Effective Meta-programming in Declarative Languages

Antony Francis Bowers

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 1998

Abstract

Declarative meta-programming is vital, since it is the most promising means by which programs can be made to reason about other programs. A meta-program is a program that takes another program, called the object program, as data. A declarative programming language is a programming language based on a logic that has a model theory.

A meta-program operates on a representation of an object program. In a non-ground representation, object-level variables are represented by meta-variables; in a ground representation, object-level variables are represented by ground terms. The non-ground representation is insufficiently expressive for most meta-programming tasks. The ground representation is adequately expressive, but meta-programs are complex and laborious to write, and the overhead of interpretation is excessive.

Gödel is a declarative programming language based on first-order logic. It has types and modules, improves on the expressiveness and declarative semantics of Prolog, and provides extensive support for meta-programming in the ground representation.

This thesis contributes to the design of Gödel's meta-programming facilities, and investigates the techniques required to make declarative meta-programming practical. The representation of Gödel object programs as ground terms is described. Library modules and abstract types eliminate much of the labour and complexity of meta-programming. An analysis of the sources of interpretation overhead is conducted, and an idiom for the construction of potentially efficient interpreters is developed. Carefully designed interpreters are improved significantly by a basic partial evaluator, but not all interpreter designs are susceptible. Experiments suggest that specialised interpreters can be improved still further by specific low-level mechanisms. SLDQE-resolution, a novel computational model for the compilation of programs containing universally quantified implication formulas, is also presented.

The problem of providing a language for declarative meta-programming that is both adequately expressive and efficiently executable is not completely solved, but this work demonstrates that the possibilities of a simple ground representation deserve to be taken seriously.

Acknowledgements

First, I wish to thank my supervisor, John Lloyd, for his guidance in the production of this work. His artistry and boundless reserves of patience and optimism are inspiring, and the chance to work with him has been a privilege greater than any I could have wished for. I also owe a debt of gratitude to Pat Hill; her friendship, encouragement and confidence in me shine in my memory. Pat and John are the designers of the Gödel language, the elegance of which is a testament to their fine sense of style. This work would not have been possible without them.

The realisation of a programming system as complex as Gödel is too much for one person to accomplish alone. Jiwei Wang shared the task with me over several years, and I thank him for his splendid companionship and hard work. Andrea Domenici also contributed to the Gödel implementation and was a good friend.

I would like to thank Corin Gurr for his work on the *SAGE* partial evaluator, and for the ideas he contributed to the efficiency of the ground representation. His input was invaluable.

The members of the Computer Science department at the University of Bristol have taken care to make my work here a pleasure; I am grateful to you all. Stimulating company was also provided by the members of the Compulog and Compulog 2 projects.

Finally, and most of all, I thank my long-suffering family. While I was writing this thesis, my wife, Rachel, and daughter, Leonora, cheerfully made sacrifices so that I could work undisturbed, and gave me more support than anyone could expect. I shall never be able to repay them, but perhaps now I shall be able to take Leonora's advice: "Daddy, you work too much. You should play more."

This work was partly supported by ESPRIT Basic Research Action 3012 (Compulog) and Project 6810 (Compulog 2).

To Terry, Marie, Rachel and Leonora 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 this work has previously been submitted in support of an application for a degree of this or any other university.

A. F. Bowers

Contents

1	Intr	Introduction				
	1.1	Aims		3		
	1.2	Langu	ages and Representations	5		
		1.2.1	Languages	5		
		1.2.2	Representations	7		
	1.3	Appro	paches to meta-programming	11		
		1.3.1	Non-ground Representations	12		
		1.3.2	Meta-programming in Prolog	14		
		1.3.3	Reflection and Amalgamation	16		
		1.3.4	Ground Representations	17		
	1.4	Relate	ed work	19		
	1.5	Overv	iew	22		
2	The	Göde	el Language	23		
	2.1	Featur	res of Gödel	24		
		2.1.1	Types	24		
		2.1.2	Formulas	31		
		2.1.3	Modules	34		
		2.1.4	Control	38		
		2.1.5	System Modules	43		
		2.1.6	Semantics	43		
	2.2	Meta-	programming in Gödel	44		
		2.2.1	The Syntax Module	45		
		2.2.2	The Programs Module	53		
		2.2.3	Other Meta-modules	59		

3	$\mathbf{A}\mathbf{n}$	Imple	mentation of Gödel	7 6
	3.1	Repre	senting Gödel Syntax	76
		3.1.1	Representing Symbol Names	77
		3.1.2	Representing Types	79
		3.1.3	Representing Terms, Variables and Atoms	80
		3.1.4	Representing Formulas	82
	3.2	Repre	senting Gödel Programs	86
		3.2.1	Symbol Tables	86
		3.2.2	Representing the Module Structure	88
		3.2.3	Representing the Program Language	89
		3.2.4	Representing the Program Code	95
	3.3	Comp	iling Gödel Programs	97
		3.3.1	Theoretical Aspects of Compilation	98
		3.3.2	Translation to Prolog	106
	3.4	Reflec	tive Interpreters	111
4	An	Analy	sis of Gödel Meta-programming 1	13
	4.1	Subst	itutions and Unifications	114
	4.2	Simpl	e Interpreters	119
		4.2.1	Instance Demo	120
		4.2.2	An SLD Interpreter using Composition	122
		4.2.3	Unify Demo	125
		4.2.4	The Resolve Predicate and SLD Demo	128
	4.3 Optimising SLD Demo			
		4.3.1	Partial Evaluation	136
		4.3.2	Specialising SLD Demo	139
		4.3.3	An Additional Optimisation	141
		4.3.4	Results	143
	4.4	Matte	ers Arising	
		4.4.1	Mutable Data Structures in Gödel	145
		4.4.2	Garbage Collection	148
	4.5	The B	Breadth-first Interpreter	150
	4.6	Summ	nary	154
5	Cor	ıclusio	ns 1	. 56
	5.1	An Ev	valuation of Gödel Meta-programming	156
		5.1.1	Expressiveness	156

		5.1.2	Efficiency	157	
		5.1.3	Reflective Interpreters	159	
		5.1.4	Representation	160	
		5.1.5	Type Checking	161	
		5.1.6	Modules	162	
		5.1.7	Partial Evaluation	164	
	5.2	Future	e Work	165	
		5.2.1	Improving Gödel	165	
		5.2.2	Meta-programming in Escher	167	
	5.3	Contri	butions	175	
	5.4	Conclu	uding Remarks	176	
A	Definitions and Theoretical Background 178				
	A.1	Polym	orphic Many-sorted Logic	178	
	A.2	A Pro	gram and its Completion	182	
	A.3	SLDQ	E-resolution	184	
B Gödel Meta-programming Facilities				187	
	B.1	The S	yntax Module	187	
	B.2	The P	rograms Module	206	
Bi	bliog	graphy		228	

List of Figures

2.1	A simple Gödel program	26
2.2	A Gödel program for flattening a nest	29
2.3	A Prolog program for flattening a nested list	30
2.4	A Gödel program for flattening a simplified nest	32
2.5	The definition of Sorted	42
2.6	An assimilation algorithm for a Knowledge Based System $$ $$ 6	32
2.7	Generating and filtering transactions	33
2.8	Finding a goal in an SLD-tree	35
2.9	Selecting an atom from the body of a goal	66
2.10	Generating transactions from goals	37
2.11	Filtering transactions	39
2.12	Testing the assimilator	70
2.13	Transforming transactions to visible syntax	71
2.14	A family tree knowledge base	74
2.15	Some integrity constraints for the family tree knowledge base	75
3.1	A program for reversing a list)()
3.2	The normal form of ReverseList)1
3.3	The definition of try)9
4.1	A non-declarative simulation of unification in Prolog 11	l5
4.2	The Instance Demo interpreter	21
4.3	An interpreter based on composition of substitutions 12	23
4.4	The predicates Select and Mgu	24
4.5	A predicate for testing interpreters	26
4.6	Unify Demo	29
4.7	The Resolve predicate	30
4.8	SLD Demo	₹9

4.9	Handling system predicates with Compute
4.10	Performance of interpreters
4.11	A vanilla interpreter in Gödel
4.12	The export part of module <code>ONaive</code>
4.13	The local part of module ONaive $\dots \dots \dots$
4.14	Specialisation of Demo wrt Append
4.15	Single-threading of substitutions
4.16	Effect of optimising interpreters
4.17	Part of Unify Demo specialised to Append 146
4.18	The Breadth First interpreter
4.19	Generating child nodes
5.1	An Escher program
5.2	An Escher rewrite rule
5.3	A trivial monadic interpreter
5.4	A monadic depth-bounded interpreter

Chapter 1

Introduction

Declarative meta-programming is vital. It is the most likely means by which computer systems of the future will be able to reason about programs, both their own and those of the other systems with which they interact. If we can teach them to do that effectively, the expansion in their potential will be enormous. If we cannot, the prospect that computer systems will one day be capable of learning, exchanging knowledge and interacting autonomously, dims and recedes.

Hill and Lloyd [39] summarise declarative programming as being "much more concerned with writing down what should be computed and much less concerned with how it should be computed". There is a clear separation between the control component of an algorithm (the "how") and the logical component (the "what"). This separation makes declarative programs easier to understand and easier to reason about both formally and intuitively than their counterparts in imperative languages.

The key idea of declarative programming [51] is that a program is a theory in some suitable logic, and computation is deduction from the theory. A suitable logic should at least have a model theory, a proof theory, and a soundness theorem. The model theory provides the declarative semantics of the program. It is a starting point for the construction of programs: the programmer begins with an intended interpretation, which is an understanding of the domain in which the program is to operate, and writes statements that are true in this interpretation. I therefore regard it as essential to the success of a declarative language that it has a model theory simple enough to be grasped intuitively and effortlessly by programmers. The proof theory

provides the mechanism by which a computer system executes the program, and the soundness theorem guarantees that every answer computed is true in the intended interpretation. Suitable logics include first-order logic and its sorted variants, which is the basis for logic programming (perhaps more properly called *relational programming*), and λ -calculus, which is the basis for functional programming languages.

Declarative languages are a powerful tool for the construction of complex programs, not only because they have declarative semantics, but also because they are compact, high-level and expressive. Beginning with Aristotle (384-322 B.C.), mathematical logic has developed from attempts to formalise and automate aspects of human reasoning, and is therefore close to the way we naturally express ideas about the world.

A program, in any programming language, that takes one or more programs as data is a *meta-program*. The programs that provide the data for a meta program are called *object programs*. When the object program is declarative, the meta-program can make use of the declarative semantics and related theoretical results to manipulate it and reason about it. When the meta-program is declarative, it not only possesses the attributes and advantages of a declarative program, but can also bring these advantages to another meta-program when it itself takes its turn as an object program.

The subject of this thesis is *effective* declarative meta-programming, in which both the meta-programs and object programs are declarative. An effective programming system is one that is both expressive and efficient: expressive enough to allow most common algorithms to be expressed succinctly, and efficient enough for the resulting programs to execute in a reasonable time on readily available hardware.

The applications of meta-programming are numerous. They include compilers, program transformers, program synthesisers, program analysers, and debuggers. All these applications benefit from the power of declarative programming. Since it is an ideal formalism for the representation of knowledge, declarative programming also plays a crucial role in applications related to artificial intelligence, such as expert systems, knowledge based systems, machine learning, intelligent agents and the simulation of modal logics of knowledge and belief.

Logic programming languages are based on first-order logic, with restrictions that make efficient proof procedures possible. In its most basic form,

a logic program is a finite set of Horn clauses, which are formulas of the form $A \leftarrow W$, where A is an atom and W is a conjunction of atoms. A goal is a formula of the form $\leftarrow W$, where W is a conjunction of atoms; goals are evaluated using a proof procedure called SLD-resolution. Logic programming developed out of work on automated theorem proving in the 1960s, particularly the work of Robinson [70] who introduced the resolution principle. Kowalski [45] made the fundamental observation that predicate logic could be used as a programming language. The first logic programming system was Marseille Prolog [22], and the most influential early Prolog implementation was DEC-10 Prolog, created by Warren [65].

This thesis assumes the basic concepts of first-order logic and logic programming, which are well known. Good introductions to first-order logic are [58, 26]. For logic programming, I follow the definitions and terminology of [50]. An alternative formulation of SLDNF-resolution can be found in [2].

1.1 Aims

The main topic of this thesis is the design of an architecture for meta-programming in the logic programming language Gödel. The Gödel programming language was designed by Hill and Lloyd [39]; it is a declarative language based on first-order logic, intended to be more expressive than Prolog, and to have greatly improved declarative semantics. Gödel has a type system based on polymorphic many-sorted logic, and a module system. Statements in Gödel are allowed arbitrary first-order formulas in the body. Programmers can provide control information, in the form of commits that prune the search tree, and control declarations called DELAY declarations that affect the selection rule. Through its module system, Gödel provides a collection of system modules that enrich the language with data types such as integers, lists, strings, and sets.

Gödel places particular emphasis on declarative meta-programming. Although a meta-program and its object program do not necessarily have to be written in the same language, when they are, meta-programs can be self-applicable. It is difficult to provide comprehensive support for meta-programming with object programs in a variety of different languages, and self-application is the most useful case. Gödel meta-programming therefore concentrates on the manipulation of Gödel object programs. As in Prolog,

support for meta-programming in Gödel is most easily provided by a set of system predicates designed for the purpose, and these can be made available from a special group of system modules.

The main issues in the design of a meta-programming architecture are those of expressiveness and computational efficiency. If the language is expressive, the majority of meta-programs will be clear and concise. Computational efficiency is best understood by comparing the direct execution of an object program with its execution under the control of a meta-program; the overhead of interpreting an object program in this way can be very great. The aims of this research can be summarised as follows.

- 1. To develop a comprehensive library of predicates for the manipulation of Gödel object programs, sufficiently expressive for a wide variety of meta-programming tasks.
- 2. To investigate the efficiency of declarative meta-programs, and discover techniques that reduce the computational overhead sufficiently to make declarative meta-programming practical.

The development of a complete, truly declarative meta-programming system for a declarative language as full-featured as Gödel has not previously been undertaken.

It is worth emphasising that in the context of declarative programming, meta-programming is more than just another application: it is distinguished by being such an important strength of declarative languages that it is required as a basic feature. A declarative language lacking meta-programming facilities is severely restricted in applicability. Library design is also properly considered an aspect of language design: the features of the standard system libraries provided by a language are eventually seen as an integral component of the language itself.

In the remainder of this chapter I discuss general issues related to meta-programming: how object programs are represented in meta-programs; the different approaches to meta-programming in logic programming to be found in the literature; related work. Finally I give an overview of the succeeding chapters.

1.2 Languages and Representations

A meta-program must have access to a representation of its object program or programs. More precisely, the language of the object program must be represented in the language of the meta-program. The representation is a mapping from the symbols of the object language to the symbols of the meta-language, but this mapping is not simply a translation from one language to another: the mapping must also enable the meta-language to make statements about structures of the object language, and must therefore also be a quotation mechanism. In the simplest case, both meta-program and object-program are theories in first-order logic [58, 26]. This section explains the terminology of meta-programming.

1.2.1 Languages

To make the following discussion concrete, I give as an example the definition of a many-sorted language, since it is convenient to begin the discussion of representation with a logic that has simple types¹. Many-sorted logic extends first-order logic by the addition of sort declarations for the variables, constants, functions and predicates. The logic of Gödel, polymorphic many-sorted logic, has a more structured notion of type; the definition of a polymorphic many-sorted language can be found in Appendix A. The concept of representation generalises naturally to both (unsorted) first-order languages, which can be regarded as having only one sort, and to polymorphic many-sorted languages.

The alphabet of a many-sorted theory contains types, variables, constants, functions, propositions, predicates, connectives, and quantifiers. At least one type must be present, and also at least one proposition or predicate. Types are denoted by lowercase Greek letters such as τ and σ . For each type τ , there are denumerably many variables $v_{\tau}^1, v_{\tau}^2, \ldots$ Each constant has a type. Functions of arity n have types of the form $\tau_1 \times \ldots \tau_n \to \tau$, and predicates of arity n have types of the form $\tau_1 \times \ldots \tau_n$. For each type τ , there is a universal quantifier \forall_{τ} and an existential quantifier \exists_{τ} .

A term is defined as follows.

1. A variable of type τ is a term of type τ .

¹In the terminology of programming languages, a sort is called a type.

- 2. A constant of type τ is a term of type τ .
- 3. If f is a function of type $\tau_1 \times \ldots \tau_n \to \tau$, and t_i is a term of type τ_i , $i \in \{1, \ldots, n\}$, then $f(t_1, \ldots, t_n)$ is a term of type τ .

A many-sorted formula is defined as follows.

- 1. A proposition p is an atomic many-sorted formula.
- 2. If p is a predicate of type $\tau_1 \times \ldots \tau_n$, and t_i is a term of type τ_i , $i \in \{1, \ldots, n\}$, then $p(t_1, \ldots, t_n)$ is an *atomic* many-sorted formula.
- 3. If F and G are formulas, then so are $\neg F$, $F \land G$, $F \lor G$, $F \leftarrow G$, $F \rightarrow G$, and $F \leftrightarrow G$.
- 4. If F is a many-sorted formula and v_{τ}^{i} is a variable of type τ , then $\exists_{\tau}v_{\tau}^{i}F$ and $\forall_{\tau}v_{\tau}^{i}F$ are many-sorted formulas.

The many-sorted language given by an alphabet consists of the set of all formulas constructed from the symbols of the alphabet.

A formula is said to be *closed* if it contains no free variables; that is, every variable occurring in the formula appears within the scope of a quantifier for that variable.

A many-sorted theory consists of a many-sorted language and a set of closed formulas in this language, called the axioms of the theory.

The set of type declarations for the constants, functions and predicates of a many-sorted theory T is called the signature of T.

It is usual to speak of the language \mathcal{L}_T of a theory (or program) T, where the underlying alphabet is defined by the declarations in the signature of T, or, if T is a theory in unsorted first-order logic, by the constants, functions, propositions and predicates that appear in T. The notation \mathcal{L}_O denotes the language of an object program (object theory) O and \mathcal{L}_M denotes the language of a meta-program (meta-theory) M. \mathcal{L}_M is referred to as a meta-language and \mathcal{L}_O as an object language.

By an abuse of terminology, many authors treat the language of a theory as though it also contains types and terms, and this useful convention is also adopted here. A type (resp., term) of a language \mathcal{L}_T is a type (resp., term) constructed from the symbols of the underlying alphabet of \mathcal{L}_T .

Note that the concept of a *programming language* denotes a class of languages that conform to a specific syntax. The language of a program

written in a programming language is one member of this class, given by the alphabet of symbols declared or used in the program.

A syntactic expression of a language \mathcal{L}_T is any component of the language or its underlying alphabet. For example, the syntactic expressions of a many-sorted language are types, functions, predicates, terms, and formulas.

1.2.2 Representations

A representation of a language \mathcal{L}_O in a language \mathcal{L}_M is a mapping from syntactic expressions of \mathcal{L}_O to terms of \mathcal{L}_M . This mapping is sometimes called a naming relation.

A representation maps a syntactic expression $E \in \mathcal{L}_O$ to a term E' in \mathcal{L}_M . The term E' is referred to as the representation (or name) of E in \mathcal{L}_M . The notation $\lceil E \rceil_M$ denotes the representation of E in the meta-language \mathcal{L}_M . The suffix is omitted where the intended meta-language is clear from the context.

