) execute at between 100 and 200 times slower than when they are executed at the object level. In the results described

Example Program ($I : P$)	Runtime		Speedup
	$I(P, Q)$	$I_P(Q)$	
Model Elimination: T_1	22.56s	0.29s	77.79
Model Elimination: T_2	26.19s	0.35s	74.83
SLD: Transpose (8x8)	2.94s	0.14s	21.00
SLD: Transpose (8x16)	5.80s	0.23s	25.21
SLD: Fibonacci (10)	11.68s	0.13s	89.85
SLD: Fibonacci (15)	118.34s	1.13s	104.73
SLD: Fibonacci (17)	347.85s	2.84s	122.48
Coroutine: BM-Sort(7)	2.98s	0.14s	21.29
Coroutine: BM-Sort(13)	14.08s	0.52s	27.08
Coroutine: EightQueens	5.12s	0.21s	24.38

Figure 5.1: The First Futamura Projection

by the above table we have found that a corresponding comparison for the *specialised* interpreters indicates that they will execute at approximately 4-6 times slower for `SLD` and 7-8 times slower for `Coroutine` when compared to the relevant object programs.

The main residual expense in the specialised interpreters is incurred by the residual calls to the WAM-like predicates. The current implementation of these predicates is reasonably efficient, allowing us to produce specialised meta-programs with an execution time comfortably within one order of magnitude of the execution time of the object programs which they interpret. It is conceivable that with a suitable implementation of the WAM-like predicates the above results could be improved yet further, leading to specialised meta-programs which would interpret object programs in a time comparable to executing them directly at the object level. We describe the current implementation of the WAM-like predicates in Appendix A.

5.1.2 The Missing Link

Specialising meta-programs by the first Futamura projection can be relatively straightforward and indeed has been performed many times before in a range of languages. The meta-programs specialised in the previous section are relatively simple programs and therefore not particularly difficult to specialise. When we consider the self-application of *SAGE* we are specialising a meta-program that consists of approximately five thousand lines of Gödel code and the task is therefore much more difficult.

An additional complication in the self-applicability of *SAGE* has been the fact that Gödel is a

language that is still under development. In fact the development of *SAGE* has often run ahead of the implementation of Gödel. For example, the original implementation of *SAGE* was completed long before the ground representation was implemented in Gödel and therefore *SAGE* could not be tested for some time after it was first implemented.

The implementation of the ground representation proceeded in stages with the first simple meta-programs being able to execute by May of 1992. The first implementation of the ground representation was extremely slow, mostly due to the inefficient implementation of the representation of substitutions. It was at this time that the current implementation of substitutions described in Appendix A was developed.

The first implementation of the ground representation did not support the system predicates `Compute` and `ComputeAll`, which are necessary to specialise closed Gödel code. Consequently several months passed before Gödel would allow *SAGE* to specialise any programs which either imported Gödel system modules or used the equality predicate, `=`. We shall discuss below further problems that have arisen from the implementation of `Compute`.

The system module `Scripts` supports the representation of the residual scripts constructed from a partial evaluation. However, this module was not implemented until May of 1993 and until that time the results of a partial evaluation performed by *SAGE* had to be constructed by hand into something resembling an executable Gödel program. Obviously this was impractical when the partially evaluated program was of a significant size and therefore we have only relatively recently been able to experiment with the self-application of *SAGE*.

To date, complications such as those above have, frustratingly, prevented us from testing fully the self-application of *SAGE*. Instead a cut-down version of *SAGE* has been implemented. The main differences between this partial evaluator and *SAGE* are that the smaller partial evaluator:

- uses a simplified version of *SAGE*'s static analysis
- does not specialise committed formulas in the manner described in Chapter 2
- does not specialise conditional formulas to their fullest extent.

We refer to this smaller partial evaluator as $SAGE^C$. In the next section we describe in more detail the differences between $SAGE^C$ and *SAGE* and present our preliminary results in the self-application of $SAGE^C$.

Before we present our results for the second and third Futamura projections we discuss the three main deficiencies of the current implementation of Gödel which affect the self-application of *SAGE*. The first is the fact that the current implementation of Gödel does not support the full commit. The second is the residual inefficiencies that can be caused by Gödel's delay conditions and, particularly, its implicit delays. The third is the implementation of the Gödel system predicate `Compute` and the affect that this has on partial evaluation.

The Full Commit

The current implementation of Gödel does not support the full commit operator. Commits in Gödel programs and scripts are therefore restricted to bar commits and single-solution commits only. Consequently *SAGE* cannot implement the unfolding of committed formulas in the manner of Section 2.3. As a result we have implemented a reduced version of *SAGE*, $SAGE^C$, which simply removes all commits from the residual code to ensure correctness.

Although $SAGE^C$ removes commits from the residual code, this has little effect in the residual scripts for the meta-programs in the previous section. In these programs the commits are used exclusively to prune failed computation when a choicepoint has been introduced by the use of the ground representation. For example, when the definition for the predicate `VarsInTerm1` from Chapter 3 with commits added:

```
VarsInTerm1(term,vars,[term|vars]) <-
  Variable(term) |.
VarsInTerm1(term,vars,vars) <-
  ConstantTerm(term,name) |.
VarsInTerm1(term,vars,vars1) <-
  FunctionTerm(term,name,args) |
  VarsInTerm2(args,vars,vars1).
```

is specialised, $SAGE^C$ produces the residual definition:

```
VarsInTerm1(Var(v,n),vars,[Var(v,n)|vars]).
VarsInTerm1(CTerm(name),vars,vars).
VarsInTerm1(Term(name,args),vars,vars1) <-
  VarsInTerm2(args,vars,vars1).
```

In the original definition for `VarsInTerm1` the commits were used to prune mutually exclusive cases. In the residual definition the symbols in the ground representation have been promoted into the heads of the statements and the Gödel system is able to perform first-argument indexing on this definition. Consequently the commits are now superfluous and the efficiency of the residual code has not been impaired by their removal.

When we consider the self-application of $SAGE^C$ a significant proportion of the commits in $SAGE^C$ will be used in similar situations to that described above. Therefore the removal of these commits does not impair the efficiency of the specialised code for $SAGE^C$. However, not all of the commits in $SAGE^C$ are used in this way. For example, at several stages $SAGE^C$ records variable bindings as a list of pairs `Var("v",n) ! t`. As at most one binding is recorded per variable, a call `Member(Var("v",1) ! value,list)` for example, where `list` is a ground list, can have at most one solution. However, the definition of `Member` is such that all elements of the list will be tested even after the correct element has been found. To avoid this redundant computation we would

place this call within a single-solution commit. Once $SAGE^C$ is specialised however, this commit would be removed and the redundant computation would be re-introduced.

Until the full commit operator is implemented in Gödel and we are able to consider the self-application of *SAGE*, the results of the self-application of $SAGE^C$ will suffer due to the inefficiency of computing failing computations which could have been pruned in the original code.

Delays in Gödel

Delays in Gödel are of two kinds. The first are *explicit* delays provided by the `DELAY` declarations for a program. The second are *implicit* delays which appear most notably in negated formulas and conditionals. We shall discuss the impact that each of these kinds of delays has on the efficiency of residual code in turn.

SAGE (and $SAGE^C$) does not heed `DELAY` declarations for open code while computing partial evaluations. From the point of view of self-applicability the author-defined delays in *SAGE* have little or no effect on the results of its self-application. However, the closed code of any program cannot be specialised and thus the `DELAY` declarations for this code remains in the residual script. These delays are an expense which we could potentially significantly reduce by an analysis of the variable dependencies in a statement.

For example, consider the following definition in a specialised meta-program:

```
 ManyUnifies(term, subst, new_subst) <-
   UnifyTerms(term, T1, subst, subst1) &
   UnifyTerms(term, T2, subst1, subst2) &
   UnifyTerms(term, T3, subst2, subst3) &
   UnifyTerms(term, T4, subst3, subst4) &
   UnifyTerms(term, T5, subst4, new_subst).

DELAY  ManyUnifies(x,y,_) UNTIL GROUND(x) & GROUND(y).
```

where `T1-T5` were the ground representations of terms. The `DELAY` declaration for the system predicate `UnifyTerms` delays a call until the first three arguments are ground. In the above example the second argument to each call is already ground. The first argument is equivalent for each call and therefore need only be checked once to see if it is ground. Knowledge of the implementation of `UnifyTerms` tells us that a call will terminate with the fourth argument instantiated to some ground term. Consequently, if we could safely assume knowledge of the computation rule, we could produce an optimised definition for `ManyUnifies` in which the above `DELAY` declaration is the *only* delay that would affect the computation of such a call. The delays for the calls to `UnifyTerms` could be entirely removed.

When we consider that the representation of a substitution may potentially record several thousand variable bindings and that the terms that these variables are bound to may be extremely large and complex, we can see that an optimisation such as the example above could significantly reduce the computation time of the residual code.

Implicit delays appear most notably in negated formulas and conditionals. Gödel's current computation rule will delay a negated formula and the formula in the condition of a conditional until the free variables in that formula are all ground. This is a strong restriction which could be optimised in the above manner, analysing the dependencies of the relevant free variables to determine whether this delay could be either relaxed or removed. This would be particularly beneficial in the specialised code for a call to `ResolveAll`. In a call to `ResolveAll` the subsidiary calls to `Resolve` appear as the conditions of conditional formulas and thus will suffer from the expense of the implicit delays on conditionals. Previous work in the analysis of variable-dependencies [32, 48, 52, 53] may be applied here.

For a range of meta-programs we have estimated that in an unspecialised meta-program the expense of the explicit and implicit delays will cause that program to execute at approximately 15–20% slower than if there were no delays. When the meta-program is specialised a significant proportion of these delays remain in the residual code and their expense remains as a constant factor.

For example, let us say that a meta-program P executes in 115 seconds of cpu-time, of which some 15 seconds is caused by either explicit or implicit delays. We could specialise P to reduce by a factor of 25, say, the expense of using the ground representation. However, this reduction does not apply to the delays in P and therefore the runtime in cpu-seconds of the specialised version of P would be $(115 - 15)/25 + 15 = 19$ seconds. Thus the speedup for the specialised version of P appears to be only $115/19 = 6.1$ times (approximately).

As we have stated, the residual expense of author-defined `DELAY` declarations in *SAGE* is minimal. However, the expense of explicit delays in closed code and the implicit delays, particularly in conditional formulas, is a significant overhead in the residual code for the self-application of *SAGE*. Gödel's implementation of these explicit and implicit delays and the potential for their removal through some form of static analysis are therefore very important issues that must be addressed before the full potential of the self-application of *SAGE* may be realised.