The expressions of \mathcal{L}_O are in general represented by a subset of the terms of \mathcal{L}_M . There is a discussion of the desirable properties of naming relations in [82]; the only required property is that the naming relation should be injective: no term of \mathcal{L}_M should represent more than one expression of \mathcal{L}_O . Naming relations are also usually total, so that every expression in \mathcal{L}_O has a name in \mathcal{L}_M , although this is not essential.

A term that contains no variables is said to be ground. In mathematical treatments of representations, it is usual to stipulate that the expressions of \mathcal{L}_O are represented by ground terms in \mathcal{L}_M . If this is so, the representation is called a ground representation. An alternative form of representation is commonly used in logic programming languages, in which expressions of \mathcal{L}_O are represented by non-ground terms of \mathcal{L}_M ; in particular, variables of \mathcal{L}_O are mapped to variables of \mathcal{L}_M . This form of representation is referred to as a non-ground representation. Non-ground representations have semantic problems, as discussed further in Section 1.3.

As a simple example of a representation, I give one way to represent an object language \mathcal{L}_O in a meta-language \mathcal{L}_M , where \mathcal{L}_O is an unsorted first-order language and \mathcal{L}_M is a many-sorted language. This ground representation is a variant of one from [38], and is adequate for the representation of theories of \mathcal{L}_O that are definite programs². The universal quantifiers that implicitly appear in front of a definite statement are not explicitly represented.

Language \mathcal{L}_M has at least two types, o and μ , where o is the type of terms of the meta-language that represent terms of the object language, and μ is the type of terms that represent formulas of the object language.

For each constant $a \in \mathcal{L}_O$, there is a constant $a' \in \mathcal{L}_M$ of type o. For each variable $x \in \mathcal{L}_O$, there is a constant $x' \in \mathcal{L}_M$ of type o. For each function $f \in \mathcal{L}_O$ of arity n, there is a function $f' \in \mathcal{L}_M$ of arity n and type $o \times \ldots \times o \to o$. For each proposition $q \in \mathcal{L}_O$, there is a constant $q' \in \mathcal{L}_M$ of type μ . For each predicate $p \in \mathcal{L}_O$ of arity n, there is a function $p' \in \mathcal{L}_M$ of arity n and type $o \times \ldots \times o \to \mu$. Finally, \mathcal{L}_M also includes functions & and if of type $\mu \times \mu \to \mu$, and a constant empty of type μ .

The representation of terms of \mathcal{L}_O in \mathcal{L}_M is defined inductively as follows.

- 1. A constant $a \in \mathcal{L}_O$ is represented by the constant $a' \in \mathcal{L}_M$.
- 2. A variable $x \in \mathcal{L}_O$ is represented by the constant $x' \in \mathcal{L}_M$.
- 3. A term $f(t_1, \ldots, t_n) \in \mathcal{L}_O$ is represented by the term $f'(\lceil t_1 \rceil, \ldots, \lceil t_n \rceil) \in \mathcal{L}_M$, where $\lceil t_1 \rceil, \ldots, \lceil t_n \rceil$ are the representations of the terms t_1, \ldots, t_n in \mathcal{L}_M .

The representation of formulas of \mathcal{L}_O in \mathcal{L}_M is defined inductively as follows.

- 1. A proposition $q \in \mathcal{L}_O$ is represented by the constant $q' \in \mathcal{L}_M$.
- 2. An atomic formula $p(t_1, \ldots, t_n) \in \mathcal{L}_O$ is represented by the term $p'(\lceil t_1 \rceil, \ldots, \lceil t_n \rceil) \in \mathcal{L}_M$, where $\lceil t_1 \rceil, \ldots, \lceil t_n \rceil$ are the representations of the terms t_1, \ldots, t_n in \mathcal{L}_M .
- 3. If $F \in \mathcal{L}_O$ and $G \in \mathcal{L}_O$ are formulas containing only the connectives \wedge and \leftarrow and no quantifiers, then $F \wedge G$ is represented by $\lceil F \rceil \& \lceil G \rceil \in \mathcal{L}_M$, and $F \leftarrow G$ is represented by $\lceil F \rceil$ if $\lceil G \rceil \in \mathcal{L}_M$, where $\lceil F \rceil$ and $\lceil G \rceil$ are the representations of F and G respectively in \mathcal{L}_M .

²Richer object theories can be represented if \mathcal{L}_M has additional function symbols for the representation of a larger set of quantifiers and connectives.

In standard logic programming notation, \leftarrow is also used as a unary connective. A definite clause $F \leftarrow \in \mathcal{L}_O$ is represented by $\lceil F \rceil$ if $empty \in \mathcal{L}_M$, and a definite goal $\leftarrow F \in \mathcal{L}_O$ is represented by empty if $\lceil F \rceil \in \mathcal{L}_M$.

Given the representation defined above, if \mathcal{L}_O contains the definite statement

$$p(f(y), x) \leftarrow q(y) \land r(a, x)$$

then its representation in \mathcal{L}_M is

$$p'(f'(y'), x')$$
 if $q'(y')$ & $r'(a', x')$

which is a term of type μ .

Many different representations are possible, and the choice of representation determines, to a large extent, the expressive power of the meta-language and its computational efficiency. At one extreme, the syntactic expressions of \mathcal{L}_O , even up to theories, might be represented by constants of \mathcal{L}_M . Constants that name compound expressions in this way are called quotation names. This approach has the advantages of simplicity and abstraction from the components of expressions of \mathcal{L}_O , but has limited expressiveness. The representation of theories by constants is proposed as a means for modelling the beliefs of agents in [46].

An alternative representation, more flexible than the one in the example above, can be obtained by explicitly representing the names of the symbols of \mathcal{L}_O by constants of \mathcal{L}_M . The meta-language the contains two distinguished function symbols, term and atom, both of arity 2, which take as first argument a constant representing the name of a function symbol or predicate symbol respectively, and as second argument a list of representations of terms. In this scheme, the representation of the atom $p(f(y), x) \in \mathcal{L}_O$ in \mathcal{L}_M is

$$atom(p', [term(f', [y']), x'])$$

where constant $p' \in \mathcal{L}_M$ represents the name of predicate $p \in \mathcal{L}_M$, and constant $f' \in \mathcal{L}_M$ represents the name of function $f \in \mathcal{L}_O$. The terms that represent expressions of \mathcal{L}_O in this scheme are called *structured-descriptive* names in [6]. Such structured representations have the advantage that variables of \mathcal{L}_M can appear in the first argument of term or atom, increasing

the possibilities for representing partially-known expressions of \mathcal{L}_{O} .

As pointed out in [82], for some special purpose applications it may be desirable for the representation to encode more than just the syntactic form of expressions of \mathcal{L}_O . For example, semantic information may also be encoded. Representations of this sort are application-specific.

Another important representation uses finite sequences of characters, such as strings or lists, to represent expressions. This representation is natural because it is the form in which an object program is normally entered using a text editor, or held in a file on a computer. For example, in a metalanguage that denotes strings by enclosing a sequence of characters between double-quotes, one representation of the formula

$$p(f(y), x) \leftarrow q(y) \land r(a, x)$$

is simply

$$"p(f(y),x) \leftarrow q(y) \& r(a,x)"$$

I refer to the representation of expressions of \mathcal{L}_O by sequences of characters as representation in concrete syntax. Of course, a given expression has many different representations in concrete syntax because space and formatting characters are not significant. This representation is impractical for most meta-programming tasks, because of the cost of decoding the strings; nevertheless, because it is succinct, it is valuable as a means of making a particular expression available to a meta-program, where it can be translated into a more abstract representation. Structured ground representations such as the examples above are referred to as abstract syntax, because they abstract away details of concrete syntax, such as formatting and punctuation, that are not relevant to the majority of meta-programming tasks.

The two example representations above demonstrate that, in general, it is simplest to regard a representation as an encoding of \mathcal{L}_O in \mathcal{L}_M . Given a term $E' \in \mathcal{L}_M$ that represents an expression $E \in \mathcal{L}_O$, the constants and function symbols of E' are usually given a free (Herbrand) interpretation which encodes expression E via a naming relation, rather than taking the intended interpretation of E' to be E. The need for this approach can be seen in the concrete syntax representation, where a string is a general-purpose type, and the intended interpretation of a string is actually a sequence of characters. The distinction is important in the case of meta-programs such

as compilers, which are more than just a relation between programs because they must also be able to handle syntactically incorrect input.

Much of the literature on meta-programming is concerned with the way a meta-theory M axiomatises the provability relation of an object theory O. The idea is that the meta-theory contains a predicate demo which satisfies the following proposition. If W is a formula of \mathcal{L}_O , then

$$O \vdash_{\mathcal{L}_O} W$$
 if and only if $M \vdash_{\mathcal{L}_M} demo([O], [W])$

This is a reflective principle: it ties the semantics of demo to the semantics of the object theory. The introduction of reflective principles is attributed to Feferman [27], and they were first used for meta-programming in Weyrauch's interactive theorem prover FOL [87]. Predicates like demo that capture the proof procedure of the object program, or some operation that is a component of its proof procedure such as unification, are called reflective predicates. In logic programming terminology, demo is called a meta-interpreter, or simply an interpreter.

Meta-programs such as compilers, debuggers, program transformers and analysers can be expected to be applicable to themselves or each other. Thus, a meta-language \mathcal{L}_M may need to be represented in another meta-language, $\mathcal{L}_{M'}$, say. In this way, a tower of meta-programs can be constructed, where the base of the tower is an object program which is not a meta-program, and each higher level of the tower contains a representation of the level below it. In the terminology of meta-programming, attention is usually restricted to two adjacent levels of the tower: a meta-level, and the level immediately below it, the object level, which may or may not be a meta-program itself. The phrases meta-level term, meta-variable and so on refer to expressions in the language of the meta-level; similarly, object-level term, object variable and so on refer to expressions in the language of the object level.

1.3 Approaches to meta-programming

In this section, I briefly survey the main approaches to meta-programming in logic programming languages.

Two important characteristics of meta-programs are that they may be self-applicable and they may be dynamic. A *self-applicable* meta-program

is one that can take itself as an object program. Common examples are compilers, program analysers and program transformers. A *dynamic* metaprogram is one which modifies the object program by adding or removing statements or declarations. A typical dynamic meta-program is a knowledge assimilator that manages a knowledge base that changes over time. Metaprograms may be both self-applicable and dynamic: a partial evaluator replaces object program statements with specialised versions, and may be expected to be able to specialise itself.

1.3.1 Non-ground Representations

Historically, the representation most widely used in logic programming is an untyped non-ground representation, where \mathcal{L}_O and \mathcal{L}_M are (unsorted) first-order languages and object-level variables are represented by meta-level variables. The remainder of the representation is usually straightforward, as in the following table.

Symbols of \mathcal{L}_O	$Symbols of {\cal L}_M$
Constant a	Constant a'
Function f of arity n	Function f' of arity n
Proposition q	Constant q'
Predicate p of arity n	Function p' of arity n
Connective \wedge	Binary function &
$Connective \leftarrow$	Binary function if
Empty formula	Constant empty

Non-ground representations have the significant advantage that representations of object variables are treated identically to meta-variables by the underlying implementation, so the search and unification mechanisms of the underlying implementation can be employed directly by reflective predicates. It is therefore possible to give a simple axiomatisation of a proof procedure for definite programs, as the definition of *solve* below.

```
\begin{array}{lll} solve(empty) & \leftarrow \\ solve(x \& y) & \leftarrow & solve(x) \land solve(y) \\ solve(x) & \leftarrow & clause(x \ if \ y) \land solve(y) \end{array}
```

The definition of clause gives the representation of the object program, and

has one fact for each of its statements. For example, if the object program is

$$\begin{array}{lll} p(a) & \leftarrow \\ p(g(x)) & \leftarrow \\ p(f(x)) & \leftarrow & p(x) \end{array}$$

then the definition of clause would be

$$clause(p'(a') \ if \ empty) \leftarrow \\ clause(p'(g'(x)) \ if \ empty) \leftarrow \\ clause(p'(f'(x)) \ if \ p'(x)) \leftarrow \\$$

This interpreter is called the vanilla interpreter, or sometimes the solve interpreter. As it stands, the definition has semantic problems because the appropriate reflection principle does not hold for solve. The reason is that the variables in the definition of clause are not constrained to range over representations of object-level terms only. As a result, atoms such as solve(p'(g'(p'(a')))) are a logical consequence of the solve program with this definition of clause.

There are two solutions to this problem in the literature. Hill and Lloyd [38] propose the use of a typed representation, like that described in Section 1.2 but non-ground, in which the meta-language has two types o and μ , where o is the type of terms that represent terms of the object language, and μ is the type of terms that represent formulas of the object language. It is then possible to prove the soundness and completeness of the typed version of solve. An alternative solution, that avoids the use of types, was given by Martens and De Schreye [56, 55]. Their approach proves the correctness of solve subject to the condition that the object program is language independent. This is a rather restrictive condition; informally, if O is a language independent object program and $\leftarrow G$ is a goal for O, then any computed answer for $O \cup \{\leftarrow G\}$ will instantiate all the variables of G to ground terms.

Once these semantic difficulties have been overcome, non-ground representations still place severe limitations on the expressiveness of metaprograms, provided such meta-programs are required to be declarative. The limitations stem from the fact that the meta-program cannot inspect or modify the variables of the object program, since these are represented by meta-variables. One important limitation is that meta-variables appearing

in a partially known representation of an object-level expression cannot be distinguished from representations of object-level variables. Another is that dynamic meta-programming is impossible, because there is no way to rename variables explicitly. Thus, the non-ground representation is suited only to a limited range of meta-programming tasks, and most of these are simple variants of *solve*, such as the extensions that generate a proof-tree or impose a depth bound on the search procedure. A variety of such meta-programs written in Prolog can be found in [73].

Declarative meta-programs based on the non-ground representation are perhaps best regarded as not being meta-programs at all in the true sense. Most of the enhancements that add functionality to solve can also be added directly to the object program. For example, the object program can be made to generate a proof tree, or perform a depth-bound check, by the addition of an extra argument to each predicate. Indeed, it is well known that such a modified object program can be generated automatically from a non-ground interpreter with the appropriate functionality by partial evaluation [28, 77]. Seen in this light, it is not surprising that types are required to make sure that the intended interpretation of solve is a model: many other familiar logic programs suffer the same problem, including, famously, append. Intended interpretations are naturally typed. The true advantage of enhanced interpreters for the non-ground representation is one of modularity: they provide a programming style that enables functionality such as the creation of a proof-tree to be separated cleanly from the object program, and in a re-usable form. Despite these reservations, I shall continue to use standard terminology, and refer to the vanilla interpreter as a meta-program.

1.3.2 Meta-programming in Prolog

The Prolog language provides support for meta-programming in the non-ground representation only, and encourages the use of a simple naming relation that is effectively the identity, because function symbols and predicate symbols are not distinguished. The limitations of the non-ground representation are overcome in Prolog by the provision of non-declarative "predicates" for meta-programming, the most important of which are var/1, assert/1 and retract/1. These predicates were first introduced in DEC-10 Prolog in 1978 [65]. It seems likely that the design of Prolog was influenced by that of LISP [57], in which programs and data have the same form and are

interchangeable, giving an elegant form of self-applicability. Both languages sacrifice declarative semantics for simple and expressive meta-programming.

The semantic problems of Prolog meta-programming are well known. The meta-predicate var/1 succeeds if its argument is an uninstantiated variable and fails otherwise, giving a meta-program the ability to inspect the instantiation state of variables. Under any logical reading that treats "," as conjunction the following two goals should be equivalent, yet when evaluated in left-to-right order the first succeeds and the second fails. ³

```
:- var(X), X=a.
:- X=a, var(X).
```

When $\mathtt{assert}(t)$ is executed, and t has the form of a Prolog program clause, this clause is added to the program. When $\mathtt{retract}(t)$ is executed, the first clause in the program that unifies with t is deleted. The effect of neither predicate is undone on backtracking. Together, these predicates enable dynamic meta-programming in Prolog. Again, of the following two goals, the first succeeds and the second fails (assuming the definition of p is empty before the execution of either goal).

```
:- assert(p(a)), p(X).
:- p(X), assert(p(a)).
```

Several of the other meta-logical predicates of Prolog, such as functor/3 and arg/3, can be understood declaratively.

The non-declarative meta-logical facilities of Prolog are powerful, but programs that make use of them have no simple model theory, and can only be understood procedurally. The consequences of this violation of the central thesis of declarative programming should not be underestimated. As a result, the wealth of theoretical work on the analysis and transformation of logic programs that depends on declarative semantics does not apply to the majority of Prolog meta-programs. They are also considerably more difficult to execute in parallel, because the non-declarative predicates place constraints on the order of evaluation. In addition, indiscriminate use of

³Since SLD-resolution with a left-to-right selection strategy is incomplete, the order of conjuncts in a goal may affect *termination* without preventing a program from being understood declaratively.

non-declarative predicates by inexperienced programmers can make a Prolog program as tangled and difficult to understand as the worst imperative program.

1.3.3 Reflection and Amalgamation

Another approach to meta-programming aims to increase expressiveness by incorporating reflective principles as rules of inference. The following pair of inference rules are typical, where W is a formula of \mathcal{L}_O and demo is a distinguished predicate of \mathcal{L}_M .

$$\frac{M \vdash_{\mathcal{L}_{M}} demo(\lceil O \rceil, \lceil W \rceil)}{O \vdash_{\mathcal{L}_{O}} W} \qquad \frac{O \vdash_{\mathcal{L}_{O}} W}{M \vdash_{\mathcal{L}_{M}} demo(\lceil O \rceil, \lceil W \rceil)}$$

As stated, these inference rules are improper [6], because they mix the metaand object languages. To fix this, it is at least required that \mathcal{L}_O be a subset of \mathcal{L}_M ; if it is a strict subset, the languages are said to be weakly amalgamated, and if $\mathcal{L}_O = \mathcal{L}_M$, they are said to be strongly amalgamated [36]. A naming relation is implicit in the reflective inference rules, so programming languages that make use of them must use a fixed representation. Such programming languages inevitably have non-standard semantics, because the semantics has to take reflective inferences into account. Implicit reflection also prohibits dynamic meta-programming, since the object theory is semantically linked to the meta-theory, so modifying the object theory also changes the denotation of demo. Meta-programming schemes with amalgamated languages and reflective inference rules are useful particularly for modelling modal logics of knowledge and belief, and complex knowledge representation tasks such as legal reasoning [4].

I use the term separated to denote a meta-programming scheme in which there is no fixed relationship between \mathcal{L}_O and \mathcal{L}_M , and no reflective rules of inference, so the semantics of the object program is not directly tied to the semantics of the meta-program. There is much confusion in the literature about the precise meaning of amalgamated and separated meta-programming; for example [74] talks of completely separated languages for which $\mathcal{L}_O \cap \mathcal{L}_M = \emptyset$. There seems little point to this restriction, except that paradoxes of self-reference cannot be expressed; separated meta-programming is certainly not limited to this case. In fact, the language of the object program need not be fixed, but symbols may be added and removed

by the meta-program. This is possible, because a ground representation can encode, not just the language of a particular object program, but all the syntactic constructs of a programming language. Thus, every possible object language is representable by the same encoding. In the separated scheme, meta-programs can be both self-applicable and dynamic; a self-applicable meta-program reasons about a copy of itself. For example, a partial evaluator may specialise itself; thus, at the beginning of the specialisation process, $\mathcal{L}_O = \mathcal{L}_M$, but during the process the meta-program may introduce new predicates into the object program, so that eventually $\mathcal{L}_O \supset \mathcal{L}_M$. Separated meta-programming thus provides considerable flexibility, which is needed for some kinds of application.

1.3.4 Ground Representations

Of all approaches to meta-programming so far invented, separated ground representations give the greatest opportunities for expressiveness, while being completely straightforward semantically. A meta-program written in first-order logic can take its semantics directly from standard interpretations and models for first-order theories. At the same time, it can operate on the syntax of the object program at any desired degree of granularity, from representations of complete theories down to the names of variables and symbols. Nevertheless, although much theoretical attention has been paid the the ground representation in the literature, there has been no prior attempt to provide extensive support for this style of meta-programming in a programming language, because it also has two significant drawbacks.

To see what these drawbacks are, consider the following interpreter for the ground representation, which is intended to be a simple axiomatisation of SLD-resolution, similar to the vanilla interpreter for the non-ground representation.

```
demo(prog, head \ if \ body) \qquad \leftarrow \\ empty(body) \\ demo(prog, head \ if \ body) \qquad \leftarrow \\ select(body, atom, rest) \ \land \\ member(stat, prog) \ \land \\ rename(stat, head \ if \ body, a \ if \ w) \ \land \\ mqu(atom, a, subst) \ \land \\
```

```
conjoin(w, rest, body1) \land apply(subst, head if body1, nresultant) \land demo(prog, nresultant)
```

In this representation, the function symbol if represents implication, and programs are represented as lists of representations of statements. Other details of the representation are not necessary for an understanding of this interpreter. The predicate demo is intended to be true when its first argument is the representation of a program, and the second is the representation of a resultant such that this program and resultant have an SLD-refutation. The first clause of demo states that a resultant with an empty body has an SLD-refutation. The second clause states that a resultant has an SLD-refutation if:

- 1. an atom is selected from the body of the resultant;
- 2. a statement is selected from the program;
- 3. the statement is renamed, so it contains only variables distinct from those in the resultant;
- 4. an mgu can be found for the selected atom and the head of the renamed clause;
- 5. the body of the renamed statement is conjoined with the remainder of the body of the resultant to form the body of a new resultant;
- 6. the mgu is applied to this new resultant;
- 7. the new resultant with the mgu applied has an SLD-refutation.

This interpreter is more complex than the vanilla interpreter, and instead of making use of the search and unification mechanism of the underlying system, it is defined in terms of seven high-level meta-predicates that perform explicit tasks such as unification and renaming variables. The ground representation devolves all responsibility for the management of variables onto the meta-program. A programmer in a language such as Prolog, which has no support for the ground representation, would have to implement all these predicates in order to have a working interpreter. This is the first

drawback: without good support, interpreters for the ground representation require a relatively large amount of code to implement.

Interpreted programs always suffer a loss of efficiency in comparison to direct execution, largely because interpreters are not object-program specific, and so the optimisations normally made by a compiler for the object program are not available to the interpreter. This is the price that is paid for the power of interpreters to monitor and modify the execution of the object program. The second drawback of the ground representation is that, when realised in a real programming language such as Prolog, this interpreter adds a great deal more⁴ computational overhead than the vanilla interpreter does, because of the explicit management of variables.

Many meta-programs are dynamic, updating the object program by adding and removing statements. In the interpreter above, the object program was represented by a list of representations of statements. This simple representation shows how the ground representation enables dynamic meta-programming: the object program can be updated by simply adding and removing elements from the list. This form of dynamic meta-programming was elaborated in [37].

1.4 Related work

An in-depth survey of the state of the art in meta-programming was written by Hill and Gallagher [36]. This work covers every aspect of meta-programming, including the non-ground and ground representations, self-applicable and dynamic meta-programming, reflective predicates and amalgamated languages, and the specialisation of interpreters by partial evaluation. Barklund [6] contributed another useful survey.

The use of a ground representation in logic originates with the mathematician Kurt Gödel, after whom the Gödel language is named. Gödel used a representation scheme based on an encoding of formulas as natural numbers in the proof of his famous incompleteness theorem [29, 35].

Many of the key ideas about meta-programming in logic programming can be found in the seminal paper by Bowen and Kowalski [11]. These authors show how a ground representation can be used to define a pred-

⁴How much more will be seen in Section 4.

icate demo that represents the provability relation for an object language and satisfies a reflective principle. They also propose the use of reflective principles as inference rules, and discuss applications of meta-programming to knowledge base management and non-monotonic reasoning. The paper shows that no axiomatisation of provability for an object theory can also represent unprovability by satisfying

$$O \not\vdash_{\mathcal{L}_O} W$$
 if and only if $M \vdash_{\mathcal{L}_M} \neg demo(\lceil O \rceil, \lceil W \rceil)$

This result follows from the undecidability of first-order logic, and is related to the undecidability of the Halting Problem [80].

The paper by Hill and Lloyd [38] gives a theoretical foundation for metaprogramming in logic programming, for both the ground and non-ground representations, by proving the correctness of simple interpreters for both. The work of Subrahmanian [74] provides a similar theoretical foundation. Hill and Lloyd went on to demonstrate [37] how the ground representation can be used for dynamic meta-programming.

A number of logic programming language variants have been developed with the idea of making meta-programming more expressive. MetaProlog, by Bowen and Weinberg [12], was intended as a practical demonstration of the ideas of Bowen and Kowalski. In MetaProlog, object programs are given names, and predicates are provided to add a clause to, drop a clause from, or prove a goal with respect to a given object program. The representation is however non-ground, and explicit quantifiers are used to delimit the scope of object variables. For many meta-programming tasks, MetaProlog programs must still resort to the non-declarative predicates of Prolog.

QuoteLog (or 'LOG) [19] is a meta-programming system based on Prolog that provides a ground representation. It uses both quotation names, which are constants, and structured-descriptive names to represent object level expressions, and has a mechanism for translating between the two naming schemes. QuoteLog offers little support for the complex operations necessary for meta-programming in the ground representation, and does not seem to have been developed very far.

Reflective Prolog is a language based on pure Prolog that incorporates reflective inference rules, designed by Costantini and Lanzarone [23]. Reflective Prolog uses a ground representation, and has types in the sense that variables ranging over representations of object-level formulas have a

different syntax from other variables. Reflective inferences are performed automatically by the system using a sophisticated control mechanism, by application of implicit functions for representing and de-representing formulas. Programs are limited to Horn clauses only, and the language has a non-standard semantics given by the Least Reflective Herbrand Model. Because of its reflective inferences, Reflective Prolog appears to be unsuitable for dynamic meta-programming. It also provides little support for the ground representation beyond the ability to make reflective inferences.

Barklund's language Alloy [9, 4] is based on the ideas of Reflective Prolog, but has named object theories and object-to-meta reflection is explicit. Alloy has been applied to the representation of legal knowledge. The language is still under development.

Jiang's Ambivalent Logic [44] is not a programming language, but a logic intended to provide a theoretical foundation for the style of meta-programming commonly used in Prolog, where there is no distinction between function symbols and predicate symbols, and the same symbol can serve in both capacities. Ambivalent logic provides a justification for Prolog statements such as the following:

solve(X) :- X.

The correctness of the vanilla interpreter in ambivalent logic is proved in [43]. Ambivalent logic takes a non-ground approach to meta-programming; a ground representation is still necessary to obtain the fine degree of control over object-level syntax that is needed for many meta-programming applications.

A paper by van Harmelen [82] argues that the naming relations used in meta-programming systems should be definable by the programmer. The reasons for this view are compelling: a definable naming relation can be used to encode pragmatic and semantic information relevant to the application, enhancing the expressiveness of the meta-language. There are difficulties with this approach however. Firstly, the naming relation is not directly expressible in the meta-language, it is only expressible in a higher-level meta-language in which both object language and meta-language are represented. Secondly, the provision of a support library for meta-programming essentially requires a fixed representation. A meta-program is always free to translate from one representation to another; for example, a parser is a

meta-program that translates from concrete syntax to an abstract syntax representation.

Meta-programming in functional programming languages has received much less attention. A rough equivalent to Prolog's non-declarative treatment of a non-ground representation can be seen in LISP [57], where an expression can either be evaluated, or quoted and treated as a representation; this approach clearly has semantic problems. Modern functional languages such as Haskell [66] have types and modules, and take great care to have clean declarative semantics. As a result, the use of a ground representation for meta-programming in these languages is routine; see for example [68, 48]. However, Haskell has no standardised support library for meta-programming, and as a result is underutilised in meta-programming applications, for which it is otherwise well suited.

1.5 Overview

The remainder of this thesis is structured as follows.

Chapter 2 describes the Gödel language, defining the technical concepts required for an understanding of Gödel meta-programming. The reader is then taken on a tour of Gödel's meta-programming facilities, and an extended example is presented.

In Chapter 3, I explain the implementation of Gödel, starting with a description of the data structures that are used to represent object programs. The second part of this chapter shows how Gödel programs are compiled into Prolog, and introduces SLDQE-resolution, which is a computational model for the evaluation of universally quantified implication formulas used in the compilation of Gödel statements.

An analysis of efficiency issues in Gödel meta-programs is presented in Chapter 4, in the form of an evolutionary sequence of interpreters. The results of some experiments in the partial evaluation of interpreters are given, and a case is made that specific support from the low-level implementation can improve the efficiency of interpreters still further.

In conclusion, I give an evaluation of the meta-programming facilities and techniques developed in this thesis. Ways in which the lessons learned from this research might be applied to declarative languages of the future are examined. Finally, the results are summarised.

Chapter 2

The Gödel Language

Gödel is designed both to be a full-scale programming language with rich facilities to support software engineering, and also to be a language for meta-programming. Meta-programming in Gödel means manipulating complete Gödel object programs in the ground representation. Therefore, every additional language feature complicates the representation, and hence complicates the meta-programming aspect of the language. In Gödel a fine balance has been struck between the provision of all the essential features of a modern programming language and the retention of a reasonably simple and comprehensible library of predicates for meta-programming.

In Section 2.1 I give a technical description of the main features of the Gödel language, together with a few example programs. Knowledge of these features is essential to an understanding of Gödel's meta-programming library. Note that for the purposes of this thesis, a slightly simplified version of the language is presented. For example, I say little about pruning or set processing, and do not discuss constraint solving. The full syntax and definition of the language is contained in [39]; the language definitions and some of the examples in this section are taken from that source. Some of the material in this section is also taken from a tutorial on the Gödel language [13]. Note that the formal definition of the language in [39] is entirely respected by the design and implementation of the meta-programming library, and the language definition ultimately determines the precise meaning of the predicates that the library provides.

Section 2.2 contains a tour of the meta-programming facilities of Gödel, and a few example meta-programs. It is not exhaustive; a complete listing

2.1 Features of Gödel

The main features of the Gödel language relevant to this thesis are types, the extended statement syntax, modules, and control. Each of these is described in turn in this section. Note that the syntax of Gödel differs fundamentally from the syntax of Prolog, in that Gödel variables begin with a lowercase letter, and other symbols such as constants, functions and predicates begin with an uppercase letter.

2.1.1 Types

The Gödel type system is based on many-sorted logic with parametric polymorphism. In this section I introduce some simple Gödel programs, and describe the Gödel type system. The formal syntax for Gödel is not given here; instead then syntax is demonstrated by example. A complete grammar can be found in [39].

The simplest form of a Gödel program consists of a single module, which starts with the keyword MODULE followed by the name of the module. The remainder of the module contains declarations and statements. The declarations define the language of the program, and the statements are formulas in this language. An example of a simple Gödel program is module M3 in Figure 2.1.

Gödel declarations each begin with one of the six keywords BASE, CONSTRUCTOR, CONSTANT, FUNCTION, PROPOSITION and PREDICATE. These declare the symbols of the language, which belong to six *categories*: bases, constructors, constants, functions, propositions, and predicates. The symbols in these categories all begin with an uppercase letter. The language of a Gödel program also includes type variables, called *parameters*, and variables. Both parameters and variables begin with a lowercase letter, and do not have declarations. Together, the parameters, variables and six categories of symbol define a *polymorphic many-sorted language* in the sense of Appendix A.

Types in Gödel are formed from bases, parameters and constructors in the same way that terms are formed from constants, variables and functions in standard logic. Thus a base is a type, and a parameter is a type, and if τ_1, \ldots, τ_n are types and c is a constructor of arity n, then $c(\tau_1, \ldots, \tau_n)$ is a type. The declaration for a constructor gives the arity of the constructor; the constructor List in Figure 2.1 has arity 1. A parameter can be instantiated to any type, and an instance of a type can be created by instantiating the parameters that appear in it. A type containing no parameters is called a ground type or monotype.

The declaration for a constant gives the type of the constant. Thus the constant Sunday in Figure 2.1 has type Day, and the constant Nil has type List(a).

The declaration for a function symbol gives its domain type, which is the tuple of types of its arguments, and its range type, which is the type that the function maps these arguments into. So the function Cons in module M3 maps a pair of arguments of type a and List(a) into the type List(a). Similarly, the declaration for a predicate gives the tuple of types of its arguments.

A symbol is *polymorphic* if its declaration contains a parameter; otherwise, it is *monomorphic*. A polymorphic symbol can be understood as standing for a collection of monomorphic symbols. For example, the declaration for the constant Nil in module M3 could be regarded as as a countably infinite set of declarations for constants with types List(t), where t ranges over all ground types.

Every base, constructor, constant, function, proposition and predicate in a Gödel program must have a declaration. The name given to a symbol in its declaration is its *declared name*. Several symbols in the same category and with the same type can be declared together, as is done for the constants of type Day in M3.

Statements in Gödel are formulas in polymorphic many-sorted logic. The definition of a term and a formula in polymorphic many-sorted logic is given in Appendix A. In essence, it must be possible to find an instantiation of type parameters so that the type of every argument to a predicate or function symbol is equal to the type given for that argument position in the declared type of the predicate or function symbol, and to assign types to variables so that every occurrence of the same variable has the same type. For example, given the declarations in module M3, the term Cons(Monday, Cons(x, Nil)) has type List(Day), and the variable x is assigned the type Day in this term. The expression

```
MODULE M3.
BASE Day, Person.
CONSTANT
   Monday, Tuesday, Wednesday, Thursday,
   Friday, Saturday, Sunday : Day;
   Fred, Bill, Mary : Person.
CONSTRUCTOR List/1.
CONSTANT Nil : List(a).
FUNCTION Cons : a * List(a) -> List(a).
PREDICATE Append : List(a) * List(a) * List(a).
Append(Nil, x, x).
Append(Cons(u, x), y, Cons(u, z)) <-
   Append(x, y, z).
            Append3 : List(a) * List(a) * List(a) * List(a).
PREDICATE
Append3(x,y,z,u) \leftarrow
   Append(x,y,w) &
   Append(w,z,u).
```

Figure 2.1: A simple Gödel program

Cons (Monday, Cons (Fred, Nil)) is not a valid term, because constants Monday and Fred have different types. The expression Cons(y, y) is also not a valid term, because no type can be found for y: it cannot have type a and type List(a) simultaneously. It is decidable whether or not an expression is a formula in polymorphic many-sorted logic; a proof can be found in [41]. The usual definitions of interpretation, model, logical consequence and so on can be extended to polymorphic many-sorted logic; see for example Appendix A of [39].

Parameters and variables do not have declarations in Gödel. The type of a variable (and any corresponding quantifier) is deduced from its context. The same variable may have different types in different statements.

A statement in a Gödel module is either an atom A, or a formula A <- W where A is an atom and W is a formula. A is called the head and W the body of the statement. Any variables in A and any free variables in W are assumed to be universally quantified at the front of the statement. Every statement in a module must be a formula in the language of the module, and satisfy the additional condition given below. The set of statements in a module with a given predicate in the head is called the definition of that predicate.

A goal for a Gödel program is a formula of the form <- W, where W is a formula called the *body* of the goal. Any free variables in W are assumed to be universally quantified at the front of the goal. W must be a formula in the goal language of the program, which for a program such as M3 consisting of a single, simple module is the language of the module.

A precise definition of the language and goal language of a module depends on a full definition of the Gödel module system, and is given in Section 2.1.3.

A desirable property of a type system is that programs can be executed without the need for run-time type checking. For this to be so in Gödel, two conditions need to be imposed on Gödel programs.

- 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.
- A declaration for a function is *transparent* if every parameter appearing in the declaration also appears in the range type.

If every statement in a Gödel program satisfies the head condition, and every

function declaration is transparent, it can be shown that no type-checking is needed if a program and goal are executed using standard proof procedures such as SLDNF-resolution [41, 61]. These conditions are natural, and rarely inconvenience programmers in practice.

An instructive example of the value of types in logic programming is given by a program for flattening a nested list. A nested list is a list, the elements of which are either non-list terms or nested lists. The flattened form of a nested list is a list containing all the non-list terms in the nested list, and no nested list terms. Of course, such a nested list structure cannot be directly represented in Gödel, because the Gödel type system demands that all elements of a list have the same type. This appears to be a disadvantage of the Gödel type system, but such an assessment is superficial, as the discussion below demonstrates. Notice that an analogue of the nested list structure can be constructed, by using additional function symbols to label the elements of the list as either nested lists or non-list terms, and map them to a single type.

In the example, the system module Lists is imported by means of an IMPORT declaration. Importing this module provides a type constructor List/1 and makes the conventional Prolog square-bracket notation for lists available. A formal description of the Gödel module system is given in Section 2.1.3.

The Gödel program in Figure 2.2 uses difference lists to construct the flattened form of a Gödel "nest". The function N labels a list element that is a nest. The function E labels a non-list term in the nest, and because the program is polymorphic, all non-list terms in the structure must have the same type. If it is required to store a known range of distinct types of term in a nest, a non-polymorphic version can easily be constructed, with a different function to label each type of term in place of function E. A typical goal for Flatten is

$$\leftarrow$$
 Flatten(N([E(1), N([N([]), E(2)])]), x)

which gives the answer x = [1,2].

It is interesting to compare the program in Figure 2.2 with similar program written in Prolog. There are many examples in Prolog textbooks, the one in Figure 2.3 is taken from [73]. Since Prolog is untyped, the nested list can be represented directly.

```
MODULE Flatten.
IMPORT Lists.
CONSTRUCTOR Nest/1.
FUNCTION
   E : a -> Nest(a);
   N : List(Nest(a)) \rightarrow Nest(a).
PREDICATE
   Flatten : Nest(a) * List(a);
   Flatten1 : Nest(a) * List(a) * List(a);
   Flatten2 : List(Nest(a)) * List(a) * List(a).
Flatten(nest, list) <-</pre>
   Flatten1(nest, list, []).
Flatten1(E(x), [x|list_t], list_t).
Flatten1(N(x), list, list_t) \leftarrow
   Flatten2(x, list, list_t).
Flatten2([], list_t, list_t).
Flatten2([x|xs], list, list_t) <-</pre>
   Flatten1(x, list, list_1) &
   Flatten2(xs, list_1, list_t).
```

Figure 2.2: A Gödel program for flattening a nest

```
flatten(Xs, Ys) :-
   flatten_dl(Xs, Ys\[]).

flatten_dl([X|Xs], Ys\Zs :-
   flatten_dl(X, Ys\Ys1),
   flatten_dl(Xs, Ys1\Zs).

flatten_dl(X, [X|Xs]\Xs) :-
   constant(X), X \= [].

flatten_dl([], Xs\Xs).
```

Figure 2.3: A Prolog program for flattening a nested list

The Prolog program depends on the use of the predicate (constant)¹ to identify non-list terms. A correct implementation of constant in Prolog should generate a run-time error if it is called when its argument is an uninstantiated variable, so the program will not work with a partial data structure. It will also fail if given a list containing compound non-list terms; correcting this fault requires additional tests. Finally, notice that the selection of one of the three clauses for flatten_d1/3 for a resolution step cannot be made by examining the top-level function symbol in first argument of the call, because the first argument of the second clause for flatten_d1 is a variable. Thus, the execution of the program is less efficient than it should be, because WAM-style first argument indexing cannot be used.