`Compute` and Closed System Modules

The predicates `Compute` and `ComputeAll` were originally developed to support *SAGE* as complements to the predicates `Resolve` and `ResolveAll`. Just as `Resolve` performs renaming wrt the integers in its third and fourth arguments, so does `Compute` perform renaming wrt the integers in *its* third and fourth arguments. Similarly, where `Resolve` has the substitutions before and after

resolution as its fifth and sixth arguments, so does `Compute` have the substitutions before and after computation as *its* fifth and sixth arguments. Just as `ResolveAll` calls `Resolve` repeatedly to perform all possible resolutions, so does `ComputeAll` call `Compute` repeatedly to return all possible computations.

The major difference between `Resolve` and `Compute` is that `Resolve` may only be used on the open code of a Gödel program. `Compute` was originally implemented to address this deficiency by emulating the interpretation of some goal wrt a Gödel program. `ComputeAll` is used by *SAGE* to specialise closed atoms as far as possible.

To correctly emulate the Gödel computation rule `Compute` must respect any relevant delays when emulating the execution of a goal wrt some program. `Compute` implements this by returning as its seventh argument the last goal in the computation. This last goal will be empty if the original goal succeeded and non-empty if it floundered.

The implementation of `Compute` has been particularly problematical as it is a most sophisticated predicate and there are two aspects of this implementation which adversely affect the efficiency of the partial evaluations computed by *SAGE*. The first of these is to do with the implementation of most Gödel system modules as closed modules. The second is to do with the implementation of `Compute`, in particular in its handling of floundering goals.

The current implementation of `Compute` is extremely efficient and in fact can, for large computations, execute faster than a comparable set of calls to `Resolve` that interpret the same computation. However, `Resolve` may be specialised as described in Chapter 3 to produce a specialised version which executes considerably faster, as the results in Table 5.1 show. `Compute` is a predicate declared in a closed system module and therefore may not be specialised at all when it is not sufficiently instantiated to be executed. This means that residual calls to `Compute` in a specialised meta-program are as expensive as in an unspecialised meta-program. The execution of a computation with such a residual call to `Compute` is considerably slower than a comparable set of specialised calls to `Resolve` that interpret the same computation. Most meta-programs, particularly relatively large and sophisticated meta-programs such as *SAGE*, will have many calls with closed system predicates and the expense of emulating such calls with `Compute` will be seriously detrimental to the potential speedup of specialised versions of these meta-programs.

The second problem with `Compute` is in its handling of floundering goals, particularly goals containing constraints. Gödel has constraint-solving capabilities in the domains of integers and rationals and the interpretation of such constraints must be handled by `Compute`.

Consider for example the specialisation of `Resolve`. As described in Section 3.3, the renaming of variables is performed in `Resolve` by incrementing an integer value which records the maximum variable index in the current computation. When we specialise `Resolve` this value is unknown and renaming a variable will leave the residual term `Var("v",n+1)`, where `n` is the current maximum

variable index. Due to the manner in which `Compute` is currently implemented, when a goal which contains such a constrained term is executed by a call to `Compute` there are a number of undesirable side-effects. These side-effects will cause the substitution returned by `Compute` to add a superfluous variable binding and also add a superfluous binding of some new variable to a duplicate of the constrained term.

These side-effects do not affect the correctness of `Compute`, but in a large computation where many such constrained terms appear (such as in the self-application of *SAGE*) they will have two undesirable effects. The first is that subsequent calls to `Compute` will become more and more expensive as they need to handle more and more superfluous constrained terms. The second is that these terms can appear as superfluous constraints in the residual programs constructed from such partial evaluations. These residual programs, such as the results of the second and third Futamura projections presented below, will noticeably suffer from the expense of computing these many superfluous constraints. A yet more sophisticated implementation of `Compute` is needed to remedy this problem and this is currently under investigation.

5.1.3 The Second and Third Futamura Projections

In this section we present our preliminary results of the self-application of the partial evaluator $SAGE^C$. First we present an overview of $SAGE^C$ and its comparison to *SAGE*.

Reducing *SAGE* to $SAGE^C$

In the previous section we described how, as Gödel does not yet support the full commit, $SAGE^C$ removes the commits from specialised code. Next we describe the remaining two differences between $SAGE^C$ and *SAGE*. These are that $SAGE^C$ implements a simplified version of *SAGE*'s static analysis and that $SAGE^C$ does not specialise conditional formulas to their fullest possible extent.

The static analysis performed by *SAGE* seeks to achieve two main goals. Firstly to detect a subset of the selectable predicates which are unsafe in the partial evaluation. Secondly, for those unsafe predicates, to calculate a most specific generalisation of all atoms with this predicate symbol which occur in the partial evaluation, so that this most specific generalisation may be specialised.

Implementing the principles of the static analysis has proved to be a complex task which it has proved most difficult to satisfactorily test until relatively recently. The current implementation of the static analysis performed by *SAGE* has proved sufficient to deal with the meta-programs of the previous section. Unfortunately it is only recently that we have been able to apply this analysis to *SAGE* itself. We have discovered that the analysis computed for the self-application of *SAGE* is incorrect.

By the results of the previous chapter we know that the principles upon which *SAGE*'s static

analysis is based are sound and that it is therefore merely the *implementation* which is currently flawed. A pragmatic short-term solution to this problem has been for the user to supply the partial evaluator with the set of unsafe predicates prior to the static analysis being computed. A reduced analysis then needs only to compute the most specific generalisations of all atoms with these predicates.

Next we turn to $SAGE^C$'s approach to the specialisation of conditional formulas.

$SAGE$ specialises conditionals in the manner described in Chapter 2. First the condition is specialised. If it is possible to safely determine whether the condition has succeeded or failed then $SAGE$ will proceed to specialise the then-part or else-part respectively, discarding the redundant part. If the condition is insufficiently instantiated for such a decision to be made then $SAGE$ specialises the then-part and else-part and uses the results of this specialisation to construct the specialised version of the conditional.

$SAGE^C$ differs from the above method in the latter case, when the condition is insufficiently instantiated for it to be able to safely determine whether the condition has succeeded or failed. In this case $SAGE^C$ will *not* specialise the then-part and else-part, but will construct the specialised conditional using the specialised condition and unspecialised then-part and else-part.

The reason for this simplification is that we were unable to compute the self-application of a version of $SAGE^C$ which *did* implement $SAGE$'s full specialisation of conditionals. The self-application of this meta-interpreter failed because Gödel failed due to an error in SICStus Prolog [12], in which it is currently implemented.

The error in SICStus Prolog occurred during garbage-collection and caused a complete termination of the Gödel environment, including the current partial evaluation. When $SAGE^C$ was implemented *without* the full specialisation of conditionals then this problem did not appear and we were able to successfully perform the second and third Futamura projections.

Simplifying the specialisation of conditionals in the above manner had in fact only a slight impact on the self-application of $SAGE^C$. The majority of conditional formulas in $SAGE^C$ which were affected were of the form `IF` C `THEN x1=t1 &...& xn=tn ELSE x1=s1 &...& xn=sn` and thus the specialisation of the then-part and else-part would have had little effect anyway.

There was only one conditional formula in the entire code for $SAGE^C$ which caused a problem. The problem was that an atom with a non-recursive predicate appeared in the then-part of this conditional. When $SAGE^C$ was specialised wrt a meta-program I the definition of this predicate was deleted, as it would have been superfluous had the conditional been fully specialised. We then edited the residual program $SAGE^C_I$ to replace the definition of this predicate producing the correct compiler $SAGE^{C*}_I$. When the third Futamura projection was performed we repeated this edit to produce the compiler-generator $SAGE^{C*}_{SAGE^C}$. When this compiler-generator was used to specialise an interpreter I to produce the compiler $SAGE^C_I$ it was therefore necessary to repeat this edit to

	Runtime		
Example Program $(I : P)$	$SAGE^C(I, P)$	$SAGE^C_I(P)$	Speedup
Model Elimination: T_1	856s	263s	3.3
Model Elimination: T_2	902s	269s	3.4
SLD: Transpose	402s	113s	3.6
SLD: Fibonacci	424s	118s	3.6
SLD: QuickSort	557s	199s	2.8
Coroutine: BM-Sort	761s	253s	3.0
Coroutine: EightQueens	1186s	447s	2.7
Coroutine: QuickSort	676s	250s	2.7

Figure 5.2: The Second Futamura Projection

produce the correct compiler $SAGE^{C*}_I$ again.

We present the results of this self-application of $SAGE^C$ below.

The Second Futamura Projection

Figure 5.2 illustrates the results of the second Futamura projection, where we have specialised $SAGE^C$ wrt the three meta-programs in Section 5.1.1 to produce three compilers. This table illustrates the comparison in the time taken to produce compiled code between using the first Futamura projection and using these compilers. We have produced compiled code from the object programs and theories used in Table 5.1 and, in the case of the interpreters `SLD` and `Coroutine`, the quick-sort program appearing in Section 1.2. It should be emphasised that the results in this and the following tables indicate comparable times for the execution of the entire subject program. That is, unlike the results in Table 5.1 where we considered the timings for the execution of the interpretation phase of the meta-programs only, the results in this section consider also the preprocessing and postprocessing phases of the partial evaluator $SAGE^C$.

It is probably more accurate to compare the runtimes of programs for which the time taken for garbage-collection has been factored out. Table 5.3 repeats the timings from Table 5.2 with the times for garbage-collection removed. The figures in brackets indicate the number of garbage-collections that were performed in a computation.

The Third Futamura Projection

Figures 5.4 and 5.5 illustrate the results for the third Futamura projection. The compiler-generator was originally generated using an earlier version of `Compute` and took approximately 23 hours to

Example Program ($I : P$)	Runtime (GC's)		Speedup
	$SAGE^C(I,P)$	$SAGE_I^C(P)$	
Model Elimination: T_1	582s (240)	181s (73)	3.2
Model Elimination: T_2	623s (244)	184s (74)	3.4
SLD: Transpose	295s (96)	78s (32)	3.8
SLD: Fibonacci	308s (101)	84s (31)	3.7
SLD: QuickSort	397s (136)	140s (55)	2.9
Coroutine: BM-Sort	505s (243)	170s (77)	3.0
Coroutine: EightQueens	800s (312)	330s (107)	2.4
Coroutine: QuickSort	440s (184)	171s (73)	2.6

Figure 5.3: The Second Futamura Projection minus Garbage-Collection

Example Program (I)	Runtime		Speedup
	$SAGE^C(SAGE^C, I)$	$SAGE^C_{SAGE^C}(I)$	
Model Elimination	45.6hrs	11.9hrs	3.8
SLD	45.3hrs	11.8hrs	3.8
Coroutine	45.7hrs	11.9hrs	3.8
$SAGE^C$		27.2hrs	

Figure 5.4: The Third Futamura Projection

produce. When `Compute` was rewritten to remove a bug only those parts of the compiler-generator that were affected were regenerated. At this time we do not have accurate results for the time taken to achieve the third Futamura projection with the current implementation of `Compute`, although the figures in Tables 5.4 and 5.5 indicate that it will take around 100 hours (including garbage-collection).