The Gödel program has none of these faults. The terms stored in the nest can be uninstantiated variables or compound terms without affecting the operation of the program. The goal <- Flatten(x, [1,2,3]) generates all the possible nest structures with the flattened form [1,2,3] (there are infinitely many). First-argument indexing can be used to select the clauses of Flatten1 and Flatten2. All these benefits stem from the presence of function E in the structure, and the presence of this symbol was originally required to satisfy the type system. Of course, the same program can be written in Prolog, using an analogous function symbol to label the non-list

¹The Prolog predicate (constant) can be viewed as either a type predicate or a metalogical test.

elements in the nested structure.

The program in Figure 2.2 is rather cluttered, however, and can be improved by observing that a nested list is really a binary tree. The Gödel nest structure has two kinds of non-leaf nodes, represented by function symbols Cons and N, and these can be condensed into a single node with function T, say. The result of this simplification is the Gödel program of Figure 2.4. A typical goal for the Flatten2 module is

which gives the answer x = [1,2].

This example is intended to demonstrate that the discipline of a type system is beneficial to logic programming. Without it, even experienced programmers can be tempted to design poor data structures, and resort to using meta-logical or non-declarative features of Prolog as a remedy. Although types are not required to solve the problem of flattening a nested list, the correct program eluded the authors of [73]; on the other hand, Gödel's strong typing makes the correct version almost unavoidable. The type system also has many other benefits, such as the detection of many common programming errors at compile-time. Types add to the complexity of meta-programming in Gödel, as shall be seen in Sections 2.2, 3.1 and 3.2. However, typed meta-programs are significantly easier to understand and maintain, and the presence of types in the object program is useful in many meta-programming applications; for example learning algorithms can make use of types to reduce their search space by limiting attention to well-typed formulas.

2.1.2 Formulas

Connectives

The body of a statement in Gödel is an arbitrary formula. It can include any of the connectives & (conjunction), \/ (disjunction), ~ (negation), <- (left implication), -> (right implication) and <-> (equivalence). The universal quantifier is ALL and the existential quantifier is SOME; both quantifiers are followed by a list of quantified variables and the formula in the scope of the quantifier. These connectives and quantifiers have their usual meaning.

Variable names beginning with underscore (_) have a special meaning in Gödel. When such a variable appears in the body of a statement or goal, it

```
MODULE Flatten2.
IMPORT Lists.
CONSTRUCTOR Nest/1.
CONSTANT Nil : Nest(a).
FUNCTION E : a -> Nest(a);
         T : Nest(a) * Nest(a) \rightarrow Nest(a).
PREDICATE Flatten : Nest(a) * List(a);
          Flatten1 : Nest(a) * List(a) * List(a).
Flatten(nest, list) <-</pre>
   Flatten1(nest, list, []).
Flatten1(E(x), [x|list_t], list_t).
Flatten1(T(x, xs), list, list_t) <-</pre>
   Flatten1(x, list, list_1) &
   Flatten1(xs, list_1, list_t).
Flatten1(Nil, list, list).
```

Figure 2.4: A Gödel program for flattening a simplified nest

stands for a unique variable that is existentially quantified immediately in front of the atom in which it appears. Thus the goal

<- SOME [z] Mother(x,z) &
$$\sim$$
 SOME [y] Father(y,x).

can be written

<- Mother(
$$x$$
,_) & ~ Father(_, x).

A variable beginning with an underscore in the head of a statement stands for a unique variable universally quantified at the front of the statement.

Conditionals

Gödel also provides a set of conditional constructs. The basic form of a conditional is

IF
$$C$$
 THEN V ELSE W

where C, V and W are formulas. This formula is equivalent to

$$(C \& V) \setminus (C \& W)$$

If the above formula was used directly in a statement, the formula C would be evaluated twice. The conditional is implemented in a way that avoids this inefficiency. This is the motivation for conditionals in Gödel: they enable a choice to be made without recomputing the condition. In fact, conditionals are a benign form of pruning. Soundness is ensured if the conditional formula is not selected until the condition C contains no free variables.

A richer form of conditional is also available, which allows the formulas C and V to share variables. This construct is due to Naish [62]. It is:

IF SOME
$$[x_1, \ldots, x_n]$$
 C THEN V ELSE W

This formula is equivalent to

(SOME
$$[x_1,\ldots,x_n]$$
 ($C \& V$)) \/
("SOME $[x_1,\ldots,x_n]$ $C \& W$)

Note that the scope of the existential quantifier in the condition is non-standard, in that it extends over both C and V.

Both conditionals have a short form in which the ELSE part is omitted. In this short form, if the condition is fails, the conditional itself succeeds.

Operators

Unary functions and predicates may employ prefix or postfix notation, and binary functions and predicates may employ infix notation. This is achieved by adding an *indicator* to a FUNCTION or PREDICATE declaration, which gives precedence and associativity information for the operator. Indicators for binary functions have the form xFx(N), xFy(N) or xFy(N), where N is the precedence and xFy, yFx and xFx denote right-, left- and non-associativity respectively. Indicators for unary functions have the form xF(N) or yF(N) (prefix), and Fx(N) or Fy(N) (postfix). Indicators for predicates have the form zPz, Pz or zP, denoting infix, prefix or postfix respectively.

As an example, the following declaration for the function + declares it to be left-associative with precedence 510.

```
FUNCTION + : yFx(510) : Integer * Integer -> Integer.
```

Built in Propositions and Predicates

Gödel has two built-in propositions, True and False, and two built-in predicates, = and ~=. These symbols are available in every module without being explicitly imported. The definition of the proposition True is just the atom True, and False has the empty definition.

Equality and disequality have the following declaration:

```
PREDICATE =, ^{\sim} = : zPz : a * a.
```

For user defined types, equality in Gödel is just syntactic equality. Certain system types, such as Integer and Rational have a more sophisticated equality theory, which takes account of the standard interpretations of those types. Disequality is defined to be the negation of equality.

2.1.3 Modules

Gödel has a simple module system. A Gödel module consists of two parts, an export part and a local part. The *local part* of a module is indicated by a LOCAL or MODULE module declaration. The *export part* of a module is indicated by an EXPORT or CLOSED module declaration. In these declarations, the *part keywords* LOCAL, MODULE, EXPORT and CLOSED are followed by the name of the module. In fact, a module may have a local and an export part, or just a local part, or just an export part. If a module consists *only* of a

local part, then this is indicated by using a MODULE declaration instead of a LOCAL declaration. The meaning of a CLOSED declaration used in place of an EXPORT declaration will be explained later in this section.

IMPORT declarations, name imported modules. The export part of a module contains zero or more IMPORT declarations, language declarations, and control declarations. The local part of a module contains zero or more IMPORT declarations, language declarations, control declarations, and statements. Control declarations are defined in Section 2.1.4.

A definition of the concepts of importation, accessibility and language for Gödel modules follows. These definition are simplified compared to those in [39]; in particular, the definition of importation does not mention the *type lifting* facility of Gödel's module system. The definitions are, however, consistent with the presentation of the ground representation of Gödel programs in Section 3.2.

A part of a module refers to a module N if it contains a declaration of the form

IMPORT N.

The local part (resp. export part) of a module declares a symbol if it contains a declaration for the symbol.

A module *declares* a symbol if either the local or export part of the module declares the symbol.

A part of a module M imports a symbol S from a module N if the export part of N declares S, and either

- the part of the module refers to N, or
- there is a module L such that the part of M refers to L, and the export part of L imports S from N.

Importation in Gödel is therefore transitive.

A module M imports a symbol S from a module N if either the local or export part of M imports S from N.

A symbol S is accessible to the local (resp., export) part of a module if the module (resp., export part of the module) either

- declares the symbol, or
- imports the symbol from some module N.

A module *exports* a symbol if the symbol is accessible to the export part of the module.

Five module conditions, M1 through M5, are enforced by the Gödel system to ensure the integrity of Gödel programs. Condition M1 uses the relation depends upon, which is defined to be the transitive closure of refers to.

- M1: No module may depend upon itself.
- M2: For every name appearing in a part of a module, there must be a symbol having that name accessible to that part.
- M3: Distinct symbols cannot be declared in the same module with the same category, name, and arity.
- M4: In a module, the type in a constant declaration or the range type in a function declaration must be either a base type declared in the module or a type with a top-level constructor declared in the module.
- M5: A module must declare every proposition or predicate defined in that module.

Condition M3 is a restriction on the naming of symbols. It is weak enough to allow symbols to be overloaded. For example, the name – can be used for both unary and binary minus, because the arities differ; the name + can be used for addition of integer and rational numbers, and for the set union operation, provided each of these functions is declared in different module. When an identifier appearing in a statement in a Gödel module could refer to one of several symbols with the same declared name, the Gödel parser uses all available type and indicator information in an attempt to resolve the ambiguity. Usually, only one of the possible symbols will yield a well-typed statement. If the overloading cannot be resolved, an error is reported.

Condition M4 ensures that a module cannot extend an imported type by declaring new constant or function symbols of that type. For example, a user module that imports Integers cannot declare a new constant of type Integer. Condition M5 prevents the definition of a predicate from being split across several modules.

A program consists of a set of modules $\{M_i\}_{i=0}^n$ $(n \geq 0)$, where $\{M_i\}_{i=1}^n$ is the set of modules upon which M_0 depends, and where each module satisfies the module conditions given above. The module M_0 is called the main module of the program.

A module is *closed* if the module declaration in its export part contains a CLOSED keyword. A module is *open* if it is not closed. Although in principle all propositions and predicates declared in a module have a definition in Gödel code, closed modules allow some propositions or predicates to be implemented by another method. The main difference between a closed module and an open module concerns meta-programs. When a meta-program manipulates the ground representation of an object program containing a closed module, the declarations and statements in the local part of the closed module are not accessible to the meta-program. The other significant difference between closed and open modules affects the goal language of a program with a closed main module, as defined below.

It is now possible to give definitions of the language of a module and the goal language of a program.

The language of a module M in a program is the polymorphic many-sorted language given by the language declarations of all symbols accessible to the local part of M.

The export language of a module M in a program is the polymorphic many-sorted language given by the language declarations of all symbols accessible to the export part of M.

The goal language of a program P is the language of the main module M_0 of P, if M_0 is open, or the export language of M_0 , if M_0 is closed.

Thus, a statement in a module M in a program P is a polymorphic many-sorted formula in the language of M in P, and a goal for P is a polymorphic many-sorted formula in the goal language of P. In fact, this definition needs to be extended to allow commits to appear in the body of a statement or goal; this extension can be found in Chapter 7 of [39]. In essence, only one bar commit is permitted in a clause, and bracketed commits are only allowed around top-level conjuncts in the body.

Before I come to the the tour of the meta-programming facilities of Gödel in Section 2.2, it is useful to give some definitions related to the flat form of a Gödel program, since it is the flat form that is represented in Gödel's ground representation. Module condition M3 implies that a symbol in a

Gödel program is not uniquely identified by its declared name, but can be uniquely identified by the quadruple consisting of the name of the module in which the symbol is declared, the declared name of the symbol, its category, and its arity. The arity of bases, constants and propositions is taken to be zero for this purpose. The quadruple (M, S, C, A), where M is the name of a module, S is the declared name of a symbol declared in module M, C is the category of S, and A is the arity of S, is called the *flat name* of the symbol.

The *flat form* of a program P is obtained from P by replacing each occurrence of the declared name of a symbol in P by the flat name of the symbol.

The flat language of a program P is the polymorphic many-sorted language given by the set of language declarations in the flat form of P.

The flat language of a module M in a program P is the subset of the flat language of P given by the all the symbols in the language of M in P.

The flat export language of a module M in a program P is the subset of the flat language of P given by all the symbols in the export language of M in P.

The following definition is used in Gödel meta-programming to distinguish the treatment of symbols potentially declared in the local part of a closed module from that of symbols declared in open modules.

A term is *opaque* if it is a constant declared in the local part of a closed module, or it has a top-level function symbol declared in the local part of a closed module; otherwise it is *non-opaque*. Opaque types and atoms are defined analogously.

2.1.4 Control

Gödel has a sophisticated control mechanism. The first responsibility of control in Gödel is to ensure that certain constructs, such as negation and conditionals, are only evaluated under conditions when such evaluation is sound. Gödel has a pruning operator called *commit*, which can be used to prune subtrees from the search tree. Gödel also allows the programmer control over the order in which atoms are selected for expansion by means of DELAY declarations.

Commit

The Gödel commit has two simple forms: the bar commit, written |, which is close in meaning to the commit of the concurrent logic programming languages, and the one-solution commit, {...}, which encloses a formula.

The bar commit can be written in place of &, and declaratively its meaning is just conjunction. There can be at most one | in a statement. 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 cut. The procedural meaning of | is that only one solution is found for the formula in its scope and all other branches in a search tree arising from the other statements in the predicate definition which contain a | are pruned.

The scope of the one-solution commit {...} is the formula between the { and the }. When the scope is solved, all possible alternative solutions to the scope are pruned away. The one-solution commit can be used in goals or in the bodies of statements in a program.

The class of programs containing | is not closed under unfolding. For this reason, Gödel has a more powerful pruning operator, the *labeled commit* $\{...\}_n$, where n is a positive integer called the *commit label*. This form of commit includes the | and $\{...\}$ as special cases, and gives a class of programs closed under the usual program transformations. Simple commits are transformed into labelled commits enclosing the same scope by assigning suitable labels. Within the definition of a predicate, every labelled commit replacing a | is assigned the same label, whereas every labelled commit replacing $\{...\}$ is assigned a unique label. The procedural meaning of the labelled commit can be found in [40].

The commit operator is useful only under certain relatively rare circumstances; most well-written Gödel programs do not contain commits.

DELAY Declarations

DELAY declarations are syntactic variants of when declarations, which are due to Naish [79]. The main reasons for using DELAY declarations are that they can assist efficiency by enabling coroutining, can ensure termination, and can be used to control pruning. They are particularly useful in conjunction with a module system, as they can be used to prevent the use of library

predicates in modes in which they are incomplete due to pruning or do not terminate.

As an initial example, consider the following definition for Append3.

```
PREDICATE

Append3: List(a) * List(a) * List(a) * List(a).

Append3(xs, ys, zs, us) <-

Append(xs, ys, ws) &

Append(ws, zs, us).
```

The predicate Append has the usual definition. It is exported by the system module Lists, where it has the DELAY declaration

```
DELAY Append(x, _, y) UNTIL NONVAR(x) \/ NONVAR(y).
```

This DELAY declaration means that any call to Append is suspended until either the first argument or the third argument is instantiated to a non-variable term. Without the DELAY declaration for Append, and using a "left-most literal" computation rule, the goal

```
<- Append3(x, y, z, [1,2,3]).
```

gives all possible ways to split [1,2,3] and then goes into an infinite loop. With the DELAY declaration, the computation gives all possible ways to split the list and then terminates.

It is interesting to observe that the DELAY declaration given above does not ensure the termination of Append in all circumstances. For example, every resolvent in an SLD-derivation for the goal

```
<- Append([1|y], z, y)
```

satisfies the DELAY declaration, and yet the computation does not terminate. Thus NOVAR conditions can have subtle effects, and their use to ensure termination requires care; the effect of GROUND conditions is much more comprehensible.

I now informally define the syntax and semantics of DELAY declarations. A DELAY declaration has the form

```
DELAY Atom UNTIL Cond.
```

where Atom is an atom, which is not a proposition, and Cond is a $delay\ condition$, which is constructed as follows. NONVAR(x), GROUND(x), and TRUE, where x is a variable that appears in Atom, are delay conditions. If D and E are delay conditions, then $(D \setminus E)$ and (D & E) are delay conditions (the parentheses may be omitted). The reserved word & stands for conjunction, \setminus stands for disjunction, TRUE stands for the truth value true, NONVAR is true if and only if its argument is a non-variable term, and GROUND is true if and only if its argument is a ground term.

A DELAY declaration for a predicate can only appear in the module where the predicate is declared. A predicate may have several DELAY declarations, and these can be compacted into a single one of the form

```
DELAY Atom_1 UNTIL Cond_1; \vdots Atom_n UNTIL Cond_n.
```

Two conditions are placed on *Atom*.

D1: No pair of *Atoms* in the set of **DELAY** declarations for a predicate can have a common instance.

D2: An atom in the head of a DELAY declaration must not contain repeated variables.

Condition D1 simplifies the semantics of DELAY declarations. Without it, a situation could arise where one declaration allowed a call to proceed, while another delayed it. Condition D2 allows an implementation to avoid the occur check when checking DELAY declarations. Note that each occurrence of an underscore (_) in the head of a DELAY declaration denotes a unique variable.

Now suppose an atom A is an instance by a substitution θ of an Atom in a DELAY declaration. Then A satisfies the corresponding condition Cond in this DELAY declaration if, when θ is applied to the variables in Cond, the resulting condition has truth value true using the above meanings given to the various reserved words. Otherwise, A does not satisfy the corresponding condition.

Then DELAY declarations cause calls to be delayed according to the following rules:

Figure 2.5: The definition of Sorted

- An atom in a goal is delayed if it has a common instance with some *Atom* in a DELAY declaration but is not an instance of this *Atom*.
- An atom in a goal is delayed if it is an instance of an *Atom* in a DELAY declaration but does not satisfy the corresponding condition *Cond*.

Thus DELAY declarations give programmers some influence over the computation rule and can be used to ensure that certain calls will not be run until they are sufficiently instantiated.

A more complex example is given by the definition of the predicate Sorted in Figure 2.5. Sorted is intended to be true when its argument is a list of integers in ascending order.

The meaning of the DELAY declaration for Sorted is as follows. If the argument of a call to Sorted is not an instance of either [] or $[_|x]$, then the call delays. Thus the call Sorted(x) delays. If the argument of a call to Sorted has the form [S|T], where T is a variable, then the call delays. Thus the call Sorted([1|x]) delays. If the argument of a call to Sorted is either [] or has the form [S|T], where T is not a variable, then the call can proceed. Thus the calls Sorted([]), Sorted([x]), and Sorted([3,2|x]) can proceed.

2.1.5 System Modules

Gödel has a rich set of system modules, which provide various data types and operations on those types. Among these system modules are Integers, Rationals, Floats, Lists, Strings and Sets. Some of these types have special equality theories. For types such as List and Set, importing the module makes additional syntax available that is appropriate to the type.

I discuss the Sets module briefly because it is one of the most interesting, and sets are used in the example at the end of this chapter. Sets provides a type constructor Set/1. In a module that imports Sets, the syntax $\{\}$ denotes the empty set, and $\{t_1, \ldots, t_n\}$ denotes an extensional set containing values t_1, \ldots, t_n . The Set type has a special equality theory, so that the goals

$$\leftarrow \{5, 6\} = \{6, 5\}$$

and

$$\leftarrow \{5, 6\} = \{5, 6, 6\}$$

both succeed. **Sets** also provides a binary infix predicate **In** for set membership, and binary infix functions *, + and \ for set-theoretic intersection, union and difference respectively.

Intensional set terms are written $\{T: W\}$, where T is a term with free variables y_1, \ldots, y_n , say, and W is a formula which has y_1, \ldots, y_n amongst its free variables. Informally, $\{T: W\}$ means "the set of all instances of T corresponding to the instances of W which are true".