We must emphasise at this point that, barring the as yet undiscovered bug in the implementation of *SAGE*'s static analysis, it is only improvements to the implementation of Gödel that are preventing us from replacing the above tables with tables that show both the self-application of *SAGE* rather than $SAGE^C$ and also speedups closer to those in Table 5.1. We hope that in the near future an improved implementation of those parts of the Gödel language described previously, that are causing the majority of the remaining computational expense, will at a single stroke both replace the above tables in this manner and also improve the execution times of unspecialised Gödel programs.

Example Program (I)	Runtime (GC's) $SAGE^C(SAGE^C, I)$	$SAGE^C_{SAGE^C}(I)$	Speedup
Model Elimination	25.2hrs (9645)	7.9hrs (4860)	3.2
SLD	25.0hrs (9636)	7.9hrs (4855)	3.2
Coroutine	25.1hrs (9652)	7.9hrs (4861)	3.2
$SAGE^C$		20.2hrs (5498)	

Figure 5.5: The Third Futamura Projection minus Garbage-Collection

5.2 Future Work

SAGE was designed as an experiment to prove that an effective self-applicable partial evaluator could be constructed in a logic programming language which supported meta-programming with a ground representation. To be effectively self-applicable it was necessary for *SAGE* to be able to effectively specialise all of the facilities of the Gödel language which it utilised. In effect, this meant that *SAGE* needed to be able to specialise the vast majority of Gödel's facilities.

That *SAGE* is self-applicable is beyond question. The results of the previous section provide a strong argument for the fact that *SAGE* has the potential to be effectively self-applicable. Without doubt the results obtained in Section 5.1.1 by specialising some simple meta-programs are exceptional and indicate that *SAGE*'s specialisation of the ground representation effectively removes the vast majority of the expense of handling substitutions explicitly. The subsequent results of Section 5.1.3 fall short of this high standard. Fortunately this is due not to any fundamental flaw in *SAGE*, but rather to the fact that there are still outstanding areas in the residual code produced by *SAGE* where there is a residual expense waiting to be removed. We have identified the major areas of potential expense in the residual for the self-application of *SAGE* as being the need for the full-commit operator, the expense of delays in Gödel system modules and the expense of the implicit delays in negated formulas and conditionals.

In this section we identify areas in which *SAGE*, or any future program transformer or program specialiser for the Gödel language, may be extended or improved. We also speculate on potential solutions or changes to the implementation of Gödel which will remove the remaining expense in the residual code of specialised Gödel programs.

5.2.1 Extending *SAGE*

SAGE is a partial evaluator based primarily on finite unfolding techniques. Other specialisation operations such as folding may be introduced, although the theory needs to be well-founded. The

major improvement that can be made to *SAGE* without introducing new specialisation techniques such as unfolding is to enhance the static analysis.

Two possible future extensions to the static analysis are discussed below. Note that these are both introduced as areas for investigation as opposed to deficiences in the current static analysis. We then discuss the specialisation of the set-processing facilities provided by Gödel.

Non-ground Terms

At the moment the static analysis performed by *SAGE* relies upon the checking of *ground* terms in atoms with recursive predicates. For the example meta-programs we have specialised to date this has proved to be sufficient. Nevertheless it would seem that extending this checking to all non-variable terms could potentially lead to yet greater effectiveness in the identification of a safe subset of the selectable predicates that is as large as possible.

Determinancy Analysis

In *SAGE* it is only the unsafe atoms which are not unfolded when partially evaluating. We have discussed in Section 3.4 how this restriction aids the avoidance of code explosion. Another potential cause for code explosion is the unrestricted unfolding of atoms which are non-deterministic. The static analysis could be extended to detect an unacceptable level of non-determinancy and flag such atoms as unsafe also. This would have the effect of preventing the unfolding of such atoms while allowing the definition of the matching predicate to be specialised independently. However, given that the ground representation is an abstract data-type, it is quite possible for the unfolding of an atom to produce a large number of resolvents, all bar one of which may cause the branch to fail at a later date. For example if we were to unfold the atom `VarsInTerm1(`$\mathcal{T}$`,v,v1)`, where $\mathcal{T}$ is some ground term, with respect to the definition:

```
VarsInTerm1(term,vars,[term|vars]) <-
  Variable(term) |.
VarsInTerm1(term,vars,vars) <-
  ConstantTerm(term,name) |.
VarsInTerm1(term,vars,vars1) <-
  FunctionTerm(term,name,args) |
  VarsInTerm2(args,vars,vars1).
```

we would produce three new resultants. However, as $\mathcal{T}$ is a ground term we may predict that at least two of these resultants will belong to a failing branch in this partial evaluation and will thus be removed later. With the ground representation as an abstract data type non-determinancy therefore becomes a much less simple matter to predict or detect.

Intensional Sets

SAGE can specialise all of the facilities of the Gödel language other than finite sets, provided as a Gödel data structure by the system module `Sets`. Gödel supports the representation of finite sets both as *extensional* set terms and as *intensional* set terms. The operations on extensional sets are provided by the closed module `Sets` and thus may be specialised as for other closed code. Intensional set terms are not specialised by *SAGE*, for the reasons described below, but we believe that the techniques used by *SAGE* to specialise negated formulas and conditionals may be extended in a straightforward manner to cover intensional set terms.

There is a complication caused by the use of intensional set terms. A program which imports the system module `Sets` may have intensional set terms which appear as arguments, or subterms of arguments, of atoms in the language of this program. This may cause a problem for meta-programming by virtue of the fact that formulas may now appear as subterms of terms in the representation of an object program.

For example, an intensional set term is of the form `{T : W}`, where `T` is a term and `W` is a formula. Intensional set terms may appear in statements in a program or in goals such as

```
<- x = {p : Likes(p, Tennis)}.
```

Here the formula `Likes(p,Tennis)` appears as a subterm of one of the arguments for the equality atom in the goal.

As formulas may now appear as subterms of arguments of other formulas, any meta-program which performed some form of processing of formulas in the language of an object program would potentially need to examine every subterm of every argument of any atom, rather than just examining atoms alone.

For example, the postprocessing optimisations performed by *SAGE* will remove superfluous terms in some atoms by replacing such atoms with equivalent new atoms. When the partial evaluation produced by *SAGE* is large we will need to examine a large number of atoms in the residual code to determine whether they may be replaced. When the object program makes use of intensional set terms we must now examine every subterm of every argument for each atom in the residual code to determine which atoms should be replaced. This will be significantly more expensive.

A solution has been suggested to the above problem. If Gödel were to implement the representation of intensional set terms in such a way that they only appeared as arguments of an equality atom, as in the above example, then it would be much less expensive to detect such terms. Once this facility is implemented a meta-program such as *SAGE* could make use of it to handle intensional set terms in a reasonably efficient manner.

5.2.2 Delays In Gödel

The expense of delays in residual scripts has already been highlighted. This is the major remaining expense in specialised Gödel programs and therefore merits further research. The possibility of an analysis based upon variable-dependencies or similar has already been discussed and it should be noted here that the benefits of such an analysis apply also to unspecialised Gödel programs, if not necessarily to the same extent.

5.2.3 The Full Commit

In Section 2.3 we described how *SAGE* uses the full commit to avoid the enforcement of the regularity condition when performing a partial evaluation. As was stated then, enforcing regularity is potentially both expensive and detrimental to the efficiency of the results of a partial evaluation. Unfortunately the current implementation of Gödel does not support the full commit and therefore the techniques described in Section 2.3 may not be used to their full effect. In addition, it is a simple matter to construct an example where the full commit becomes necessary to support even a single unfolding step in a partial evaluation.

It is possible that we may be able to retain some of the power of the commits in the original program through the judicious assemblage of a residual script. Consider the case where Gödel allows programs to use either the bar or one solution commit operators only, as in the current implementation. Assuming that a programmer uses only 'clean' commits for the sake of efficiency it is possible that we will see one of two cases in general. The first is that when a bar commit is used in the definition of some predicate then atoms with this predicate are only unfolded when they are sufficiently instantiated to allow the necessary pruning. In this case the commits are unnecessary in the residual script, as in the example in Section 5.1.2. If the atom is not sufficiently instantiated then it is hoped that this atom would not be unfolded into some other residual and therefore the profusion of full commits would not be allowed to occur. In general, we may not make these assumptions and it is certainly far from clear if we can avoid problems in this manner with the one solution commit. Consequently if we are prevented from making use of the full commit it is most important that this issue bear closer inspection to determine whether there is a reasonable class of programs for which we may retain bar and single solution commits only in the residual code.

5.2.4 Opening the System Modules

The majority of Gödel system modules are currently implemented as closed modules. In fact at present all system modules other than `Syntax` are closed modules. The justification for making the system modules closed is that it hides the details of the implementation of these modules from the

user.

In actuality there are only a small number of the system modules for which there is a strong argument that their current implementation should be hidden and there are even fewer for which *all* of their implementation needs to be hidden. Certain modules, `Syntax` included, have a part of their code which it would be advisable to hide and a part that could easily be made open.

It is be possible that the majority of system modules could be split into an open module which imported a smaller (possibly so small as to be unnecessary) closed module which contained the hidden part. This has been performed for the current implementation of `Syntax` which imports the module `Substs` that supports the implementation of term and type substitutions described in Appendix A.2.

Ideally all Gödel system modules should be made open. In this case any partial evaluator would be able to access the code for a far more substantial portion of Gödel programs than is currently possible. While a partial evaluator would not in all cases benefit from being able to specialise Gödel system code to a greater extent than at present, it is certain that this is an issue that should ideally be left as a choice to the *designer* of a program transformer for Gödel, rather than the Gödel system.

The effect of opening a more substantial portion of the code in the system modules would be to allow this code to be specialised by specialising the relevant calls to `Resolve` or `ResolveAll`. Closed code cannot be specialised in this way and must be interpreted by a call to `Compute` or `ComputeAll`. The current implementations of `Compute` and `ComputeAll` are both sophisticated and efficient. However, we assert that far greater efficiency could be obtained for a specialised Gödel program if these calls were replaced by calls to `Resolve` or `ResolveAll` and specialised in the manner described in Chapter 3.