As an example of a goal involving an intensional set, the following

$$-x = \{\{n^2 : 1 = < n = < m\} : 1 = < m = < 5\}.$$

gives the answer $x = \{\{1\}, \{1,4\}, \{1,4,9\}, \{1,4,9,16\}, \{1,4,9,16,25\}\}.$

Gödel's system modules include a group of modules Syntax, Programs, ProgramsIO and Scripts which provide types for meta-programming. These modules are described in some detail in Section 2.2.

2.1.6 Semantics

I give only a very brief account of the semantics of Gödel; a detailed description is available in [39].

To obtain the declarative semantics of a Gödel program, its statements are first transformed into a *canonical form*. This transformation essentially

removes all commits, inserts explicit existential quantifiers for underscore variables, and replaces conditionals and intensional sets by formulas giving their meaning. The declarative semantics of the program is then given by the completion of this transformed program (Appendix A), together with any special equality theories for data types appearing in the program.

The procedural semantics of Gödel are not specified precisely. Instead, [39] gives a notion of an abstract search tree to which an implementation must conform. The abstract search tree gives considerable freedom to the implementation, but constrains it to respect DELAY declarations and commits, so that programmers have some control over the computation rule, and pruning behaves as expected.

2.2 Meta-programming in Gödel

The meta-programming facilities of Gödel were initially designed by Hill and Lloyd, from their work in [38, 37]. Their design was developed and improved considerably through the work presented in this thesis.

In this section, I give a tour of the meta-programming facilities of Gödel, followed by an extended example of a Gödel meta-program. The predicates described here are those used in subsequent examples. The ground representation in Gödel is provided as a collection of abstract data types, by the system modules Syntax and Programs. The Programs module imports Syntax in its export part, so a user program has only to import the Programs module to obtain access to a rich set of predicates for manipulating representations of the syntactic structures of object programs, such as types, terms, formulas, and complete programs. These representations are all terms in the language of the meta-program.

The abstract data type approach completely hides the details of the representation from the programmer, and this approach has several important advantages. The first is simplicity: Gödel is a full-featured programming language, and the representation of a Gödel program as a term in a metalanguage is necessarily complex. If the representation were explicit, the authors of Gödel meta-programs would have to learn the meaning and use of the large number of function symbols that make up the representation. Instead, Gödel provides a handful of abstract types, and a comprehensive library of well-documented and high-level predicates for operating on those

types. A second advantage is integrity: because an abstract type only permits a fixed set of well-defined operations to be performed on it, it is possible for the system to guarantee that instances of the type always possess certain properties. For example, a term representing a Gödel program can always be translated into a syntactically correct Gödel program in concrete syntax. In contrast, an explicit representation leaves the programmer free to construct a term that does not represent a syntactically valid object program, within the limitations of the Gödel type system. An explicit representation would therefore increase the amount of validity checking that must be performed by meta-level library predicates, if they are to have precise semantics, with a consequent loss of efficiency. A third advantage is that the abstract type creates the possibility of an alternative implementation, other than in Gödel code, provided the declarative reading of the exported library predicates is preserved. Finally, there is the usual benefit of abstract types, which is that Gödel meta-programs are guaranteed not to depend on the details of the implementation, so the implementation can be changed freely provided the semantics of the interface is preserved, and programs can be ported unchanged between different implementations of the Gödel system modules.

There is one disadvantage to the abstract data-type approach, which is that pattern matching on explicit function symbols is natural to the normal logic programming style. The lack of pattern matching makes programs slightly less succinct, and costs the possibility of optimising clause selection by indexing on function symbols, although it is possible to regain the loss of efficiency by partial evaluation. It is well-known that abstract types are incompatible with pattern matching; for example, a solution for the Haskell programming language is proposed in [85]. For types such as those that make up the ground representation of Gödel programs, this disadvantage is less significant than the advantages given above.

2.2.1 The Syntax Module

The Syntax module provides programmers with facilities for manipulating representations of the syntactic expressions of the object program, such as types, terms, formulas and substitutions. Syntax has no knowledge of the representation of higher level constructs such as modules and programs, which are formed from collections of these syntactic expressions, and possible

uses for Syntax are not limited to the representation of object programs that conform precisely to the definition of a Gödel program. The types provided by Syntax could be used as a basis for the representation of a variety of different program-like structures, provided these employ a similar syntax to Gödel at the level of terms and formulas. For example, a general first-order theory can be represented as a list of representations of formulas, using the formula representation from Syntax.

The system module Strings is imported into the export part of Syntax, and Strings indirectly imports Lists and Integers, making these three system types available to modules that import Syntax.

The abstract types provided by Syntax are Name, Type, Term, Formula, TypeSubst, TermSubst and VarTyping. There are also two concrete types, FunctionInd and PredicateInd, which are used to represent function indicators and predicate indicators respectively. The declarations for the constants and functions of these types can be found in Appendix B; they are a straightforward represention of precedence and associativity information for operators.

A term of type Name represents the name of a symbol. Note that is the flat name rather than the declared name of a symbol that is represented, because the declared name does not contain sufficient information to identify a symbol uniquely within a program. See Section 2.1.3 for an explanation of flat names. This is not simply due to overloading, but comes about because representations of all symbols in the object program are accessible to the meta-program, including those hidden in the local parts of open modules, and without flattening name clashes can occur.

Because Syntax has no concept of modules, and flat names incorporate module names, Syntax does not provide a facility for creating terms of type Name, or for accessing their components. Instead this facility is provided by a collection of predicates in the Programs module, described in Section 2.2.2.

The abstract type Type is provided for the representation of types, and the abstract type Term is provided for the representation of terms of the object language. Terms of both these types contain terms of type Name as components.

The abstract type Formula is intended for the representation of formulas in the object language; terms of type Formula may contain subterms of type Term. Terms of type Formula do not contain a direct representation

of every syntactic construct of the Gödel language, because bar commits, one solution commits, and underscore variables are preprocessed away by the parser when the representation of a formula is created. When parsing a formula, the Gödel parser translates bar commits and one solution commits into the full labelled commit, by assigning appropriate labels as explained in Section 2.1.4. Underscore variables are removed by the insertion of explicit existential quantifiers. In every atom containing n underscore variables, n > 0, these are replaced by unique new variables v_1, \ldots, v_n and the quantifier SOME v_1, \ldots, v_n is inserted before the atom. This preprocessing has a simplifying effect for meta-programs, by reducing the number of cases that need to be considered, and avoiding the need to for special treatment of underscore variables.

Usually, terms of type Type, Term and Formula represent types, terms and formulas in a given object language, and predicates in Syntax that manipulate these representations could check that this is the case. To do so, each predicate would need an argument containing a representation of the object language in some form, and it must be remembered that every Gödel program defines several different languages, such as the language of a module in the program, or the goal language of the program. Performing such validity checks is expensive, adds complexity, and places unnecessary restrictions on the meta-program. Validity checking can also have unexpected consequences. For example, a term may not be valid in a given language because it contains an ambiguous symbol, but when the same term appears as an argument in an atom, its type may be restricted sufficiently to resolve the ambiguity. For these reasons, the predicates in Syntax do not check type, term and formula representations for validity in particular languages; instead, it is the responsibility of the meta-program to ensure that only valid representations are constructed.

Terms of type TypeSubst and TermSubst represent type substitutions and term substitutions, respectively. A type substitution is a finite mapping from parameters to types, and a term substitution is a finite mapping from variables to terms.

The last abstract type provided by Syntax is VarTyping. A term of this type represents a variable typing, which is a finite mapping of variables to types. Variable typings are useful for type checking, particularly when type checking a subformula which contains variables whose types are constrained

by their appearances elsewhere in the formula. A variable typing then provides a context in which the types of the variables in the subformula can be determined.

I now discuss a representative selection of the predicates exported by the Syntax module. These predicates can be divided into groups with similar functions. A complete listing of the export part of Syntax can be found in Appendix B. Throughout this discussion, the phrase "true when" is used as an abbreviation for "true in the intended interpretation if and only if". The notation $\lceil X \rceil$ is used to denote the representation of an object-level syntactic expression X, where X can be a type, term, formula or program. It is also convenient to use informal language occasionally, and speak of the representation of an object-level expression as though it were the expression itself; the meaning should always be clear from the context.

The first group of predicates is concerned with the representation of formulas. They can be used in any mode, and do not have DELAY declarations. Using these predicates, formulas can be constructed and combined by the addition of connectives, or broken up into subformulas. Partial formula representations can also be created, with meta-variables in place of unknown subformulas. There is one such predicate for every connective in Gödel, including conditionals and commit. A selection appears below.

```
PREDICATE

And : Formula * Formula;

AndWithEmpty : Formula * Formula * Formula;

Some : List(Term) * Formula * Formula;

Commit : Integer * Formula * Formula;
```

And is true when its first and second arguments are representations of formulas, and its third argument represents their conjunction. AndWithEmpty is a useful predicate for forming conjunctions; its meaning is like that of And, but differs if either of the first two arguments is the representation of the empty formula. If the first argument of AndWithEmpty is the representation of a formula W and the second argument is the representation of a formula V, then if W is empty the third argument is $\lceil V \rceil$, otherwise if V is empty then the third argument is $\lceil W \& V \rceil$. Some is true when its first argument is a list of the representations of variables

 v_1, \ldots, v_n , its second argument is the representation of a formula W, and its third argument is the representation of the formula SOME $[v_1, \ldots, v_n]$ W. Commit is true when its first argument is a commit label n, its second argument is the representation of a formula W, and its third argument is the representation of the formula $\{W\}_n$, in which W is enclosed in the scope of a commit with label n.

The next group of predicates deals with the representation of types, terms and atoms. For example, a term can be constructed from a function symbol and list of arguments, or the function symbol end arguments can be extracted from a given term. These predicates can be used in any mode, and can be used to construct partial representations.

```
PREDICATE

BaseType : Type * Name;

FunctionTerm : Term * Name * List(Term);

PredicateAtom : Formula * Name * List(Term);

IntensionalSet : Term * Formula * Term.
```

The predicate BaseType is true when its first argument is the representation of a base type, and its second argument is the representation of the flat name of this base type. FunctionTerm is true when its first argument is the representation of a non-opaque² term with a function symbol at the top level, its second argument is the representation of the flat name of this function, and its third argument is the list of representations of the arguments of this function in the term. PredicateAtom is true when its first argument is the representation of a non-opaque atom with a predicate symbol at the top level, its second argument is the representation of the flat name of this predicate, and its third argument is a list of representations of the arguments of this predicate in the atom.

Although the meaning of an intensional set in Gödel is a formula, sets take the place of terms syntactically, and are represented by terms of type Term. IntensionalSet is true when its first argument represents a term T, its second argument represents a formula W, and its third argument represents the intensional set $\{T:W\}$.

Another group of predicates provides facilities for managing parame-

²Opaque terms and atoms are defined in Section 2.1.3.

ters and variables. The representation of variables is a critical aspect of the ground representation. It is desirable that the representation retains the variable names that appear in the object program source, but metaprograms, and especially interpreters, must be able to create new and unique variable names easily, for purposes such as standardisation apart. To meet these requirements, the representation of a variable in Gödel has two components, a root which is a string, and an index, which is an integer. The representation of an object-level variable \mathbf{x} is given root " \mathbf{x} " and index 0. The maximum index of all variables occurring in a formula is a compact representation of a superset of the variables appearing in the formula. A variable guaranteed not to occur in the formula is easily created by allocating an index greater than this maximum.

```
PREDICATE

VariableName : Term * String * Integer;

TermVariables : Term * List(term);

FormulaMaxVarIndex : List(Formula) * Integer;
```

VariableName is true when its first argument is the representation of a variable, its second argument is the root of the variable, and its third argument is the index of the variable. This predicate can be used to create variables or extract the root and index from a variable. TermVariables is true when its first argument is the representation of a term, and its second argument is a list (in some definite order) of the representations of all the free variables occurring in the term. Note that the potential presence of a subterm that is an intensional set means that a term can contain bound variables. FormulaMaxvarIndex is true when its first argument is a list of formulas, and its second argument is one more than the maximum index of all the variables (free or bound) occurring in all the formulas in the list. Both TermVariables and FormulaMaxVarIndex suspend until their first argument is ground. Syntax also provides analogous predicates for the management of parameters in types.

Meta-programs often need to determine whether or not an object-level expression satisfies some syntactic condition. Syntax exports a large group of predicates for this purpose. All the predicates in this group take a single argument, and suspend until this argument is ground.

```
PREDICATE

Variable : Term;

OpaqueTerm : Term;

GroundAtom : Formula;

NormalStatement : Formula.
```

For example, Variable is true when its argument is the representation of a variable. This predicate is a declarative counterpart to var/1 in Prolog. OpaqueTerm is true when its argument is the representation of an opaque term. GroundAtom is true when its argument represents an atom containing no free variables. NormalStatement is true when its argument represents a formula that is syntactically a normal statement, but as previously stated, this statement is not checked for validity in an object language.

There are also predicates in Syntax that test whether a formula is a standard body, normal body, definite body, standard resultant, normal resultant, definite resultant, standard goal, normal goal, or definite goal. In each case, commits may only bracket top-level conjuncts in a body, body of a resultant, or body of a goal. A precise definition of each kind of formula, in the form of a grammar, can be found in Chapter 11 of [39].

Handling representations of term substitutions is essential to programming interpreters in Gödel; this topic is discussed at length in Section 4.

```
PREDICATE

EmptyTermSubst : TermSubst;

ApplySubstToFormula : Formula * TermSubst * Formula;

ComposeTermSubsts : TermSubst * TermSubst * TermSubst;

RestrictSubstToFormula : Formula * TermSubst * TermSubst.
```

EmptyTermSubst is true when its argument is the representation of the empty substitution. ApplySubstToFormula is true when its first argument is the representation of a formula W, its second argument is the representation of a substitution θ , and its third argument is the representation of the formula $W\theta$. ComposeTermSubst is true when its first argument is the representation of a substitution σ , its second argument is the representation of a substitution θ , and its third argument is the representation of their composition $\sigma\theta$. The use of this predicate is discouraged, for reasons explained

in Section 4. RestrictSubstToFormula is true when its first argument is the representation of a formula W, its second argument is the representation of a substitution θ , and its third argument is the representation of a substitution obtained from θ by restricting it to the free variables of W. This predicate is useful for obtaining a computed answer according to the definition in [50].

The ability to simulate unification for representations of terms and atoms is another important facility provided by the ground representation.

```
PREDICATE

UnifyTerms : Term * Term * TermSubst * TermSubst;
UnifyAtoms : Formula * Formula * TermSubst * TermSubst.
```

UnifyTerms is true when its first two arguments represent terms S and T respectively, its third argument is the representation of a substitution σ , and its fourth argument is the representation of the substitution $\sigma\theta$ where θ is a specific mgu for S and T. UnifyTerms provides a convenient method for adding a single binding to a substitution, as in the case when S is a variable and T is a term not containing S. UnifyAtoms is analogous to UnifyTerms, except that the first two arguments are representations of atoms.

The predicate StandardiseFormula below is specifically designed for the operation of standardisation apart in an interpreter simulating resolution. As described above, variable indexes provide a convenient mechanism for this purpose.

```
PREDICATE

StandardiseFormula : Formula * Integer * Integer * Formula.
```

StandardiseFormula is true when its first argument is the representation of a formula W, its second and third arguments are variable indexes i and j, with i > 0, j > 0 and $i \le j$, and its fourth argument is the representation of the formula obtained from W by systematically replacing the variables in W by variables with indexes $\ge i$ and < j. StandardiseFormula suspends until its first two arguments are ground.

The last predicate described here is Resolve, which is a sophisticated predicate specifically designed to perform one step in an SLD-derivation.

Resolve was designed by Gurr [32, 15]; the rationale behind it is explained in detail in Section 4.2.4.

```
PREDICATE

Resolve : Formula * Formula * Integer * Integer

* TermSubst * TermSubst * Formula.
```

Resolve is true when the first argument is the representation of an atom A, the second argument is the representation of a statement $H \leftarrow W$ the third and fourth arguments are variable indexes i and j, with i > 0, j > 0, and i < j, the fifth argument is the representation of a substitution σ , the sixth argument is the representation of a substitution $\sigma\theta$, and the seventh argument is the representation of a formula $W'\theta$, where

- 1. all the variables in $H \leftarrow W$ are systematically replaced by variables with indexes $\geq i$ and < j, to give the statement $H' \leftarrow W'$, and
- 2. θ is a specific mgu of $\{A, H'\}$.

A call to Resolve suspends until its first, second, third and fifth arguments are ground.

2.2.2 The Programs Module

The Gödel system module Programs provides a library of predicates for manipulating the ground representation of Gödel object programs. An object program is represented as a single term of an abstract type, so the details of the representation are concealed. This term encodes the import declarations, symbol declarations, statements, control declarations, and module structure of the object program. This encoding is at the level of abstract syntax; for example, the textual layout of the object program is not represented. Nevertheless, the representation contains sufficient information to allow the recovery of a syntactically correct textual version of the object program. The representation of a Gödel program is created by the Gödel parser; a deparser (or decompiler) is also available for recovering an object program from its representation. Both the parser and deparser can be accessed through predicates exported by the module Programs IO, described in Section 2.2.3.

Syntax is imported into the export part of Programs, and the abstract types of Syntax are used for the representation of the types and formulas within the object program representation.

Programs exports an abstract type Program, and Gödel object programs are represented by terms of this type. Note that the integrity checks performed by the predicates that create and modify terms of type Program, together with the DELAY declarations that enforce modes for those predicates, guarantee that it is impossible for a Gödel meta-program to create a term of type Program that does not represent a syntactically valid object program.

Programs also exports another abstract type, Condition, which is the type of terms representing the condition in a DELAY declaration. The meaning and use of Condition is straightforward, and it is not discussed further. The last type provided by Programs is the concrete type ModulePart, which is the type of constants representing the keywords EXPORT, LOCAL, CLOSED and MODULE which appear in Gödel module declarations.

```
CONSTANT Export, Local, Closed, Module : ModulePart.
```

I now describe a representative selection of the predicates exported by the Programs module. The first of these is NewProgram, which can be used to create a new, empty program representation, or a representation of a program containing only system modules.

```
PREDICATE

NewProgram : String * Program.
```

NewProgram is true when its first argument is the name of a module, and its second argument is the representation of a program, which has a module with this name as its main module. If the name is that of one of the system modules, then the program consists of this system module and all the modules that it depends upon, otherwise the program consists of a single, empty module with both an export part and a local part.

A group of predicates is concerned with determining if an object-level syntactic expression (type, term or formula) is valid in some particular object language.

```
PREDICATE

TypeInProgram : Program * Type;

TermInModule : Program * String * ModulePart * VarTyping *

* Term * Type * VarTyping.
```

Type InProgram is true when its first argument is the representation of a program, and its second argument is the representation of a type in the flat language of this program. TypeInProgram is the simplest predicate in this group; TermInModule is among the most complex. TermInModule is true when its first argument is the representation of a program P, the second argument is the name of a module M in this program, the third argument represents a part keyword (Section 2.1.3) of this module, the fourth represents a variable typing, the fifth is the representation of a term in a language L, the sixth is the representation of the type of this term, and the seventh is the representation of the variable typing obtained by combining the variable typing in the fourth argument with the types inferred for all free variables in the term. Language L is the flat language of M in P if the part keyword represented by the fourth argument is LOCAL or MODULE, or the flat export language of M in P if the part keyword is EXPORT or CLOSED. Note that the types of any variables appearing in the term are constrained by the variable typing represented in the fourth argument.