5.3 Related Work

In this section we give a brief discussion of work related to *SAGE*'s partial evaluation techniques and self-application.

5.3.1 Partial Evaluation Techniques

SAGE is a partial evaluator based mainly on finite unfolding. Partial evaluation was first introduced into logic programming by Komorowski in [39] and partial evaluation by unfolding was put on a firm theoretical footing by Lloyd and Shepherdson in [47]. The other optimisations performed by *SAGE* use constructive negation and a post-processing reduction of terms. Constructive negation was first proposed by Chan and Wallace in [13]. The removal of redundant terms has been suggested several times, the most general presentation being by Gallagher and Bruynooghe in [22].

We have not attempted to introduce folding rules [37, 60, 63, 69] into *SAGE*, however recent results by Seki in [61] on the theory of folding could be used as a basis for a future extension of *SAGE*, as could techniques such as predicate polyvariance [11].

5.3.2 Automatic and Sound Finite Unfolding

SAGE is an *automatic* partial evaluator based upon finite unfolding in that, unlike many previous partial evaluators, *SAGE* does not rely upon annotations provided by the user to determine when to unfold. To our knowledge the first algorithm for automatic partial evaluation based upon finite unfolding which uses the results of [29] to prove its correctness is that given by Benkerimi in [3]. Benkerimi's unfolding strategy was shown by Bruynooghe *et al* in [10] to be insufficient to guarantee the termination of the partial evaluation. In that paper Bruynooghe *et al* present a dynamic strategy for selecting literals for unfolding based upon well-founded measures. This strategy has been incorporated into an algorithm for automatic partial evaluation by Martens *et al* in [49]. Experiments with a partial evaluator based on this algorithm were performed by Horváth in [31].

The partial evaluation algorithm for *SAGE* given in Figure 4.9 shares many similarities with those of [3, 49]. All three describe automatic strategies based upon finite unfolding for which correctness of the results is proved with respect to [29]. Proof of termination is given for the algorithms of both *SAGE* and [49]. All three algorithms guarantee the coveredness condition by ensuring that partial evaluation of the msg's of selectable atoms appearing in the leaf nodes of the SLDNF-trees (SLD-trees in the case of [49]) used to construct the partial evaluation are computed. The algorithms of [3, 49] achieve this by examining the leaf-nodes of these SLDNF-trees (or SLD-trees) and extracting selectable atoms. The msg's of these atoms and any atoms already partially evaluated which share the same predicate symbol are then computed. The partial evaluation of these msg's is then computed (unless it has already been computed). There is a drawback with this technique which is that several partial evaluations of successively more general atoms may be computed before the most general atom is reached.

SAGE differs from the algorithms of [3, 49] in its approach to identifying the msg's of selectable atoms which can appear in the leaf nodes of the SLDNF-trees used in computing a partial evaluation. The static analysis performed by *SAGE prior* to partial evaluation computes the set of unsafe selectable predicates and the msg's of all occurrences of atoms with these predicate symbols. As all safe selectable predicates are unfolded by *SAGE* we may guarantee that only atoms with non-selectable or unsafe selectable predicates will appear in the leaf nodes of the SLDNF-trees constructed by the subsequent partial evaluation. As the static analysis identifies msg's for the unsafe predicates, these are partially evaluated once only. The major advantage of this technique over those of [3, 49] is that the static analysis computes *abstract* partial evaluations. The computation of these abstract partial evaluations is significantly less expensive than the repeated computations

of successively more general partial evaluations as in [3, 49].

SAGE also differs from the algorithms of [3, 49] in the selection strategy it employs for selecting literals for unfolding. Both Benkerimi and Martens *et al* use a dynamic analysis of a resultant and its ancestors in order to select a literal for unfolding. In Chapter 4 we discussed the advantages for self-applicability of using a static strategy in preference to a dynamic strategy. *SAGE* uses the results of the static analysis to guide unfolding and the selection process is simplified to the extent that selection is merely a matter of selecting the leftmost safe, selectable literal.

5.3.3 Termination Analysis

The static analysis performed by *SAGE* identifies *safe* predicates for which atoms with these predicates are guaranteed not to cause non-termination by repeated unfolding. This analysis is similar to an abstract interpretation of the program and query to be partially evaluated. Abstract interpretation was originally proposed by Cousot and Cousot in [14]. Several frameworks for the abstract interpretation of logic programming languages have been proposed [8, 25, 36, 50, 65]. However, the static analysis cannot be described precisely in terms of such a framework and therefore we have given a self-contained description of it in Chapter 4.

Although we do not present *SAGE*'s static analysis in terms of a more general framework such as the above we make the comment that it shares similarities with other applications of abstract interpretation, particularly for mode, termination and flow-analysis of logic programs [17, 44, 48, 53, 73, 75]. Of these the most closely related is the work of Verschaetse *et al* [73, 75] who provide a framework for termination analysis based on level-mappings and specify sufficient conditions for termination. The outline of a system for automatic termination analysis is presented and the static analysis performed by *SAGE* can conceivably be described as being the application of similar techniques to a specific case (that is, ground meta-programs). Other techniques for automatic termination analysis have been presented by Plümer [57] and Ullman and Van Gelder [70].

5.3.4 Specialising Meta-Interpreters

The relationship between compilation and the partial evaluation of meta-interpreters was first proposed by Futamura in [20]. The potential applications of the partial evaluation of meta-interpreters are numerous in logic programming, for example in [9, 42, 55, 58, 66, 68, 71]. However, while partial evaluation is capable of removing the majority of the overheads associated with the ground representation, to date attention has focused mainly on the elimination of overheads in non-ground Prolog meta-programs.

While standard partial evaluation techniques such as unfolding can be applied directly to Gödel meta-programs which use a ground representation, this is not so simple for Prolog meta-programs

which use the non-ground representation. The partial evaluation of Prolog programs is complicated by the need to specialise the non-logical features of Prolog. This problem is most acute for non-ground Prolog meta-programs as Prolog's non-logical meta-programming features are generally the most difficult to specialise, particularly the `assert` and `retract` predicates. Gallagher [21] and Owen [56] both suggest techniques for specialising Prolog meta-predicates but we do not consider this work relevant to the specialisation of meta-programs which use a ground representation.

In Section 3.8 we laid the foundations of a methodology for meta-programming in which it is intended that the division between those parts of a meta-program which should be unfolded during partial evaluation and those which should be residual is relatively simple to deduce. The division of the code in a meta-program in this way is a significant issue in the specialisation of interpreters in any representation. Lakhotia and Sterling have also highlighted the division of meta-programs into two parts in [43], where they refer to these as *parsing* and *execution* phases. We have used our knowledge of the general structure of Gödel meta-programs, presented in Section 3.8, to simplify the partial evaluation strategy employed by *SAGE*. A similar technique was used by Levi and Sardu in [45] to simplify the implementation of their partial evaluator.

The key difference between the specialisation of ground and non-ground meta-programs lies in the need to specialise the explicit unification operations performed by ground meta-programs. The major motivation for *SAGE* as a partial evaluator for meta-programs was that it should be an effective self-applicable partial evaluator for the full Gödel language. In the following two sections we discuss related work in these areas.

5.3.5 Specialising Resolution

We have developed an implementation of the Gödel system predicate `Resolve` which is both efficient and may be specialised to produce residual code that is analogous to WAM-code.

The specialisation of an explicit unification operation was first investigated by Kursawe in [41], where the non-logical Prolog meta-predicates were used to force explicit unification of object-level terms. The specialisation of this unification algorithm to produce code comparable to WAM-code was presented as an illustration of a more general *methodology* for the development of a Prolog compiler by the specialisation of a meta-program. A similar approach has been taken by Nilsson in [54] where such a methodology is illustrated by the design of a resolution algorithm for propositional logic which can be specialised to produce residual code comparable to the WAM-instructions for the forward execution and backtracking of procedure calls.

An approach more closely related to ours was taken by de Waal and Gallagher [16] where a unification algorithm developed specifically for ground meta-programming is specialised with respect to partially known object-level terms.

5.3.6 Full Specialisation and Self-Application

SAGE is an automatic, effectively self-applicable partial evaluator for the full Gödel language. That is, *SAGE* is a partial evaluator which operates without user guidance for unfolding, can specialise all facilities of the Gödel language and can produce significantly faster residual code upon self-application. We note however, that while techniques for the partial evaluation of all facilities of Gödel have been developed and implemented, as yet not all of these techniques are fully supported by the current implementation of Gödel.

Attempts to build a self-applicable partial evaluator in Prolog are hampered by the non-logical features of that language and therefore generally only considered restricted subsets of the language. The difficulties in specialising the full Prolog language can be seen in [56] where Owen deduces no less than 18 rules (specialisation techniques) necessary to effectively specialise non-ground Prolog meta-interpreters. It can be seen that all but two of these rules concern the specialisation of the non-logical features of Prolog. The remaining two rules correspond to the computation of a mode-analysis and the partial evaluation of the msg's of atoms which appear in the residual code (comparable operations are performed by *SAGE*'s static analysis).

Techniques for specialising the cut and other non-logical Prolog features were first suggested by Venken [72] and Sahlin's Mixtus partial evaluator [59] implements sophisticated techniques to specialise full Prolog, although no attempt is made to specialise `assert`. However, Mixtus appears unsuitable for self-application due to the (often largely dynamic) complexity necessary to specialise the non-logical features of Prolog and the fact that Mixtus makes substantial use of `assert`.

The first effective self-applicable partial evaluator is the Mix system of Jones *et al* [35]. Being the first effectively self-applicable partial evaluator, Mix is of great significance and many of the strategies employed in the implementation of Mix are reflected in later approaches to self-applicability. As with *SAGE* and the self-applicable Prolog partial evaluators described below, Mix performed an abstract flow-analysis prior to specialisation and used the results to guide the specialisation process, although user-annotations were also required. Jones *et al* identified the importance of the choice of language in which a self-applicable partial evaluator should be implemented, stressing that it should be both sufficiently expressive to implement a non-trivial partial evaluator and simple to process. Mix was implemented in a pure LISP-like language with few operators and the advantage of using such a pure language is illustrated by the significant speedups gained after self-application.

The Prolog Mix partial evaluator of Fuller and Abramsky [19] similarly performed a static analysis of the program to be specialised. The importance to self-applicability of performing static analysis in preference to dynamic analysis was recognised, as was the importance of termination and correctness. Due to the difficulties of handling the non-logical features of Prolog, Prolog Mix considered only a restricted subset of the language. Prolog Mix is not fully automated as user-

annotations were required to guide the unfoldings and termination of the partial evaluation is not guaranteed. No comparison of timings for the results of the self-application of Prolog Mix were given in [19].