Another group of predicates in Programs gives access to the Gödel parser and deparser for types, terms and formulas.

```
PREDICATE

StringToProgramFormula : Program * String * String

* List(Formula);

ProgramFormulaToString : Program * String * Formula

* String.
```

StringToProgramFormula is true when its first argument is the representation of a program, its second argument is the name of a module in this program, its third argument is the string representation of a formula in the language of this module, and the fourth argument is the list of representations of formulas which have the string representation in the third argument.

Note that there may be several such formulas, because the parser may be unable to resolve all ambiguities due to overloaded symbols; all possible interpretations of the string are then returned.

ProgramFormulaToString is true when its first argument is the representation of a program, its second argument is the name of a module in this program, its third argument is the representation of a formula (subject to conditions described shortly), and its fourth argument is the representation of this formula as a string. The formula must not contain propositions or predicates not in the language of the module. If the formula contains a subterm not in the language of the module, the type of such a subterm must be in this language. In this case, the character sequence <type>, where type is the type of the subterm, appears in place of the subterm. This scheme is ideal for presenting the result of an object-level computation to a user, while ensuring that the details of any abstract data types in the object program are not revealed. Both of these predicates need access to the program representation in order to obtain precedence and associativity information for operators.

As mentioned in Section 2.2.1, Syntax does not have access to the components of terms of type Name, which represent flat names. Instead, representations of names are created and decomposed using a group of predicates in Programs.

```
PREDICATE

ProgramFunctionName : Program * String * String

* Integer * Name.
```

ProgramFunctionName is true when its first argument is the representation of a program, the second argument is the name of a module in this program, the third argument is the declared name of a function (which may or may not have a declaration in this module), the fourth argument is the arity of this function, and the fifth argument is the representation of the flat name of this function.

The next group of predicates concerns the module structure of the object program. Two such predicates, used in the examples in this thesis, are given below.

PREDICATE

```
MainModuleInProgram : Program * String.
DeclaredInClosedModule : Program * String * Formula.
```

MainModuleInProgram is true when its first argument is the representation of a program, and its second argument is the name of the main module of this program. DeclaredInClosedModule is true when its first argument is the representation of a program, the second argument is the name of a closed module in this program, and the third argument is the representation of an atom whose proposition or predicate is declared in the export part of this module.

There are predicates in **Programs** that give access to the object-level declarations of all six categories of symbol. I give one example of such a predicate.

PREDICATE

```
FunctionInModule : Program * String * ModulePart * Name
    * FunctionInd * List(Type) * Type * String.
```

FunctionInModule is true when its first argument is the representation of a program, the second argument is the name of a module in this program, the third argument is the representation of a part keyword in this module, the fourth argument represents the flat name of a function accessible in this part of this module, the fifth argument represents the indicator for this function, the sixth argument is the list of representations of the domain types of this function, the seventh argument is the representation of the domain type of this function, and the eighth argument is the name of the module in which this function is declared.

Access to the statements of the object program is provided by the predicates StatementMatchatom and DefinitionInProgram.

```
PREDICATE

StatementMatchAtom : Program * String * Formula

* Formula;

DefinitionInProgram : Program * String * Name

* List(Formula).
```

StatementMatchAtom is true when its first argument is the representation of a program, its second argument is the name of a module in this program, the third argument is the representation of an atom whose proposition or predicate is declared in this module, and the fourth argument is the representation of a statement in the definition of this proposition or predicate. Note that StatementMatchAtom is nondeterministic, and a call to it suspends until the first argument is ground.

DefinitionInProgram gives access to the entire definition of a proposition or predicate. It differs from StatementMatchAtom in that its third argument is the representation of the flat name of a proposition or predicate declared in the module named in the second argument, and the fourth argument is the list of statements in the definition of this proposition or predicate, in a definite fixed order.

Dynamic meta-programming is an important feature of the ground representation. The predicates in this group can be used to modify an object program, by adding or removing symbol declarations, statements, control declarations, and import declarations.

```
PREDICATE

DeleteProgramConstant : Program * String * ModulePart

* Name * Type * Program;

InsertStatement : Program * String * Formula * Program;
```

For example, DeleteProgramConstant deletes a constant declaration from a program. This predicate is true when its first argument is the representation of a program, the second argument is the name of a module in this program, the third argument represents a part keyword of the module, the fourth argument represents the flat name of a constant declared in this part of this module, the fifth argument is the representation of the type of this constant, and the sixth argument represents the program in the first argument with

this constant declaration deleted. A call to DeleteProgramConstant suspends until the first three arguments are ground. To preserve the integrity of the object program, a symbol declarations can only be deleted if there is no declaration or statement that makes use of the symbol.

InsertStatement is true when its first argument is the representation of a program, its second argument is the name of a module in this program, its third argument is the representation of a statement in the language of this module that satisfies the head condition, and its fourth argument represents the program in the first argument with this statement added to this module. A corresponding predicate DeleteStatement exists.

The final group of predicates provides interpreters for object programs and goals. A representative example is the predicate Succeed.

```
PREDICATE

Succeed: Program * Formula * TermSubst.
```

Succeed is true when its first argument is the representation of a program, the second argument is the representation of the body of a goal for this program, and the third argument represents a computed answer for this program and goal.

2.2.3 Other Meta-modules

The explanation of the Programs module above leaves an important question unanswered: how is the representation of an object program obtained in the first place? That is the function of the system module Programs IO. This module exports the following predicates (among others).

```
PREDICATE ProgramCompile : String * Program;

ProgramDecompile : Program.
```

ProgramCompile takes as its first argument the name of the main module of a program, and instantiates its second argument to the ground representation of this program. That is, it reads the source files of this main module and all the modules it depends upon, and parses them to construct the Program term. ProgramDecompile is the opposite of ProgramCompile: it takes the representation of a program, and writes source files for all the user

modules in this program. Both predicates perform input-output operations, and are therefore not declarative.

Another system module, Scripts, plays a specialised role in Gödel metaprogramming. It is designed specifically to support partial evaluation. The problem addressed by Scripts is that partial evaluation and module systems conflict with each other. When a program is specialised by partial evaluation, symbols are often promoted from modules low down in the hierarchy into statements in modules higher up where those symbols are not accessible. Thus the result of partially evaluating a Gödel program is not a legal Gödel program and cannot be represented as a Program term. This is where the concept of a script comes in. A script is obtained from the flat form of a Gödel program by discarding the module structure, and concatenating the flat forms of the declarations and statements. A script has an open part and a closed part: the closed part consists of the declarations and statements that appeared in closed modules in the original program; the remaining statements and declarations form the open part. Only the open part may be modified.

The Scripts module provides an abstract type Script for the representation of scripts, and a predicate

```
PREDICATE ProgramToScript : Program * Script.
```

for the construction of the representation of a script given the representation of a program. Predicates for dynamic meta-programming such as

```
PREDICATE InsertStatement : Script * Formula * Script.
```

are also provided, so that the script representation can be updated.

A partial evaluator for Gödel programs starts with the representation of a program, and produces a script representation containing the specialised statements. A module ScriptsIO allows the term representing a script to be written to a file, and a system utility, the *script compiler*, is provided to compile this term into an executable Gödel program.

2.2.4 An Extended Example

I now present an extended example of Gödel meta-programming, in the form of a knowledge assimilator for a Knowledge Based System. The example is interesting because it demonstrates many common features of Gödel meta-programs, including the manipulation of the syntax of object-level expressions, the use of a simple interpreter to gather information about an object level computation, and the use of a reflective interpreter for the full Gödel language to evaluate object-level goals. The definitions and knowledge assimilation algorithm in this example are taken from [30]. First, I introduce some terminology.

A knowledge base is a first order theory, restricted to allow efficient theorem proving techniques for processing queries. For this example, the knowledge base is restricted to be a definite program.

An integrity constraint theory is a first order theory.

A knowledge base satisfies an integrity constraint W if W is a logical consequence of the completion of the knowledge base; otherwise it violates the constraint.

An assimilator is a program that updates the knowledge base, ensuring that appropriate integrity constraint theories are satisfied.

The assimilator in this example implements an insertion procedure, which takes an atom, a knowledge base, and an integrity constraint theory, and works out how to update the knowledge base so that the atom is a logical consequence of the updated knowledge base, and the updated knowledge base still satisfies the integrity constraint theory.

A retraction (resp., assertion) of a definite clause S from (resp., into) a definite program P consists of the removal of S from (resp., addition of S to) the set of statements P. An action is a retraction or an assertion.

The retraction of S is denoted by retract(S) and the assertion of S by assert(S).

A transaction is a finite sequence of actions, denoted by $[action(S_1), \ldots, action(S_n)]$, where action is either retract or assert, and each S_i is a definite clause.

Given a definite program P and a transaction t, the updated program resulting from applying the actions in t to P is denoted by t(P).

The intuitive idea behind the algorithm for the insertion procedure is to build an SLD-tree for the goal $\leftarrow A$, where A is to be inserted, and by

```
Input: a definite program P, an atom A and an integrity constraint theory C such that P satisfies C and \exists (A) is not a logical consequence of P.

Output: \mathcal{T} = \{t : t \text{ is a transaction, } t(P) \text{ satisfies } C, \text{ and } \exists (A) \text{ is a logical consequence of } t(P)\}.

begin

T := \text{a (finite) SLD-tree for } P \cup \{\leftarrow A\};

\mathcal{T}_t := \{[assert(A_1), \dots, assert(A_n)] : \leftarrow A_1 \wedge \dots \wedge A_n \text{ is a goal in } T\};

\mathcal{T} := \{t : t \in \mathcal{T}_t \text{ and } t(P) \text{ satisfies } C\};
end
```

Figure 2.6: An assimilation algorithm for a Knowledge Based System

inserting facts into the program complete a branch in the tree so that it becomes a success branch. The insertion algorithm is given in Figure 2.6.

The correctness of this algorithm is proved in [30]. This algorithm generates assertions only; [31] gives an algorithm based on this one for inserting an atom into a normal program, and this can produce transactions involving retractions.

The Gödel module Assimilator implements the insertion procedure. The top-level of the assimilator is the predicate InsertionTransactions in Figure 2.7. This predicate takes four arguments, where the first three are inputs. The first argument is the representation of a knowledge base and integrity constraint theory; these two theories are represented as a single Program term, because doing so simplifies the assimilator. The second argument is a representation of an atom to be inserted into the knowledge base, and the third argument is the representation of the body of a goal, typically a single proposition. The integrity constraint theory is assumed to be organised so that, if the computation of this goal succeeds in the combined knowledge base and integrity constraint theory, all the integrity constraints are satisfied. The fourth argument is the list of valid transactions returned by the insertion procedure. Within the transactions, assertions are

```
MODULE Assimilator.
BASE Action.
% A = assert, R = retract
FUNCTION A, R : Formula -> Action.
PREDICATE InsertionTransactions :
                        % Knowledge base.
   Program
 * Formula
                        % An atom to be inserted.
 * Formula
                        % The body of a goal for checking
                        % the integrity constraints.
 * List(List(Action)). % Some transactions that achieve
                        % the insertion.
InsertionTransactions(prog, atom, ics, transs) <-</pre>
   RawInsertionTransactions(
      {goal : Goal(prog, atom, goal)},
      raw_transs) &
   Filter(prog, ics, raw_transs, transs).
```

Figure 2.7: Generating and filtering transactions

distinguished by the function A and retractions by the function \mathbb{R}^3 .

InsertionTransactions first computes a set of goals in the SLD-tree for the atom, using an intensional set. The goals are computed by an interpreter Goal. The predicate RawInsertionTransactions translates these goals into transactions, and the predicate Filter discards the transactions that cause the updated program to violate an integrity constraint.

The algorithm leaves open which SLD-tree for $P \cup \{\leftarrow A\}$, and which goals in the tree, are to be used to generate the transactions. Building the SLD-tree using a carefully chosen computation rule can improve the set of candidate transactions, by increasing the likelihood that one or more trans-

³As already noted, this algorithm does not generate retractions.

actions will lead to an updated knowledge base that satisfies the integrity constraints. The computation rule used in this example selects an atom only if it unifies with the head of a statement in the knowledge base. The SLD-tree is extended until no selectable atoms remain in the leaves (assuming there are no infinite branches) and the goals are collected from the leaf nodes. These goals therefore contain only atoms that are not a logical consequence of the knowledge base. There are many other good ways to construct appropriate transactions; this is not necessarily the best, but it works well for this example.

The predicate Goal in Figure 2.8 implements an interpreter that returns a representation of the body of the last goal in an SLD-derivation constructed using the above computation rule. Chapter 4 contains a detailed discussion of some similar Gödel interpreters; I only briefly sketch the operation of this interpreter here. The main loop of the interpreter is the predicate Goal1; this predicate accumulates a conjunction of unselectable atoms in its fifth argument, and returns this conjunction when the conjunction of potentially selectable atoms in its second argument becomes empty. The integer argument is a variable index, used by Resolve for standardisation apart, and given its initial value by the call to FormulaMaxVarIndex in the definition of Goal. The argument of type TermSubst represents the substitution accumulated at each resolution step, according to the usual definition of SLD-resolution. An initial empty substitution is created in Goal before Goal1 is called.

The second clause of Goal1 first chooses a potentially selectable atom via the predicate Select. In the conditional, StatementMatchAtom and Resolve are used to test whether there is a statement in the knowledge base, the head of which unifies with this atom. If there is, the body of the statement is added to the conjunction of potentially selectable atoms using AndWithEmpty. If there is not, the atom is conjoined with the unselectable atoms in the fifth argument.

The predicate Select, which chooses an atom from the body of a goal, is shown in Figure 4.4. This is a simple implementation of Select, which always chooses the leftmost atom in the goal. Because the representation of a conjunction is in general an arbitrary tree structure, the predicate AndWithEmpty is used to rebuild the tree after removal of the selected atom, so avoiding the unnecessary construction of conjunctions of empty formulas.

```
PREDICATE Goal :
                % Knowledge base.
   Program
 * Formula
                % The body W of a goal.
                % The body of a goal in the SLD-tree for <-W.
 * Formula.
Goal(prog, goal, a_goal) <-</pre>
   FormulaMaxVarIndex([goal], vi) &
   EmptyTermSubst(empty_subst) &
   EmptyFormula(empty_goal) &
   Goal1(prog, goal, vi, empty_subst, empty_goal, a_goal).
PREDICATE Goal1 :
   Program * Formula * Integer * TermSubst * Formula
 * Formula.
Goal1(_, empty, _, subst, unsel, a_goal) <-</pre>
   EmptyFormula(empty) &
   ApplySubstToFormula(unsel, subst, a_goal).
Goal1(prog, goal, vi, subst, unsel, a_goal) <-</pre>
   Select(goal, atom, left, right) &
   IF SOME [stat, new_vi, new_subst, body]
      StatementMatchAtom(prog, _, atom, stat) &
      Resolve(atom, stat, vi, new_vi, subst, new_subst, body)
   THEN
      AndWithEmpty(left, body, goal1) &
      AndWithEmpty(goal1, right, new_goal) &
      Goal1(prog, new_goal, new_vi, new_subst, unsel, a_goal)
   ELSE
      AndWithEmpty(unsel, atom, new_unsel) &
      AndWithEmpty(left, right, new_goal) &
      Goal1(prog, new_goal, vi, subst, new_unsel, a_goal).
```

Figure 2.8: Finding a goal in an SLD-tree

```
PREDICATE Select :
   Formula
                % A conjunction of atoms.
 * Formula
                % An atom selected from the conjunction.
                % The conjunction to the left of the
 * Formula
                % selected atom.
 * Formula.
                % The conjunction to the right of the
                % selected atom.
Select(atom, atom, left, right) <-
   Atom(atom) &
   EmptyFormula(left) &
   EmptyFormula(right).
Select(goal, atom, left, right) <-</pre>
   And(left1, right1, goal) &
   Select(left1, atom, left, right2) &
   AndWithEmpty(right2, right1, right).
```

Figure 2.9: Selecting an atom from the body of a goal

The translation of the set of goals returned by Goal into a list of transactions is performed by RawInsertionTransactions. The structure of the definition of RawInsertionTransactions in Figure 2.10 is typical of Gödel predicates that process the elements of a set. The one-solution commit around the conditional is needed to manage the unwanted nondeterminism associated with selecting an element from a set. Notice also that once an element has been selected, it must be explicitly removed from the set by a set difference operation. RawInsertionTransactions processes each representation of the body of a goal via a call to GoalTransaction, which unpacks the conjunction of atoms to form a list of assertions.

Now I come to the part of the program that filters the candidate transactions, by removing those that cause the updated program to violate an integrity constraint. This is the predicate Filter, shown in Figure 2.11. Each transaction is checked in turn, by first applying the transaction to the knowledge base to obtain an updated knowledge base, and then evaluating

```
PREDICATE RawInsertionTransactions :
   Set(Formula)
                         % Some goals.
 * List(List(Action)). % Transactions derived
                         % from these goals.
RawInsertionTransactions(goals, transs) <-</pre>
   { IF SOME [goal] goal In goals
     THEN
        GoalTransaction(goal, trans, []) &
        transs = [trans|rest] &
        RawInsertionTransactions(goals\{goal\, rest)
     ELSE
        transs = []
   }.
PREDICATE GoalTransaction :
   Formula
                         % A conjunction of atoms.
 * List(Action)
                         % The list of assertions of
                         % atoms in the conjunction.
 * List(Action).
                         % The tail of the list.
GoalTransaction(atom, [A(fact)|actions_t], actions_t) <-</pre>
   Atom(atom) &
   EmptyFormula(empty) &
   IsImpliedBy(atom, empty, fact).
GoalTransaction(conjunction, actions, actions_t) <-</pre>
   And(left, right, conjunction) &
   GoalTransaction(left, actions, actions_1) &
   GoalTransaction(right, actions_1, actions_t).
```

Figure 2.10: Generating transactions from goals

the goal supplied for integrity constraint checking in the updated knowledge base. If the goal succeeds, the integrity constraints are satisfied and the transaction is retained, otherwise it is discarded.

The predicate ApplyTransaction applies a transaction to the knowledge base, using InsertStatement to insert the clause for each assertion. The analogue for retractions is omitted for brevity, but is easily defined using DeleteStatement. Because the update is declarative, the original version of the knowledge base is unchanged by the application of the transaction, so there is no need to explicitly undo the effect of each transaction before testing the next one. This would not be the case for an analogous Prolog program using the non-ground representation and assert/1 and retract/1, which modify the knowledge base destructively. In fact, reversing the effect of a transaction in this case can be difficult to do correctly, and may require an additional mechanism such as the clause tagging scheme of SICStus Prolog [76].

The integrity constraints are checked by using the Succeed interpreter to run the appropriate computation. Note that if a call to Succeed fails, then the object-level computation either failed or floundered. It would be straightforward to modify the Filter procedure to check for floundering in the integrity constraint computation, by using Compute instead of Succeed if necessary. However, the example insertion given in this section does not cause the integrity constraint computation to flounder.

A predicate for testing the assimilator, TryInsert, is shown in Figure 2.12. This predicate takes three input arguments, all of type String: the name of the main module of the knowledge base, a representation in concrete syntax of an atom to be inserted, and a representation in concrete syntax of the body of a goal that succeeds if and only if the integrity constraints are satisfied. It returns a list of transactions that achieve the insertion in its fourth argument, with the statements in the actions translated from the abstract type Formula into concrete syntax, so that they are visible to the user of the program. The use of concrete syntax representations in the user interface is a standard meta-programming technique in Gödel, and is an important function of the predicates StringToProgramFormula and ProgramFormulaToString.