A comparable system has been developed by Fujita and Furukawa [18]. This system does not handle the non-logical features of Prolog and is not automated, relying upon user-annotations to guide unfolding. Speedups of up to two times are quoted for self-application, although no correctness or termination proofs are provided.

The most recent self-applicable partial evaluator, and the most similar to *SAGE*, is the Logimix system of Mogenson and Bondorf [34, 51]. Like *SAGE*, Logimix is based upon finite unfolding and uses an analysis to divide the program code into static and dynamic parts for specialisation. Logimix considers a subset of the Prolog language and relies upon user-annotations to guide unfolding.

Although Logimix uses a ground representation to represent object programs, resolution is implemented by reflection to the object level. This means that Logimix executes significantly faster than *SAGE*. However, resolution in Logimix cannot be specialised in the manner of resolution in Gödel meta-programs. The results quoted in [51] are impressive for the first Futamura projection, giving speedups of the order of 15 times. However the speedups after self-application are more modest, being only 1.78 times faster following the second Futamura projection and 1.35 following the third.

No proof of either the correctness or termination of Logimix is given and the current implementation of Logimix relies upon "side-effects" to record certain information during a partial evaluation. While the use of certain of Prolog's "side-effecting" predicates can be declarative this, as with Logimix's unification by reflection to the object-level, is subject to certain restrictions and thus cannot be assumed to be declarative without proof. *SAGE* differs from Logimix and the other partial evaluators above, being the only partial evaluator which is fully automatic, can specialise the full language in which it is implemented, is proved to be correct and terminating and is effectively self-applicable, in that the results after self-application show a significant improvement in execution time.

5.4 Contributions

In this final section we summarise the main contributions of this thesis.

5.4.1 Specialising Gödel Programs

In this thesis we have presented basic techniques for the specialisation of the facilities of the Gödel language. We have implemented the specialisation of negated formulas by constructive negation [13] and extended this to cover conditional formulas. The specialisation of Gödel programs has led to the

development of the concept of a *script* based on a Gödel program. Scripts are a necessary addition to the Gödel language for the purposes of program specialisation and transformation. Finally, we have examined the specialisation of Gödel's pruning operator, the commit, and extended the results of [29] by removing the regularity condition from Theorem 3.2 of that paper. These techniques have all been implemented in *SAGE* and may also be exploited by any future program specialiser or program transformer.

5.4.2 Implementing the Ground Representation

In seeking to specialise Gödel meta-programs we have developed an implementation for resolution in the ground representation which we argue is highly efficient. In addition, this implementation of resolution (via the Gödel predicates `Resolve` and `ResolveAll`) may be specialised with a simple unfolding strategy to produce residual code which can potentially execute in a time comparable to a similar resolution step performed at the object level.

The implementation of `Resolve` and `ResolveAll` is such that even after specialisation the implementation of the representation of substitutions is treated as an abstract data type. The advantages of this approach are that even *specialised* Gödel meta-programs are independent of any particular implementation of the representation of substitutions. We have also sought an efficient implementation of substitutions. The details of this implementation are presented in Appendix A.

We argue that this implementation of substitutions and the operations upon them is one of the major contributions of this thesis. The ground representation is receiving increasing recognition as being essential for serious applications of declarative meta-programming and yet the expense it incurs has long been considered an unacceptable hindrance to its use. With the above results we have demonstrated that a practical implementation of the ground representation has been achieved in which the expense of using a ground representation may be largely eliminated by a relatively simple program specialisation.

5.4.3 A Methodology of Meta-Programming

In [7] three main historical criticisms of using a ground representation for meta-programming were identified. These were that:

- the representation of object programs as terms is too complex;
- meta-programming in the ground representation is laborious, because large procedures are required to do simple things such as unifying object-level terms;
- object-level variables and their bindings must be handled explicitly, an overhead that makes the meta-program unacceptably slow.

The first of the criticisms is successfully answered in [7], where it was argued that presenting the ground representation as an abstract data type avoided the need for the user to have knowledge of its implementation. The third criticism has been successfully countered by the results of the previous section, where the expense of the ground representation has been largely removed by *SAGE*.

The Gödel meta-programming system modules `Syntax`, `Programs`, `Theories` and `Scripts` provide substantial support for meta-programming by providing predicates that support most if not all of the basic meta-programming operations and a range of the more sophisticated ones. We expect that greater familiarity with Gödel's meta-programming facilities and further application of them to serious meta-programming examples will serve to demonstrate that the second criticism is not valid for the Gödel language. *SAGE* is one such serious meta-programming application, being a self-applicable partial evaluator for full Gödel that has been successfully implemented in Gödel. The meta-programming framework presented in section 3.4 is sufficiently general to describe a substantial range of ground meta-programs and it is hoped that this will form the basis for a future methodology for meta-programming with the ground representation.

5.4.4 A Framework for Self-Applicability

We have applied the basic framework for meta-programs presented in section 3.4 to program specialisation, illustrating *SAGE* in terms of this framework. This framework has been enhanced to give a framework that is specific to program specialisers, while still being sufficiently general to cover a substantial range of partial evaluators. Using this framework we have explored the problems associated with the self-application of a partial evaluator and identified two pertinent issues. These are the advantage of the simplicity of a partial evaluator and the advantage of static computation versus dynamic computation, particularly with respect to the selection strategy for a partial evaluator. By emphasising this framework we hope to clarify similar ideas put forward during the previous work on self-applicability referenced in Section 5.3.

SAGE is an effectively self-applicable partial evaluator which exemplifies the advantages of concentrating on these two issues. In the first case we were able to make *SAGE* a relatively simple partial evaluator largely due to the fact that Gödel is, barring I/O and (sound) pruning, a purely declarative language and is thus easily specialised. In the second case by basing *SAGE*'s selection strategy mainly on a static analysis we were able to implement *SAGE* in a manner which was largely amenable to self-specialisation.

We argue that *SAGE* is a successful demonstration of our solutions to the problems associated with self-application. These techniques may therefore be used in the future to build potentially more sophisticated program specialisers than *SAGE* which would also be effectively self-applicable.

We argue that the implementation of an effectively self-applicable partial evaluator in Gödel,

combined with the specialisation of Gödel meta-programs described in Chapter 3, form the major contribution of this thesis. By implementing *SAGE* and the techniques necessary to support it in Gödel and demonstrating its success in achieving the three Futamura projections we have demonstrated both the power, potential and practicality of meta-programming with the ground representation and the success of Gödel as a practical, reasonably efficient and (mostly) declarative language for meta-programming.

5.4.5 Generating a Compiler-Generator

SAGE is an effectively self-applicable partial evaluator that has been implemented in Gödel. Using *SAGE* we have automatically produced a compiler-generator by the third Futamura projection. Producing the compiler-generator serves as perhaps the most significant example in demonstrating the success of *SAGE* as a partial evaluator for Gödel meta programs and it is also an extremely useful meta-program in its own right.

We refer to the compiler-generator $SAGE_{SAGE}$ as *CoGen*. *CoGen* is a tool that can considerably reduce the time taken to develop Gödel meta-programs. Consider a Gödel meta-program *I*. Using *CoGen* we are able to produce the compiler $SAGE_I$ in a relatively short time. This compiler may be used to remove the overheads inherent in the ground representation for a range of object programs. Specialising the meta-program *I* with respect to an object program *P* by using the compiler $SAGE_I$ will be significantly faster than specialising it by a call to *SAGE(I,P)*.

The compiler-generator *CoGen* may also be used in the production of a new compiler-generator. For example, if $SAGE^*$ were an enhanced version of *SAGE*, the compiler-generator $CoGen^*$ which specialised $SAGE^*$ to meta-programs to generate compilers, would be produced be a call to $CoGen(SAGE^*)$. Of course, $CoGen^* = SAGE_{SAGE^*}$ and not $SAGE^*_{SAGE^*}$. To produce the compiler-generator $SAGE^*_{SAGE^*}$ would require the third Futamura projection once more.

5.4.6 Conclusions

Finally we reflect on the aims that were presented in the introduction of this thesis. These were to:

1. develop techniques to allow the specialisation of the full Gödel language
2. develop an implementation and a methodology for meta-programming with the ground representation which was
   - efficient
   - amenable to specialisation
3. design and implement an effectively self-applicable declarative partial evaluator in Gödel.

The results of Section 5.1 have, it is hoped, demonstrated that in the development of *SAGE* we have succeeded in the first two of these aims and that we have made significant progress in achieving the third.

The experience of developing and testing *SAGE* has also convinced the author of the power Gödel gains by virtue of being a declarative programming language with considerable support for declarative meta-programming. We claim that this work provides strong support in arguing that Gödel provides a highly suitable language for the implementation of program transformation, specialisation and other meta-programming techniques, and also provides a highly suitable language in which to write the object programs which these meta-programs manipulate. In the near future we hope to see, in addition to a full version of *SAGE* and the compilers and compiler-generators it can produce, the emergence of Gödel meta-programs such as declarative debuggers and program control generators which will make Gödel an ideal environment for meta-programming.

Appendix A

The Implementation of Substitutions

In this appendix we present the Gödel code that uses the WAM-like predicates to implement `UnifyTerms`, `UnifyTypes` and `UnifyAtoms`. As has been stated before, the WAM-like predicates support the representations of substitutions as an abstract data type and are therefore independent of any particular implementation of substitutions. In Section A.2 we present Gödel's current implementation of the representation of substitutions. In Section A.3 we present the implementation of the WAM-like predicates that is supported by the substitutions of Section A.2.

A.1 `UnifyTerms`, `UnifyTypes` and `UnifyAtoms`

The unification of two terms, types or atoms with respect to a substitution is computed in a similar manner to the unification of an atom with the head of a statement in `ResolveAll` in section 3.3.2. The one significant difference is that while with `ResolveAll` we may be certain that variables in the statement do not appear in either the atom being resolved or in the current substitution, in `UnifyTerms`, `UnifyTypes` and `UnifyAtoms` we have no such guarantee. When `UnifyTerms` is used to unify two terms with respect to the current substitution the variables in *both* terms being unified may potentially appear in the other term and/or in the substitution. In order to deal with this possibility we firstly apply the current substitution to the first argument. Having done this we can then guarantee at least that variables in this new term will not be bound to any values in the substitution.

```
UnifyTerms(term1, term2, subst, subst1) <-
  ApplySubstToTerm(term1, subst, term3) &
  UnifyTerms0(term3, term2, subst ,subst1).

UnifyTerms0(Var(n, i), term, subst_so_far, new_subst) <-
  GetVariable(term, Var(n, i), subst_so_far, new_subst).
UnifyTerms0(Term(name,args), term, subst_so_far, new_subst) <-
  GetFunction(term, Term(name, args1), mode, subst_so_far, subst1) &
```

```
  TermOccurCheck(mode, term, args) &
  UnifyTermArgs(args, args1, mode, subst1, new_subst).
UnifyTerms0(CTerm(name), term, subst_so_far, new_subst) <-
  GetConstant(term, CTerm(name), subst_so_far, new_subst).

TermOccurCheck(Read, term, args) <-
  TermNotOccur(args, term).
TermOccurCheck(Write, _, _).

TermNotOccur([], _).
TermNotOccur([arg|rest], var) <-
  TermNotOccur1(arg, var) &
  TermNotOccur(rest, var).

TermNotOccur1(Var(s,i), var) <-
  var ~= Var(s,i).
TermNotOccur1(Term(_,args), var) <-
  TermNotOccur(args, var).
TermNotOccur1(CTerm(_), _).

UnifyTermArgs([], [], _, subst, subst).
UnifyTermArgs([arg1|rest1], [arg2|rest2], mode, subst, subst1) <-
  UnifyTerms1(arg1, arg2, mode, subst, subst2) &
  UnifyTermArgs(rest1, rest2, mode, subst2, subst1).

UnifyTerms1(Var(n, i), term, mode, subst_so_far, new_subst) <-
  UnifyKnownVariable(mode, term, Var(n, i), subst_so_far, new_subst).
UnifyTerms1(Term(name,args), term, mode, subst_so_far, new_subst) <-
  UnifyFunction(mode, term, Term(name, args1), mode1, subst_so_far, subst1) &
  TermOccurCheck(mode1, term, args) &
  UnifyTermArgs(args, args1, mode1, subst1, new_subst).
UnifyTerms1(CTerm(name), term, mode, subst_so_far, new_subst) <-
  UnifyConstant(mode, term, CTerm(name), subst_so_far, new_subst).
```

The definition of `UnifyTypes` is virtually identical to that for `UnifyTerms`, and so we do not need to present the code for it here. Having defined `UnifyTerms` the code for `UnifyAtoms` requires little extra.

```
UnifyAtoms(PAtom(name), PAtom(name), s, s).
UnifyAtoms(Atom(name, args), Atom(name, args1), subst, new_subst) <-
  UnifyingTermSubst(args, args1, subst, new_subst).
```

```
UnifyingTermSubst([], [], subst, subst).
UnifyingTermSubst([arg1|rest1], [arg2|rest2], subst, subst1) <-
  UnifyTerms(arg1, arg2, subst, subst2) &
  UnifyingTermSubst(rest1, rest2, subst2, subst1).
```

A.2 Representing Substitutions

Substitutions are generally used as a method of recording the variable bindings produced by a computation. These bindings are mainly constructed through calls to the unification predicates presented above and the resolution predicates presented in Chapter 3. These predicates all rely upon the WAM-like predicates to perform any necessary variable bindings. Gödel also supports the representation of type substitutions, whose implementation is virtually identical to the following implementation of term substitutions.

There are two main operations that are performed upon the variable bindings that comprise a substitution. These are the *application* of a particular variable binding and the *addition* of some new variable binding to a substitution.

A.2.1 Applying Variable Bindings

To apply a variable binding that appears in some substitution we must look up this binding to find what term the variable is bound to. The operation of looking up (or *accessing*) a variable binding can be very expensive without a suitably efficient representation of substitutions.

List-Substitutions

An obvious way of implementing substitutions in Gödel is as a list of variable bindings such as:

```
[ Var("x",0)! CTerm(A'), Var("y",0)! CTerm(B'), Var("z",0)! CTerm(C')]
```

which represents the substitution {`x/A,y/B,z/C`}. We refer to an implementation of this form as a *list-substitution*. Unfortunately the lookup operation for a representation of this form is unacceptably inefficient when there are more then a handful of variable bindings in a substitution. When specialising a meta-program with *SAGE*, substitutions may contain several thousand variable bindings and therefore a more efficient implementation is required.

Array-Substitutions

As described in Section 3.3, a variable with the name `x_n` (where `x` is a string and `n` an integer) is represented in Gödel by the term `Var("x",n)`. The string `"x"` is called the *root* of this variable and the integer `n` its *index*.

In Section 3.3 we described how new variables, introduced by the predicates `Resolve`, `StandardiseFormulas`, `RenameFormulas` and so on, are represented as `Var("v",n)`, where `n` is an integer greater than the index of any other variable in the current computation. The majority of variables bound in a substitution are likely to be of this form.

Considering the case where *all* variables bound in a substitution are of the form `Var("v",n)`, for some `n`, we may use an implementation of substitutions which we refer to as an *array-substitution*. In an array substitution we have an array A, whose elements, $\mathcal{T}$, are the representations of Gödel terms such that $A(i) = \mathcal{T}$ represents the binding of `Var("v",i)` to $\mathcal{T}$. We must assume that this array supports some null value N, where $A(i) = N$ if variable `Var("v",i)` is not bound in the substitution.

In common with most logic programming languages, Gödel does not provide any direct support for arrays. Nevertheless we are able to implement an approximation of an array for which the access time for an element of an array is logarithmic upon the size of that array. This compares very favourably with the access time for an element of a list. With a suitable support for arrays we could potentially improve this to a constant access time.

Gödel Substitutions

In the representation of a substitution the majority of variable bindings will be for variables of the form `Var("v",n)`, with generally only a few bindings for variables of the form `Var(s,n)`, where `s` is some string other than `"v"`. These latter variables will in general be the variables that appeared in the query being interpreted.

In order to support the bindings of both kinds of variable Gödel represents a substitution as the binary term `TermSubst(array,list)`, where `array` is an array-substitution used to record bindings of variables whose root is `"v"` and `list` is a list-substitution used to record bindings of variables whose root is some string other than `"v"`.

To facilitate the process of determining whether the root of a variable is the string `"v"` or not, Gödel provides a second representation of variables as `Var(n)`, where `n` is an integer. A variable with root `"v"` and index `n` is represented by the term `Var(n)` and a variable with root `s` (where `s` $\neq$ `"v"`) and index `n` is represented by the term `Var(s,n)`. Thus in the representation of a substitution, `TermSubst(array,list)`, `array` is an array-substitution used to record bindings of variables represented as `Var(n)` and `list` is a list-substitution used to record bindings of variables represented as `Var(s,n)`, where `s` $\neq$ `"v"`.

A.2.2 Adding Variable Bindings

In the definition of most correct unification algorithms, such as that in [46], when a variable binding v/t is to be added to a substitution θ, then the substitution θ must be *composed* with the substitution $\{v/t\}$. We take the following definition for composition of substitutions from [46].

Definition Let $\theta = \{u_1/s_1, \ldots, u_m/s_m\}$ and $\sigma = \{v_1/t_1, \ldots, v_n/t_n\}$ be substitutions. Then the *composition* $\theta\sigma$ of θ and σ is the substitution obtained from the set

$$\{u_1/s_1\sigma, \ldots, u_m/s_m\sigma, v_1/t_1, \ldots, v_n/t_n\}$$

by deleting any binding $u_i/s_i\sigma$ for which $u_i = s_i\sigma$ and deleting any binding v_j/t_j for which $v_j \in \{u_1, \ldots, u_m\}$.

From this definition we see that to add a binding v/t to a substitution $\theta = \{u_1/s_1, \ldots, u_m/s_m\}$ we must apply the binding $\{v/t\}$ to the terms $\{s_1, \ldots, s_m\}$. When θ is a large substitution (of the order of thousands of bindings, for example) then this operation will be extremely expensive. Our solution to this expense is to reduce it to nothing by simply not performing this operation.

The final operation in adding the binding v/t to θ would be to add v/t only if v is not already bound in θ. We avoid both the need to test for this case and the need to perform the above application of the binding v/t to θ, by implementing a substitution as a set of variable bindings for which composition has not been performed. We refer to such a substitution as an *uncomposed substitution*.

In an uncomposed substitution θ we may have a chain of references for some variable, v say. That is to say, v may be bound to a variable w, which is in turn bound to a variable x, which is in turn bound to a variable... and so on. To handle uncomposed substitutions we must introduce the concept of *dereferencing* a variable in an uncomposed substitution.

Definition Let v be a variable, θ an uncomposed substitution. Then t, the *dereferenced value* of v in θ, is the unbound variable or non-variable term t such that $v/v_1, v_1/v_2, \ldots, v_n/t$ are all variable bindings in θ.

When attempting to add a binding for a variable v to a term t in a substitution θ, we first dereference v in θ. If the dereferenced value, v', of v is a variable then, by the above definition, v' is not bound in θ. Therefore we may add the binding v'/t to θ without needing to test whether v' is already bound in θ.

By implementing the representation of substitutions as uncomposed substitutions we have avoided entirely the expense of performing composition. Unfortunately we have introduced an extra expense into the application of variable bindings as a variable, and all subterms of the term to which this variable is bound, must now be dereferenced when we apply the relevant binding to this variable. This is an acceptable trade-off however, as the extra expense introduced by the need

to dereference variables is minuscule compared to the expense of the composition operation. In fact it can be seen that representing substitutions as uncomposed substitutions is analogous to the representation of substitutions implemented by the heap (or global stack) of the WAM [2, 76].

A.2.3 Implementing Substitutions

Before we present the Gödel code that implements the representation of substitutions we give a brief overview of the implementation of array-substitutions.

The two operations of applying and adding a variable binding in an array-substitution are implemented by the predicates `Contents` and `AddBinding` respectively. These predicates essentially emulate the retrieval and insertion of elements in an array. A call to `Contents(i,a,v)` for integer `i` and array `a` returns the value, `v`, of element `a(i)`. A call to `AddBinding(i,v,a,a1)` for integer `i` and array `a` returns the array, `a1`, resulting from adding the value `v` as element `a(i)` to the array in the third argument.

Gödel's array substitutions use the term `N` (the null value), to record an unbound variable. `R(i)`, where `i` is an integer, records a variable that is bound to the variable `Var(i)`. `V(var)` records a variable bound to the variable `var`, where `var` is a variable of the form `Var(s,i)`. `T(t)` records the binding of a variable to a non-variable term `t`. With the above information on the implementation of array-substitutions we may now present the code that implements Gödel's representation of substitutions.