TryInsert reads in the knowledge base using ProgramCompile from ProgramsIO, and parses the strings in the second and third arguments

```
PREDICATE Filter :
   Program
                        % Knowledge base P.
 * Formula
                        % Body of a goal for checking
                        % the integrity constraints.
 * List(List(Action))
                        % Some transactions.
 * List(List(Action)). % The transactions t such
                        % that t(P) satisfies C.
Filter(_, _, [], []).
Filter(prog, ics, [trans|transs], filtered) <-</pre>
   ApplyTransaction(prog, trans, new_prog) &
   ( IF Succeed(new_prog, ics, _) THEN
        filtered = [trans|filtered1]
     ELSE
        filtered = filtered1
   ) &
   Filter(prog, ics, transs, filtered1).
PREDICATE ApplyTransaction :
   Program
                        % Knowledge base P.
 * List(Action)
                        % A transaction t.
 * Program.
                        % The knowledge base t(P).
ApplyTransaction(prog, [], prog).
ApplyTransaction(prog, [A(fact)|actions], new_prog) <-
   MainModuleInProgram(prog, mod) &
   InsertStatement(prog, mod, fact, prog1) &
   ApplyTransaction(prog1, actions, new_prog).
```

Figure 2.11: Filtering transactions

```
PREDICATE TryInsert :
   String
                          % The name of the knowledge base.
 * String
                          % The string representation of
                          % an atom to insert.
                          % The string representation of
 * String
                          % the body of a goal for integrity
                          % constraint checking.
* List(List(StrAction)). % A list of transactions that
                          % achieve the insertion, in visible
                          % syntax.
TryInsert(prog_str, atom_str, ic_str, trans_strs) <-</pre>
   ProgramCompile(prog_str, prog) &
  MainModuleInProgram(prog, mod) &
   StringToProgramFormula(prog, mod, atom_str, [atom]) &
   StringToProgramFormula(prog, mod, ic_str, [ic]) &
   InsertionTransactions(prog, atom, ic, transs) &
   TransactionStrings(transs, prog, mod, trans_strs).
```

Figure 2.12: Testing the assimilator

to obtain their representations as terms of type Formula. It then computes the transactions, and translates them into a visible representation using TransactionStrings, defined in Figure 2.13. This predicate uses ProgramFormulaToString to re-represent terms of type Formula as strings.

An example knowledge base for the assimilator is shown in Figure 2.14, and Figure 2.15 contains an integrity constraint theory for this knowledge base. Together, these form the module Family. The integrity constraint theory provides a proposition Consistent that is true if the knowledge base satisfies all the integrity constraints. Using the Family module as the object program, the meta-level goal

attempts to insert the atom Grandson (Isaac, Terach) into the knowledge

```
BASE StrAction.
FUNCTION Assert, Retract : String -> StrAction.
PREDICATE TransactionStrings :
   List(List(Action))
                          % List of transactions.
 * Program
                          % Knowledge base.
 * String
                          % Main module name.
 * List(List(StrAction)). % The transactions in
                          % concrete syntax.
TransactionStrings([], _, _, []).
TransactionStrings([trans|rest], prog, mod,
      [trans_str|rest_strs]) <-</pre>
   TransactionString(trans, prog, mod, trans_str) &
   TransactionStrings(rest, prog, mod, rest_strs).
PREDICATE TransactionString :
   List(Action) * Program * String * List(StrAction).
TransactionString([], _, _, []).
TransactionString([A(stat)|rest], prog, mod,
      [Assert(stat_str)|rest_strs]) <-
   ProgramFormulaToString(prog, mod, stat, stat_str) &
   TransactionString(rest, prog, mod, rest_strs).
```

Figure 2.13: Transforming transactions to visible syntax

base. The assimilator finds two transactions that achieve the insertion, and gives the following answer.

```
s = [[Assert("Father(Terach, Abraham) <- ")],
        [Assert("Father(Terach, Sarah) <- ")]
]</pre>
```

To demonstrate the role of the integrity constraint theory in filtering transactions, integrity constraint checking can be short-circuited by replacing the proposition Consistent by the proposition True, which succeeds immediately. Thus, the goal

```
<- TryInsert("Family", "Grandson(Isaac, Terach)", "True", s).
gives the answer

s = [[Assert("Father(Terach,Abraham) <- ")],
       [Assert("Father(Terach,Sarah) <- ")],
       [Assert("Mother(Terach,Abraham) <- ")],
       [Assert("Mother(Terach,Sarah) <- ")]
]</pre>
```

Thus two candidate transactions have been eliminated by the Filter predicate. These violate the integrity constraint that all mothers are female.

This example shows an application of moderate sophistication using the ground representation in Gödel. Of course, a program that performs approximately⁴ the same function can be constructed using the non-ground representation in Prolog, but it will not be declarative. The Gödel assimilator program is comparable to such a Prolog program in size and economy of expression. In addition, the Gödel program is considerably easier to understand than its Prolog analogue, because the Prolog program must take steps to avoid the unwanted instantiation of variables in the object program representation, and must deal with the destructive nature of assert/1 and retract/1 in checking the integrity constraints. It can be argued that the Gödel approach to meta-programming scales much better that the Prolog approach does; the more complex the application, the more the non-ground

⁴Not exactly, because the transactions will be in the non-ground representation, with all the attendant disadvantages.

representation requires the use of non-declarative features, and programs become correspondingly more difficult to read. Thus the assimilator example demonstrates that one aim in the design of Gödel has been met: that of making the ground representation easy to use for routine programming tasks.

```
MODULE Family.
BASE Person.
CONSTANT
   Terach, Abraham, Nachor, Haran, Isaac, Lot,
   Sarah, Milcah, Yiscah: Person.
PREDICATE
   Father
                : Person * Person;
                : Person * Person;
   Mother
   Male
                : Person;
   Female
                : Person;
   Parent
                : Person * Person;
                : Person * Person;
   Son
   Grandson
                : Person * Person.
Father(Terach, Nachor).
                            Father (Terach, Haran).
Father(Abraham, Isaac).
                            Father (Haran, Lot).
Father(Haran, Milcah).
                            Father (Haran, Yiscah).
Mother(Sarah, Isaac).
Male (Terach).
                            Male (Abraham).
Male(Nachor).
                            Male (Haran).
Male(Isaac).
                            Male(Lot).
Female(Sarah).
                            Female (Milcah).
Female(Yiscah).
Parent(x, y) \leftarrow Father(x, y).
Parent(x, y) \leftarrow Mother(x, y).
Son(x, y) \leftarrow Parent(y, x) & Male(x).
Grandson(x, y) \leftarrow Son(x, z) \& Parent(y, z).
```

Figure 2.14: A family tree knowledge base

```
PROPOSITION
   Consistent,
   IC1, IC2, IC3, IC4, IC5.
Consistent <- IC1 & IC2 & IC3 & IC4 & IC5.
% No-one is both male and female.
IC1 <- ALL [x] ~(Male(x) & Female(x)).
% All mothers are female.
IC2 <- ALL [x] (SOME [y] Mother(x, y) -> Female(x)).
% All fathers are male.
IC3 <- ALL [x] (SOME [y] Father(x, y) \rightarrow Male(x)).
% No-one has two (distinct) fathers.
IC4 <- ALL [x, y] (</pre>
   SOME [z] (Father(x, z) & Father(y, z)) ->
      x = y).
% No-one is a parent of his/her parent.
IC5 <- ALL [x, y] (Parent(x, y) -> "Parent(y, x)).
```

Figure 2.15: Some integrity constraints for the family tree knowledge base

Chapter 3

An Implementation of Gödel

The implementation of Gödel is best described by considering the ground representation first. Compilers are archetypal meta-programs, so the Gödel compiler is naturally constructed using Gödel's meta-programming library. The greater part of the discussion of the implementation therefore concerns the ground representation and, in particular, the structure of the term of type Program that represents a Gödel object program. An earlier version of this work was reported in [14].

A major aim of the implementation of the ground representation in Gödel is that it should permit effective meta-manipulation of meta-programs themselves. For example, it should be possible to perform effective partial evaluation of Gödel meta-programs. For this reason, much of the implementation of the Gödel meta-predicates, particularly those exported by the Syntax module, has to be in pure Gödel code, so that the logic can be made available to meta-programs. This requirement to make the logic explicit limits the possibility of achieving efficiency by programming at a lower level.

3.1 Representing Gödel Syntax

This section explains how the symbol names, types, terms and formulas of a Gödel object program are encoded as compound terms in a Gödel metaprogram.

A term in the representation can be understood as standing for a particular syntactic expression at the object level, independently of any particular object program, and the Syntax module is concerned with manipulation

of object-level expressions without reference to any context of language or module structure.

A Gödel term only has a type in the context of the language of a particular program or module within a program; it may have different types in different contexts, and indeed will not be well-formed with respect to all languages. Similarly, its representation in concrete syntax (that is, the syntax that a programmer would use to write down this term) may also depend on the context of a language to provide fixity and associativity information for any operators it contains. This property is captured in Gödel's meta-representation by the division of function between the Syntax and Programs modules. Thus, an object program representation is required to compute a type, or concrete syntax representation, for an object-level term, and this information is not encoded in the term representation. The predicates in the Syntax module are free to manipulate term representations regardless of their meaning or validity in any particular object program context.

The syntax representation is formed from constants and functions declared in the local part of the the Syntax module.

3.1.1 Representing Symbol Names

The abstract type Name is provided for the representation of symbol names. Elements of the type are the basic units from which the representation of syntax is constructed.

Because of Gödel's foundation in polymorphic many-sorted logic (Appendix A), a Gödel meta-program has (with the exception of those belonging to certain special types, most notably the integers) only a finite number of constant symbols and function symbols with which to represent the symbols of the object program, and these are statically determined by the declarations in the meta-program when it is compiled. This is in contrast to Prolog, where new function symbols can be created dynamically at run-time as required. It is possible to imagine a strongly-typed programming language in which typed symbols can be created dynamically, perhaps with the aid of run-time type-checking, but interpretations of first-order theories are defined over a fixed alphabet, so assigning a declarative semantics to such a language may be problematic. The philosophy behind Gödel therefore precludes the use of the common naming convention in which $\lceil c \rceil = c'$ for some constants $c \in \mathcal{L}_O$ and $c' \in \mathcal{L}_M$.

The names of object-level symbols are represented in flattened form, which guarantees that the representation stands for a unique symbol within a syntactically correct object program. Unique names could be constructed in one of several different ways; in Gödel it is done by encoding the name of the module in which the symbol was declared, its syntactic category, and its arity, within the name. None of the operations provided by Syntax for the Type, Term and Formula data types depend on the internal structure or properties of individual names: terms of type Name are regarded as atomic objects within the Syntax module. Terms of type Name can only be constructed in the presence of an object program representation, which emphasises that the creation of flat names is context dependent.

It will be seen later that for representing Gödel programs it is convenient to have the four components of Gödel flat names explicit in the Name term so that they can be easily extracted. Gödel flat names are therefore represented by the function Name, declared in the local part of Syntax.

```
FUNCTION Name :
String  % Name of module where symbol is declared.

* String  % Declared name of symbol.

* Category  % Symbol's category.

* Integer  % Symbol's arity.

-> Name.
```

The base type Category gives the category of the symbol and contains a constant for each category.

```
CONSTANT

Base, Constructor, Constant, Function, Proposition,

Predicate: Category.
```

For example, the meta-level term Name("M", "F", Function, 2) represents the flat name of a function symbol with declared name F, declared in module M, with arity 2.

3.1.2 Representing Types

Object-level types are represented by ground terms in the abstract type Type, built from the two functions BType and Type.

The BType function represents a base type. The Type function represents a compound type, with a representation of the name of a type constructor, and a list of representations of its arguments. The BType function represents a base type.

The strategy of using a different top-level function symbol to differentiate the base type from the compound type, rather than representing the base type using the Type function with an empty argument list, follows a principle of good style in logic programming (see [63]), which asserts that the top-level function symbol of a term should express as much information as possible about the term. This makes it easy and efficient for programs to make case-based decisions concerning the treatment of the term. This stylistic principle is sometimes neglected in Prolog programs, but Gödel's strong typing encourages its use, as demonstrated in the Flatten example of Figure 2.2.

Type parameters are represented by ground terms using the function Par.

```
FUNCTION Par :
String % String representing the parameter name.
* Integer % Parameter index.
-> Type.
```

The normal scope limitations on both variables and type parameters

mean that they are adequately identified by the string of characters that names them in the concrete syntax of the object program. However, as will be seen, keeping track of variable names is the principal obligation of metaprograms using the ground representation, and many common operations on object programs require creation of new, distinct variables in large numbers, in addition to those explicitly named in the object program. For example, new type parameters must be created when types are standardised apart, in the process of finding the type of a term (see the definition of a polymorphic many-sorted term in Appendix A). The integer index argument is provided so that new type parameters can be created easily by simply incrementing the index.

It shall become apparent that it is very useful to have available a set of type parameter (and, more importantly, variable) representations that can easily and efficiently be determined to differ only in the value of the index, without the need to perform string comparisons. To this end, a second function Par of arity 1 is provided, where for example Par(1) is defined to represent the object-level parameter p_1.

It will be observed that there are now three distinct representations for parameters such as p_1, being Par("p",1), Par("p_1", 0) and Par(1), although use of the second representation is discouraged, and the Gödel parser produces the third.

```
FUNCTION Par : % A parameter, the name of which is "p"

Integer % Parameter index
-> Type.
```

3.1.3 Representing Terms, Variables and Atoms

In a similar way to types, object-level constants and compound terms are represented by meta-level terms of type Term, using functions CTerm and Term respectively.

A variable is a term, so object-level variables are directly represented within the type Term. Of course, since this is a ground representation, object-level variables are represented by ground terms at the meta-level. Like the representation of type parameters, these terms also have an index argument, which facilitates the creation of new variables, for example when required for standardisation apart. Again, as with type parameters, there is a unary version of the Var function, so that Var(1) is defined to represent the object-level variable v_1.

```
FUNCTION Var :
String % String representing the variable name.
* Integer % Variable index.
-> Term.

FUNCTION Var : % A variable, the name of which is "v"
Integer % Variable index
-> Term.
```

There is no special representation for underscore variables, as the parser replaces them by unique variables with explicit existential quantifiers.

Now consider the representation of literals, such as strings and numbers. Although it would be possible to use the function CTerm, together with an appropriate Name term, to represent these terms, it is beneficial to distinguish them by special meta-level functions, thus avoiding unnecessary inter-conversions between the values of the literals and their representation in concrete syntax.

```
FUNCTION Int : % Represents an integer
Integer % given by this meta-level integer.

-> Term.

FUNCTION Str : % Represents a string
String % given by this meta-level string.

-> Term.

FUNCTION Flo : % Represents a floating point number
Float % given by this meta-level float.

-> Term.
```

Finally, intensional sets require an explicit representation. Notice that this is the only case in which a term contains an embedded formula.

```
FUNCTION SuchThat: % Represents the set of
Term % instances of this term
* Formula % such that this formula is true
-> Term.
```

3.1.4 Representing Formulas

Object-level atoms are represented by functions of PAtom and Atom of type Formula.

Representations of formulas are terms of type Formula. They are built from terms representing atoms, the constant Empty which represents the empty formula, the unary function $\tilde{}$ which represents negation, and functions &, $\backslash/$, \rightarrow , <-, and <->, representing the binary connectives conjunction, disjunction, left implication, right implication, and equivalence, respectively. The functions that represent binary connectives all have type

Formula * Formula -> Formula

The representation of universally and existentially quantified formulas uses the functions All and Some.

The Gödel commit is represented by the Commit function. Note that the two simple forms of the Gödel commit, the bar commit and one-solution operator, can both be viewed as syntactic sugar for special cases of the full commit (see Section 2.1.4). The full commit has a scope which encloses a formula, and an integer label. During parsing, the simple forms of the commit are expanded to full commits with suitably allocated labels.

```
FUNCTION Commit:
Integer % Commit label.
* Formula % Formula in the scope of this commit.
-> Formula.
```

Lastly, there are four functions to represent the four different flavours of Gödel conditionals. The functions ISTE and ITE represent the full conditional, in quantified and unquantified form, respectively, and similar functions IST and IT represent the quantified and unquantified form of the conditional without the ELSE part.

```
FUNCTION ISTE: % IF-SOME-THEN-ELSE.
   List(Term)
                % List of quantified variables.
 * Formula
                % Condition.
                % Then part.
 * Formula
 * Formula
                % Else part.
-> Formula.
FUNCTION ITE: % IF-THEN-ELSE
   Formula
                % Condition
 * Formula
                % Then part
 * Formula
                % Else part
-> Formula.
```

```
FUNCTION IST : % IF-SOME-THEN
  List(Term) % List of quantified variables
 * Formula % Condition
 * Formula % Then part
-> Formula.

FUNCTION IT : % IF-THEN
  Formula % Condition
 * Formula % Then part
-> Formula .
```

As an illustration, consider the representation under this scheme of the clause

The meta-level representation is syntactically much larger than the concrete syntax of the formula it represents. Let us call the representation of terms by heap structures at the lowest level within the implementation the machine level representation. The machine-level representation of the meta-level representation of an object-level term is correspondingly larger than the machine level representation of the object-level term. If a term with n arguments requires n+1 machine words plus the size of its arguments for its machine-level representation, as in a typical WAM implementation, the meta-level representation of the same term in Gödel will require 2n+9 machine words plus the size of the representation of its arguments.

The bulky form of the meta-representation in Gödel is however forced upon us by the interaction of two important design decisions:

- 1. strong typing prevents the creation of explicit new functions in the meta-program, for use in the representation;
- 2. the logic that implements the ground representation must be explicit, so that it is available to other meta-programs such as partial evaluators.

The combination of these rules effectively constrains the representation to be constructed using the available generic data structures, strings and lists.

The representation that has been described is fully general in terms of permitting the representation of partially known structures. That is, metavariables can appear in place of the Name terms that represent function or predicate symbols, or in place of some terms representing arguments, or in place of some or all of the argument list. For example, it is straightforward to use the facilities of the Syntax module to create a partial representation such as Term(y, [Var("w")|z]), where y and z are meta-variables. This represents some compound object-level term which has the variable w as its first argument, but about which nothing else is known. Although control declarations could be used to add modes to the predicates in Syntax and prevent the creation of partial representations, this restriction has been avoided to give the programmer maximum flexibility. Partial representations are occasionally useful, for example in program synthesis, where they can be used to express constraints upon parts of a program under construction. as described in [20]. Programmers working with the traditional Prolog style of meta-programming cannot represent terms with variable function symbols or arities. Some Prolog systems do permit terms with variable function symbols, but of course such a feature is outside first-order logic.

3.2 Representing Gödel Programs

This section describes the structure of terms of type Program. The Program type is exported by the Programs module. Each Program term represents an entire Gödel object program, including its module structure, language declarations, control declarations and statements.

Just as flat names are used to build the representation of types, terms and formulas in the Syntax module, so the entire object program is also represented in flat form; thus every symbol has a unique name and a single declaration, and all overloading of declared names must have been resolved prior to creation of the representation.