The application of a variable binding may by illustrated by the application of a substitution to a term :

```
ApplySubstToTerm(term, subst, term1) <-
  Dereference(term, subst, term2) &
  FullDereference(term2, subst, term1).

ApplySubstToArgs([], _, []).
ApplySubstToArgs([arg|rest], subst, [arg1|rest1]) <-
  ApplySubstToTerm(arg, subst, arg1) &
  ApplySubstToArgs(rest, subst, rest1).
```

`ApplySubstToTerm` calls `Dereference` to dereference a term in a substitution. When this term is not a variable, `Dereference` does nothing. When the term is a variable it is dereferenced to return the unbound variable or non-variable term which this variable references. The call to `Dereference` is followed by a call to `FullDereference` which recursively dereferences any subterms of the term returned by the call to `Dereference`.

```
Dereference(Var(index), TermSubst(array, list), term) <-
  Contents(index, array, v1) &
  Dereference1(v1, Var(index), array, list, term).
```

```
Dereference(Var(s, i), TermSubst(array, list), term) <-
  IF SOME [term1]  Member(Var(s, i)! term1, list)
    THEN  Dereference(term1, TermSubst(array, list), term)
    ELSE  term = Var(s, i).
Dereference(Term(name, args), _, Term(name, args)).
Dereference(CTerm(name), _, CTerm(name)).

Dereference1(N, var, _, _, var).
Dereference1(R(index), _, array, list, value) <-
  Contents(index, array, v1) &
  Dereference1(v1, Var(index), array, list, value).
Dereference1(V(var), _, array, list, term) <-
  Dereference(var, TermSubst(array, list), term).
Dereference1(T(term), _, _, _, term).

FullDereference(Var(v, n), _, Var(v, n)).
FullDereference(Var(n), _, Var(n)).
FullDereference(Term(name, args), subst, Term(name, args1)) <-
  ApplySubstToArgs(args, subst, args1).
FullDereference(CTerm(term), _, CTerm(term)).
```

Before a binding is performed by the WAM-like predicates presented in the following section, the variable to be bound is dereferenced. Consequently the only variables which may be bound in a substitution are unbound variables. This means that the predicate `BindVariable`, which binds an unbound variable in a substitution, needs merely to insert this binding.

```
BindVariable(Var(n, i), term, TermSubst(a, l), TermSubst(a, [Var(n, i)! term|l])).
BindVariable(Var(i), term, TermSubst(a, l), TermSubst(a1, l)) <-
  AddTermBinding(term, i, a, a1).

AddTermBinding(Var(s, i), var, array, new_array) <-
  AddBinding(var, V(Var(s, i)), array, new_array).
AddTermBinding(Var(index), var, array, new_array) <-
  AddBinding(var, R(index), array, new_array).
AddTermBinding(Term(name, args), var, array, new_array) <-
  AddBinding(var, T(Term(name, args)), array, new_array).
AddTermBinding(CTerm(name), var, array, new_array) <-
  AddBinding(var, T(CTerm(name)), array, new_array).
```

A.2.4 Implementing `ComposeTermSubsts`

The only remaining operation on substitutions worth noting is composition. This operation is performed using the predicate `ComposeTermSubsts` provided by the system module `Syntax`. There is a problem with the implementation of this operation in that it is unclear how it may be correctly implemented due to the inherently uncomposed nature of the above implementation of substitutions.

Before we discuss this problem we present the current implementation of `ComposeTermSubsts`. We first present the code for the predicate `ComposeTermSubsts1`.

```
ComposeTermSubsts1(TermSubst(a, l), subst, TermSubst(a1, l1)) <-
        RationaliseTermList(l, l1, subst) &
        RationaliseTermArray(a, subst, a1).

RationaliseTermList([], [], _).
RationaliseTermList([var! term|rest], [var! term1|rest1], subst) <-
        ApplySubstToTerm(term, var, subst, term1) &
        RationaliseTermList(rest, rest1, subst).
```

A call to `ComposeTermSubsts1(s1,s2,s3)` returns the substitution $\{v_1/t_1\theta, \ldots, v_n/t_n\theta\}$ as argument `s3`, where `s1`=$\{v_1/t_1, \ldots, v_n/t_n\}$ and `s2`=θ. A special case of this is the call `ComposeTermSubsts1(s,s,s1)` which has the effect of instantiating `s1` to the *idempotent* substitution derived from `s`. That is, the substitution θ such that $\theta = \theta\theta$ and v/t is a binding in θ iff t is the fully dereferenced binding for v in the substitution `s`.

We refer to the process of deriving an idempotent substitution from an uncomposed substitution in the above manner as the *rationalisation* of a uncomposed substitution. We use this property of the predicate `ComposeTermSubsts1` to define `ComposeTermSubsts`.

```
ComposeTermSubsts(subst1, subst2, subst3) <-
  ComposeTermSubsts1(subst1, subst1, rational1) &
  ComposeTermSubsts1(rational1, subst2, compose1) &
  ComposeTermSubsts1(subst2, subst2, rational2) &
  ComposeTermSubsts2(rational2, compose1, subst3).

ComposeTermSubsts2(TermSubst(a1, s1), TermSubst(a2, s2), TermSubst(a3, s3)) <-
  ComposeArrays(a1, a2, a3) &
  ComposeLists(s2, s1, s3).

ComposeLists([], h, h).
ComposeLists([var! term|rest], h, [var! term|h1]) <-
  DeleteBindings(h, var, h2) &
  ComposeLists(rest, h2, h1).
```

```
DeleteBindings([], _, []).
DeleteBindings([var! term|rest], var1, rest1) <-
  IF  var = var1
    THEN  rest1 = rest
    ELSE  rest1 = [var! term|rest2] &
          DeleteBindings(rest, var1, rest2).
```

Note that in the above code the predicates `RationaliseTermArray` and `ComposeArrays` perform operations on array-substitutions that are analogous to those performed on list-substitutions by the predicates `RationaliseTermList` and `ComposeLists` respectively.

The problem with the above code for `ComposeTermSubsts` (and the equivalent code for `ComposeTypeSubsts`) is due to the fact that substitutions are implemented as uncomposed substitutions.

For example, let $\theta = \{y/z\}$ and $\sigma = \{x/y\}$ be substitutions. By the above definition the composition, $\theta\sigma$, of these two substitutions is $\{y/z, x/y\}$. However, as the current implementation represents substitutions as uncomposed substitutions, this substitution will appear as if it were the substitution $\{y/z, x/z\}$ when any dereferencing is performed.

The simplest way to ensure that the implementation of `ComposeTermSubsts` is correct is to insist that substitutions are implemented as composed substitutions. Unfortunately, as we described above, this is hopelessly inefficient. At the current time no apparent implementation of substitutions has been suggested which is both at least *reasonably* efficient and captures the full correctness of the composition operator. This issue requires further investigation, although the results will have no impact upon *SAGE*. *SAGE* uses `ResolveAll` to perform unfolding and thus does not depend on the predicate `ComposeTermSubsts`.

The use of `ComposeTermSubsts` can also introduce circular bindings into a substitution by, for example, composing the two substitutions $\{x/y, y/z\}$ and $\{z/x\}$. When dereferencing a binding in an uncomposed substitution where a circular binding occurs this may cause the computation to enter an infinite loop. This problem can be avoided by testing the new bindings added during a call to `ComposeTermSubsts` and using some mechanism to highlight circular bindings. We do not present the details here.

A.3 The WAM-like Predicates

In the previous section we presented the current implementation for the representation of substitutions. We now give the current implementation of the WAM-like predicates, which is based upon the above implementation of substitutions.

The two predicates `GetConstant` and `GetFunction` which, together with `UnifyTerms`, unify arguments of the head of a statement with the matching arguments of the atom being resolved are

implemented as follows.

```
GetConstant(term, c, bind, bind1) <-
  Dereference(term, bind, term1) &
  GetConstant1(term1, c, bind, bind1).

GetConstant1(Var(n), c, TermSubst(array, subst), TermSubst(new_array, subst)) <-
  AddBinding(n, T(c), array, new_array).
GetConstant1(Var(s, i), c, TermSubst(a, z), TermSubst(a, [Var(s, i)! c|z])).
GetConstant1(CTerm(name), CTerm(name), bind, bind).

GetFunction(term, f, mode, bind, bind1) <-
  Dereference(term, bind, term1) &
  GetFunction1(term1, f, mode, bind, bind1).

GetFunction1(Var(n), f, Write, TermSubst(a, s), TermSubst(a1, s)) <-
  AddBinding(n, T(f), a, a1).
GetFunction1(Var(s, i), f, Write, TermSubst(a, z), TermSubst(a,[Var(s, i)! f|z])).
GetFunction1(Term(f, a), Term(f, a), Read, bind, bind).
```

The four predicates which perform the unification operations necessary for processing the arguments of function terms in the head of the statement are implemented as follows.

```
UnifyVariable(Write, Var(var), var, var+1).
UnifyVariable(Read, _, var, var).

UnifyValue(Write, term, term, bind, bind).
UnifyValue(Read, term, term1, bind, bind1) <-
  UnifyTerms(term, term1, bind, bind1).

UnifyConstant(Write, term, term, bind, bind).
UnifyConstant(Read, term, c, bind, bind1) <-
  Dereference(term, bind, term1) &
  GetConstant1(term1, c, bind, bind1).

UnifyFunction(Write, term, term, Write, bind, bind).
UnifyFunction(Read, term, f, mode, bind, bind1) <-
  Dereference(term, bind, term1) &
  GetFunction1(term1, f, mode, bind, bind1).
```

In order to support the implementation of `UnifyTerms` and `UnifyAtoms` two further WAM-like predicates are needed. These are called `GetVariable` and `UnifyKnownVariable` and have the following implementation.

```
GetVariable(Var(var), term, TermSubst(h, l), TermSubst(h1, l)) <-
  IF  term = Var(var)
    THEN  subst1 = subst
    ELSE  TermNotOccur1(term, Var(var)) &
          AddTermBinding(term, var, h,  h1).
GetVariable(Var(s, v), term, TermSubst(h, l), TermSubst(h,[Var(s, v)! term|l])) <-
  IF  term = Var(s, v)
    THEN  subst1 = subst
    ELSE  TermNotOccur1(term, Var(s, v)).
GetVariable(CTerm(name),  var,  subst,  subst1) <-
  BindVariable(var,  CTerm(name),  subst,  subst1)
GetVariable(Term(name, args),  var,  subst,  subst1) <-
  TermNotOccur(args, var) &
  BindVariable(var,  Term(name, args),  subst,  subst1).

UnifyKnownVariable(Write, var, var, subst, subst).
UnifyKnownVariable(Read, term, var, subst, subst1) <-
  GetVariable(term, var, subst, subst1).
```

The implementation of `UnifyTypes` given in section A.1 is similar to that for `UnifyTerms`. It relies upon six further WAM-like predicates for handling the unification operations for types. These predicates are named `GetParameter`, `GetBase`, `GetType`, `UnifyParameter`, `UnifyBase` and `UnifyType`. They are directly analogous to the WAM-like predicates for terms `GetVariable`, `GetConstant`, `GetFunction`, `UnifyKnownVariable`, `UnifyConstant` and `UnifyFunction` respectively. The implementations of these WAM-like predicates for types are similarly analogous to those for the corresponding WAM-like predicates for terms.

Bibliography

[1] S Abramsky and C Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[2] H Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

[3] K Benkerimi and J W Lloyd. A procedure for the partial evaluation of logic programs. In S Debray and M Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, Austin, Texas, November 1990. MIT Press.

[4] D Bjorner, A P Ershov, and N D Jones, editors. *Workshop on Partial Evaluation and Mixed Computation*, Denmark, October 1987. Gl. Avernaes.

[5] R Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, May 1993.

[6] K A Bowen and R A Kowalski. Amalgamating language and metalanguage in logic programming. In K L Clark and S-A Tarnlund, editors, *Logic Programming*, pages 153–172, 1982.

[7] A F Bowers. Representing Gödel object programs in Gödel. Technical Report CSTR-92-31, Department of Computer Science, University of Bristol, November 1992.

[8] M Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, 1991.

[9] M Bruynooghe, D de Schreye, and B Krekels. Compiling control. *Journal of Logic Programming*, 6:135–162, 1989.

[10] M Bruynooghe, D de Schreye, and B Martens. A general criterion for avoiding infinite unfolding during partial deduction of logic programs. *New Generation Computing*, 11:47–79, 1992.

[11] M A Bulyonkov. Extracting polyvariant binding time analysis from a polyvariant specializer. In *Workshop on Partial Evaluation and Program Manipulation*, Copenhagen, 1993. ACM Press.

[12] M Carlsson and J Widén. Sicstus prolog user's manual. Technical report, SICS, August 1992.

[13] 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.

[14] P Cousot and R Cousot. Static analysis of programs by construction on approximation of fixpoints. In *Principles of Programming Languages*, 1977.

[15] D.A. de Waal. The Power of Partial Evaluation. In *Proceedings of the Third International Workshop on Logic Program Synthesis and Transformation*. Springer-Verlag, July 1993.

[16] D.A. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation,* Manchester 1991, pages 205–221. Workshops in Computing, Springer-Verlag, 1992.

[17] S K Debray and D S Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5:207–229, 1988.

[18] H Fujita and K Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6:91–118, 1988.

[19] D A Fuller and S Abramsky. Mixed computation of prolog programs. *New Generation Computing*, 6:119–142, 1988.

[20] Y Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[21] J Gallagher. Transforming logic programs by specialising interpreters. In *ECAI-86*, pages 109–122, Brighton, 1986.

[22] J Gallagher and M Bruynooghe. Some low-level source transformations for logic programs. In *Meta90*, pages 229–244, March 1990.

[23] J Gallagher and M Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9:305–333, 1991.

[24] J Gallagher, M Codish, and E Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6:159–186, 1988.

[25] J Gallagher, M Codish, and E Shapiro. Using safe approximation of fixed points for analysis of logic programs. In H D Abramson and M H Rogers, editors, *Meta-Programming in Logic*

Programming, Proceedings of the Meta88 Workshop, June 1988, pages 233–262. MIT Press, 1989.

[26] P M Hill and J W Lloyd. Meta-programming for dynamic knowledge bases. Technical Report CS-88-18, Department of Computer Science, University of Bristol, 1988.

[27] P M Hill and J W Lloyd. Analysis of meta-programs. In H D Abramson and M H Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*. MIT Press, 1989.

[28] 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.

[29] 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.

[30] P M Hill and R W Topor. A semantics for typed logic programs. In F Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.

[31] T Horváth. Experiments in partial deduction. Msc thesis, K.U.Leuven, 1993.

[32] D Jacobs and A Langen. Accurate and efficient approximation of variable aliasing in logic programs. In *Proceedings of the North American Conference on Logic Programming*, pages 154–165. MIT Press, 1989.

[33] N D Jones. Challenging problems in partial evaluation and mixed computation. *New Generation Computing*, 6:291–302, 1988.

[34] N D Jones, C K Gomard, and P Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[35] N D Jones, P Sestoft, and H Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. *Rewriting Techniques and Applications, Lecture Notes in Computer Science 202*, pages 124–140, 1985.

[36] N D Jones and H Søndergaard. A semantics based framework for the abstract interpretation of Prolog. In S Abramsky and C Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[37] T Kanamori and K Horiuchi. Construction of logic programs based on generalised unfold/fold rules. In Jean-Louis Lassez, editor, *4th International Conference on Logic Programming*, pages 744–768. MIT Press, 1987.

[38] T Kanamori and T Kawamura. Abstract interpretation based on OLDT resolution. *Journal of Logic Programming*, 15(1&2):1–30, January 1993.

[39] H J Komorowski. A specification of an abstract prolog machine and its application to partial evaluation. Technical Report LSST 69, Linkoping University, 1981.

[40] H J Komorowski. Towards a programming methedology founded on partial deduction. In *European Conference on Artificial Intelligence*, Stockholm, Sweden, August 1990.

[41] P Kursawe. How to invent a prolog machine. *New Generation Computing*, 5:97–114, 1987.

[42] A Lakhotia. To PE or not to PE. In *Meta90*, pages 218–228, March 1990.

[43] A Lakhotia and L Sterling. How to control unfolding when specialising interpreters. *New Generation Computing*, 8:61–70, 1990.

[44] C Lecoutre, P Devienne, and P Lebegue. Abstract interpretation and recursive behaviour of logic programs. In K-K Lau and T Clement, editors, *LOPSTR 91. Workshops in Computing*, pages 147–166. Springer-Verlag, 1992.

[45] G Levi and G Sardu. Partial evaluation of metaprograms in a “multiple worlds” logic. *New Generation Computing*, 6:227–248, 1988.

[46] J W Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2 edition, 1987.

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

[48] H Mannila and E Ukkonen. Flow analysis of prolog programs. In *Proceedings 1987 Symposium on Logic Programming*, pages 205–214, San Francisco, August 1987. Computer Society Press of the IEEE.

[49] B Martens, D de Schreye, and T Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122, 1994. To appear.

[50] C S Mellish. Abstract interpretation of Prolog programs. In S Abramsky and C Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[51] T Mogenson and A Bondorf. Logimix: A Self-Applicable Partial Evaluator for Prolog. In K-K Lau and T Clement, editors, *LOPSTR 92. Workshops in Computing*, pages 214–227. Springer-Verlag, January 1993.

[52] K Muthukumar and M Hermenegildo. Determination of variable dependance information through abstract interpretation. In *Proceedings of the North American Conference on Logic Programming*, pages 166–185, 1989.

[53] K Muthukumar and M Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In K Furukawa, editor, *8th International Conference on Logic Programming*, pages 49–63, Paris, June 1991. MIT Press.

[54] U Nilsson. Towards a methodology for the design of abstract machines for logic programming languages. *Journal of Logic Programming*, 16(1&2):163–189, May 1993.

[55] G Nuemann. A simple transformation from prolog-written metalevel interpreters into compilers and its implementation. In *Proceedings of the Second Russian Conference on Logic Programming*, number 592 in LNAI. Springer-Verlag, September 1991.

[56] S Owen. Issues in the partial evaluation of meta-interpreters. In H D Abramson and M H Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 319–340. MIT Press, 1989.

[57] L Plümer. Termination proofs of logic programs. *Lecture Notes in Computer Science 446*, 1990.

[58] S Safra and E Shapiro. Meta interpreters for real. In H J Kugler, editor, *Information Processing 86*, pages 271–278. North-Holland, 1986.

[59] D Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, SICS, March 1991.

[60] H Seki. Unfold/fold transformations of stratified programs. In Giorgio Levi and Maurizio Martelli, editors, *6th International Conference on Logic Programming*, pages 554–568. MIT Press, 1989.

[61] H Seki. Unfold/fold transformations for general logic programs based on the well-founded semantics. *Journal of Logic Programming*, 16(1&2):5–24, May 1993.

[62] P Sestoft. The structure of a self-applicable partial evaluator. *Programs As Data Objects, Lecture Notes in Computer Science 217*, pages 236–256, 1986.

[63] P Sestoft and A V Zamulin. Annotated bibliography on partial evaluation and mixed computation. *New Generation Computing*, 6:309–354, 1988.

[64] D A Smith and T J Hickey. Partial evaluation of a CLP language. In S Debray and M Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, Austin, Texas, November 1990. MIT Press.

[65] H Søndergaard. Abstract interpretation of logic programs. Technical Report 86-7-10, DIKU, Copenhagen, 1987.

[66] L S Sterling and R D Beer. Meta-interpreters for expert system construction. *Journal of Logic Programming*, 6:163–178, 1989.

[67] A Takeuchi and H Fujita. Competitive partial evaluation - some remaining problems of partial evaluation. *New Generation Computing*, 6:259–278, 1988.

[68] A Takeuchi and K Furukawa. Partial evaluation of Prolog programs and its application to meta-programming. In H J Kugler, editor, *Information Processing 86*, pages 415–420, Dublin, 1986. North Holland.

[69] H Tamaki and T Sato. Unfold/fold transformations of logic programs. In S-A Tarnlund, editor, *2nd International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, June 1984.

[70] J D Ullman and A Van Gelder. Efficient tests for top-down termination of logical rules. *Journal of the ACM*, 35(2):345–373, April 1988.

[71] R Venken. A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query optimisation. In *ECAI-84: Advances in Artificial Intelligence*, pages 91–100, Pisa, 1984. North-Holland.

[72] R Venken and B Demoen. A partial evaluation system for Prolog: Some practical considerations. *New Generation Computing*, 6:279–290, 1988.

[73] K Verschaetse and D de Schreye. Deriving termination proofs for logic programs, using abstract procedures. In K Furukawa, editor, *8th International Conference on Logic Programming*, pages 301–315, Paris, June 1991. MIT Press.

[74] K Verschaetse, D de Schreye, and M Bruynooghe. Generation and compilation of efficient computation rules. In D H D Warren and P Szeridi, editors, *7th International Conference on Logic Programming*, pages 700–714, Jerusalem, June 1990. MIT Press.

[75] K Verschaetse, S Decorte, and D de Schreye. Automatic termination analysis. In K-K Lau and T Clement, editors, *LOPSTR 92. Workshops in Computing*, pages 168–183. Springer-Verlag, January 1993.

[76] D H D Warren. An abstract prolog instruction set. Technical Note 309, SRI International, 1983.