At the top level, every term of type Program has the function Program, which takes four arguments.

```
FUNCTION Program :
String  % Name of the main module.

* Table(ModuleDefinition) % Module structure.

* Language  % Flat language of the program.

* Table(ModuleCode)  % Statements and DELAYs.

-> Program.
```

The second, third and fourth arguments represent the three main components of a Gödel program. These are the module structure, the flat language, and the program statements and DELAY declarations, expressed in this flat language. There follows a brief discussion of the design of symbol tables, and then these three structures will be described in turn.

3.2.1 Symbol Tables

A dictionary structure is required to store the declarations and statements connected with each object-level symbol, so that they are together and readily accessible in one term. For this, the Gödel system module Tables is used. This module exports a polymorphic abstract type Table(a), which implements association lists that pair keys of type String with terms of any

type. A table can be viewed as an ordered collection of such pairs, where the ordering is given by the lexical ordering of the keys.

One of the most useful predicates in the tables module is AmendTable. This predicate performs a generalised table update, in which a new value is associated with a given key, and the value previously associated with the key is returned. If the key was not previously present in the table, a default value, provided as an additional argument to AmendTable, is returned instead.

This arrangement solves a perennial problem in the design of interfaces to symbol tables. It is often difficult to specify how the addition of a key to a table should behave if the key is already present, or how the alteration of the value associated with a key should behave if the key is not present, in a way that meets all the required uses of the table. The design of AmendTable allows adaptation at the point of call to all possible specifications of simpler interfaces.

```
PREDICATE AmendTable :
  Table(a)
            % A table.
* String
            % A key.
            % A (new) value to be associated with the key.
            % A value (a default "old value").
* Table(a) % The table updated so that the new value is
            % associated with the key. (If the key is already
            % present in the table, the associated value will
            % be updated; otherwise the node with this key
            % and value will be inserted into the table.)
            % The old value associated with the key if it was
* a.
            % already present in the table, otherwise the
            % value in the fourth argument.
DELAY AmendTable(x,y,_,_,_) UNTIL NONVAR(x) & GROUND(y).
```

The AmendTable predicate has two major advantages. Firstly, it entirely avoids the need to perform an additional lookup, to determine whether or not a key is already present before updating; this is otherwise often necessary,

and the saving may be significant especially if the lookup takes worse than constant time. Secondly, by providing a suitable default value, code can be simplified considerably because it becomes unnecessary to consider the key addition and key update cases separately.

In the absence of support for a logical data structure that allows constant time lookup (this topic will be discussed at length later on), the most efficient way to implement association lists is to use a tree data structure of some kind, typically a binary tree, which allows logarithmic access times.

Meta-programs are often concerned with the automatic generation of programs. It is folklore among compiler constructors that mechanically generated programs are much more likely than hand written programs to declare large numbers of symbols in a lexical order. Such programs can cause catastrophic degradation of the performance of compilers based on simple binary tree dictionaries because the ordered insertion creates linear structures, resulting in linear access times. It therefore seems particularly important in this case to employ a balancing algorithm.

It can also be observed that in many meta-programming applications, lookup operations are performed very much more frequently than update operations on the Program structure. For example, the processes of type-checking an object-level formula requires multiple lookups, one for each symbol in the formula, whereas the addition of a statement to the object program involves a single update of the program structure, but requires that the statement be type-checked before it is added. Thus the additional cost of rebalancing after insertions and deletions should be acceptable.

The Gödel Tables module is implemented using AVL trees, named after their inventors, Adelson-Velskii and Landis. AVL trees guarantee logarithmic time for lookup, insertion and deletion, and are quite easy to implement in Gödel; algorithms for insertion and deletion can be found in [88], and implementations in Prolog can be found in [17, 81]. Alternative data structures are n-k-trees and red-black-trees, which provide similar performance, but have slightly more complex re-balancing operations.

3.2.2 Representing the Module Structure

The module structure of a Gödel program is a directed acyclic graph with a single root at the main module. The set of modules making up the program form the nodes of the graph, and the edges are formed by import declara-

tions. The edges can be divided into two classes, depending on whether the import declaration is in the local part or the export part of the importing module.

It might be possible to use an isomorphic graph structure directly to represent the module structure. However, a more convenient way to represent a graph is to use a dictionary that associates each node with a list of its children, in this case associating the name of each module with the names of the modules that it imports. This structure has less duplication, is easier to update, is more appropriate to the questions usually asked of it, and has the useful property that the representation of each node is independent of the content of its child nodes.

The module structure is represented by a dictionary of type Table(ModuleDefinition) where the keys are module names. There is one function ModDef of type ModuleDefinition.

The type OModuleKind contains constants NormalKind, ClosedKind and ModuleKind, that indicate whether the named module has an ordinary export and local part, is closed, or has a MODULE keyword in its local part and therefore no export part, respectively.

```
CONSTANT NormalKind, ClosedKind, ModuleKind: OModuleKind.
```

For example, if looking up the module name "Hats" in the module structure obtains the term ModDef(NormalKind, [], ["Lists"]) then Hats is an ordinary module with an export part and a local part that has the declaration IMPORT Lists in its local part.

3.2.3 Representing the Program Language

The structure of the term used to represent the flat language of the object program has an effect on the efficiency of the system, and it has to meet several requirements.

- 1. It must contain a representation of the declaration of every symbol in the program.
- 2. It must be able to act as the symbol table for the parser, and so deal with overloading.
- 3. It must allow fast access to the declaration of any symbol given its flat name.
- 4. It must be possible to determine the flat language of any one of the object program's modules in a straightforward way.

Requirement 2 means that, when the Gödel parser parses a symbol name in the program source, it must be able to recover from the language term a list of all the symbols in scope with that declared name. That is, in addition to being able to look up a flat name in the language term and find its declaration, it must also be possible to look up a declared name and find a list of the declarations that might apply (the list of applicable declarations is narrowed, hopefully to one, during type checking).

Rapid access to declarations from flat names is important since the declaration of every symbol in an expression is examined during the process of validating an expression with respect to a language, and such validations are performed frequently in routine meta-programming.

Requirement 4 arises because it is often necessary to validate an expression with respect to one of the various views of the program language, such as the export language of a specific module. For example, this might be necessary to determine whether a specific declaration can be legally inserted in that module part. It should therefore be possible to retrieve the subset of the flat language of the program that is accessible within either part of any one of the programs modules, and do so at reasonable cost.

It is also important that the language representation is as compact as it can be while meeting the above requirements.

A naïve approach to representing the language structure might be as a simple dictionary linking flat names with their declarations. For example, the string "Hats.Topper.Constant" might represent the flat name of the constant Topper in the module Hats, and could be used as a key to index the

declaration of this constant. However, this scheme cannot easily be made to meet requirements 2 and 4.

To solve this problem, use is made of the internal structure of the Name type from Syntax. The components of the name can be used to reduce the number of comparisons made during each lookup. The idea is to use nested dictionaries, where the outer dictionary contains an entry for each module, which is in turn a dictionary. The module dictionary links each declared name in the module with the declarations of all the symbols declared in the module with that declared name.

The Syntax module is unaware of the concepts of module, category and arity, even though they appear in the Name structure. As far as the representation of types, terms and formulas is concerned, any name can appear anywhere. However, when a declaration for a name is inserted into the representation of a program, the Programs module ensures that the components of the name are consistent with its declaration. It can therefore be safely assumed that any name that has the string "Hats" as its module component is declared in module "Hats", or has no declaration at all.

The flat language of the object program, and the other flat languages the object program defines are represented by subterms of type Language, formed from the function Language, which gives the term representing the program language a simple type.

```
FUNCTION Language :
Table(ModuleDescriptor)
-> Language.
```

The key for the dictionary Table(ModuleDescriptor) is the name of a module, and the entries are called *module descriptors*. The base type ModuleDescriptor contains one function, Module.

```
FUNCTION Module :

Accessibility  % Which symbols are accessible.

* CategoryTable  % Dictionary of symbols.

-> ModuleDescriptor.
```

The argument of type CategoryTable contains representations of the declarations of all symbols declared in the module. Its structure will be described shortly.

The Accessibility argument of Module is provided to satisfy requirement 4. It is a flag that can take two values.

```
BASE Accessibility.

CONSTANT Exported, Hidden: Accessibility.
```

The accessibility flag determines the role that this module plays in the language as a whole, by indicating which of the symbols declared in the module are actually present in the language. If the accessibility of the module is Hidden, all the symbols it declares are present. If its accessibility is Exported, only the symbols declared in the export part of the module are present in the language. That is, an accessibility of Hidden means that even the hidden symbols in the module (those declared in the local part) are visible, whereas Exported means that only the exported symbols are visible.

In the representation of the flat language of the program, all the module descriptors have an accessibility of Hidden. In the representation of the flat language of some component module (the language in which the statements of the module are written) the module descriptor for the module itself has accessibility Hidden; the only other module descriptors present are those for the modules it imports, and they all have their accessibilities set to Exported.

The CategoryTable dictionary is split into two parts to make a small optimisation. Since every symbol's category is part of its flat name, the category can be used to reduce the overhead of locating its entry in the language representation. However, the Gödel parser has also to be supported by satisfying requirement 2 above, and it is of course important for usability of the Gödel system that the parser is as fast as possible. When the parser meets a symbol in its input, it cannot determine the category of the symbol for certain in advance, but it can tell from the context whether the symbol is part of a type or part of a formula. The symbols can therefore be divided into two classes without affecting the efficiency of the parser. If all six categories were separated, the parser would have to search four dictionaries for every

symbol it encountered while parsing a statement or goal.

The two tables associate the declared name of a symbol with the list of declarations in this module for symbols with this declared name in the appropriate group of categories. Linear search is adequate to find the declaration that is actually required. Overloading is uncommon so there is usually only one entry in the list, and very rarely more than two. The representation used for declarations encodes the category, and where appropriate the arity, of the symbol so that the correct declaration can be uniquely identified from the flat name.

Every symbol is described by a structure of type SymbolDescriptor, which has the function Symbol with two arguments.

FUNCTION Symbol:

Accessibility

- * Declaration
- -> SymbolDescriptor.

The Accessibility argument is the same as the flag that gives the accessibility for the module descriptor, but here it indicates which part of the module declares the symbol. An accessibility of Exported indicates a symbol declared in the export part of the module; such symbols are always visible regardless of the accessibility of the module. An accessibility of Hidden indicates a symbol declared in the local part, which is only visible if the accessibility of the module is also Hidden.

Finally, the Declaration type has constants to represent the declarations of base and proposition symbols, and functions to represent the declarations of constructor, constant, function and predicate symbols together with their attributes, such as type and arity information.

```
CONSTANT
   BaseDecl,
   PropositionDecl : Declaration.
FUNCTION ConstructorDecl :
                         % Arity.
   Integer
-> Declaration.
FUNCTION ConstantDecl :
   Туре
                         % Туре.
-> Declaration.
FUNCTION FunctionDecl :
                         % Arity.
   Integer
 * FunctionInd
                         % Indicator.
 * List(Type)
                         % Domain type.
 * Type
                         % Range type.
-> Declaration.
FUNCTION PredicateDecl:
   Integer
                         % Arity.
 * PredicateInd
                         % Indicator.
 * List(Type)
                         % Domain type.
-> Declaration.
```

The base types FunctionInd and PredicateInd and the constants and functions that define them are declared in the export part of Syntax and so are public. These represent the indicator component of functions and predicates declared in the object program.

```
CONSTANT NoPredInd: PredicateInd.

CONSTANT ZPZ, ZP, PZ: PredicateInd.

CONSTANT NoFunctInd: FunctionInd.

FUNCTION XFX, XFY, YFX, XF, FX, YF, FY:
   Integer
-> FunctionInd.
```

3.2.4 Representing the Program Code

The third and final element of the representation is the fourth argument of the Program function, which has type Table (ModuleCode) and is the dictionary containing representations of the object program's statements and control declarations. By looking up the flat name of a predicate symbol in this dictionary, all the statements in the definition of the predicate and all the DELAY declarations for the predicate can be found. None of the predicates exported by the Programs module provides the capability of returning the statements of any predicate definition in a closed module, and any DELAY declarations made in the local part of a closed module are similarly protected from access by meta-programs. In practice these statements and declarations do not need an explicit representation, and only the DELAY declarations that appear in the export parts of the closed modules are actually present in the entries for these modules in the code dictionary.

It is convenient to organise the code dictionary analogously to the language dictionary, using two nested Table terms to exploit the flat name structure. The outer structure, with type Table (ModuleCode), is keyed on the module name. The ModuleCode type contains the representation of the Code function.

```
FUNCTION Code :
    Integer
  * Table(List(PredicateDefinition))
-> ModuleCode.
```

The Integer argument here is used in the implementation of Succeed and the predicates similar to it, called *reflective interpreters*, that simulate Gödel computations for representations of object-level goals with respect to representations of object programs. The purpose of this argument is explained in Section 3.4.

The code portion of each module is represented in the second argument, a term of type Table(List(PredicateDefinition)) which is indexed by the declared names of the predicates declared in the module. Because of overloading, there can be more than one such predicate for each declared name, but these can be distinguished by their arities. Hence there is a list of terms of type PredicateDefinition for each declared name, with the function PredDef at the top level.

It does not make sense for a predicate to have DELAY declarations in both the export part and the local part of the module that declares it, so one of the lists of representations of DELAY declarations is normally empty.

The base type Delay is not exported by Programs, so is not accessible to client modules. Instead DELAY declarations are represented externally in two parts: the head atom, represented by a term of type Formula, and the condition, represented by a term of type Condition. The Delay function joins these components together.

```
FUNCTION Delay :
Formula
* Condition
-> Delay.
```

DELAY conditions are represented by terms built from the constant TrueCond, the functions Nonvar and Ground that are used to represent atomic conditions, and the functions And and Or that are used to build the representation of compound conditions.

```
CONSTANT TrueCond : Condition.

FUNCTION Nonvar, Ground :
   Term
-> Condition.

FUNCTION And, Or :
   Condition
* Condition
-> Condition.
```

3.3 Compiling Gödel Programs

The Gödel compiler consists of a front end and a code generator. Given a module M to compile, the front end parses the files M.exp and M.loc, chases down any modules imported in those files, and constructs the ground representation of a program with M as the top-level module, ensuring it is type-correct. This ground representation is a term of type Program, with a structure exactly as described in Section 3.2. The code generator takes this term and translates the statements and control declarations for module M into Prolog, creating a Prolog module called 'M'. In this section a brief description of the Prolog constructs generated by the translation is given, using SICStus Prolog [76] as the target language; an understanding of this process will aid an appreciation of the discussion of meta-interpreters that follows in Chapter 4. Another reason for describing the translation to Prolog is that the language definition leaves the procedural semantics of Gödel largely open, to be defined by the implementation. Thus the capabilities of the Bristol implementation of Gödel depend on the details of this translation into Prolog.

The translation of Gödel to Prolog must handle two significant differences between the two languages. Firstly, Gödel has a sophisticated procedural control mechanism in the form of a flexible computation rule, which

both enforces safe negation and gives the programmer control of the selection strategy through the use of DELAY declarations. Secondly, Gödel statements are allowed to have arbitrary formulas in the body, that may include existential and universal quantifiers, all the standard connectives of first-order predicate calculus, and additional high-level constructs such as conditionals.

Traditional Prolog provides only conjunction (,), disjunction (;) and a form of negation (\+) that does not enforce the safeness requirement of SLDNF-resolution. Prolog also provides non-declarative features such as cut, if-then-else (->;) and setof, which can be exploited in the implementation of Gödel. SICStus Prolog also provides facilities for varying the leftmost-first computation rule of Prolog, by suspending the execution of subgoals until specific conditions of variable instantiation are met.

This is far from an exhaustive account of the compilation of Gödel programs. For example, there is no discussion of the implementation of the Gödel commit, nor of Gödel's limited ability to handle arithmetic constraints, nor do I say anything about set unification. Nevertheless, the contents of this section should be sufficient to give an understanding of what is taking place when a Gödel program is compiled and executed.

3.3.1 Theoretical Aspects of Compilation

Neglecting for a moment those features of Gödel that lie outside conventional formulations of first order logic, such as conditionals and commits, Gödel statements are program statements (Appendix A), which have arbitrary first-order formulas in the body. A program, being a finite set of program statements, can be translated into a normal program using the well-known Lloyd-Topor transformations [49], and a normal program can be implemented in Prolog using Prolog's conjunction and negation, together with some control annotations to ensure that no non-ground negative literal is selected.

Some theoretical issues arise from the use of the Lloyd-Topor transformations in the implementation of a practical language such as Gödel, and these form the topic of this section. First, I introduce some notation.

The set of free variables in a formula F is denoted by $\operatorname{fvar}(F)$. The notation \overline{x} denotes the sequence or set (depending on context) of variables x_1, \ldots, x_n for some n. $\forall \overline{x}F$ (resp., $\exists \overline{x}F$) means that all the variables in \overline{x} are universally (resp., existentially) quantified in F. If \overline{x} is empty, then $\forall \overline{x}F$

and $\exists \overline{x}F$ are both equivalent to F. In addition, $\forall F$ (resp., $\exists F$) is short for $\forall \overline{x}F$ (resp., $\exists \overline{x}F$) where $\overline{x} = \text{fvar}(F)$.

To reduce the number of cases and simplify the notation, a formula F is often written $X \wedge W \wedge Y$ where W, X or Y can be any formula including True, False or another conjunction. The syntactic distinction between $(X \wedge W) \wedge Y$ and $X \wedge (W \wedge Y)$ is ignored. Thus a formula is sometimes regarded as a conjunction of a number of subformulas, with the empty conjunct denoted by True. Disjunction is treated similarly, although the empty disjunct is False.

A slightly modified form of the Lloyd-Topor transformations appears below, where V and W are formulas other than True, Σ is the set of predicates in the program, and $q \notin \Sigma$ is a new predicate. The statement $A \leftarrow X \land U \land Y$ where U is not a conjunction of literals, is selected from the program. Depending on the formula U and assuming $\overline{u} = \text{fvar}(U)$, one of the actions given in the table below is performed. It is assumed that all quantified variables are already renamed uniquely.

$$1 \quad A \leftarrow X \land \neg (V \land W) \land Y, \ \Sigma \qquad \mapsto \qquad \left\{ \begin{array}{l} A \leftarrow X \land \neg V \land Y \\ A \leftarrow X \land \neg W \land Y \end{array} \right\}, \ \Sigma \\
2 \quad A \leftarrow X \land \neg (V \lor W) \land Y, \ \Sigma \qquad \mapsto \qquad A \leftarrow X \land \neg V \land \neg W \land Y, \ \Sigma \\
3 \quad A \leftarrow X \land \neg (V \to W) \land Y, \ \Sigma \qquad \mapsto \qquad A \leftarrow X \land V \land \neg W \land Y, \ \Sigma \\
4 \quad A \leftarrow X \land \neg \neg V \land Y, \ \Sigma \qquad \mapsto \qquad A \leftarrow X \land V \land \neg Y, \ \Sigma \\
5 \quad A \leftarrow X \land \neg \exists \overline{x} V \land Y, \ \Sigma \qquad \mapsto \qquad \left\{ \begin{array}{l} A \leftarrow X \land \neg q(\overline{u}) \land Y \\ q(\overline{u}) \leftarrow V \end{array} \right\}, \ \Sigma \cup \{q\} \\
6 \quad A \leftarrow X \land \neg \forall \overline{x} V \land Y, \ \Sigma \qquad \mapsto \qquad \left\{ \begin{array}{l} A \leftarrow X \land \neg V \land Y, \ \Sigma \\ A \leftarrow X \land W \land Y \end{array} \right\}, \ \Sigma \\
7 \quad A \leftarrow X \land (V \lor W) \land Y, \ \Sigma \qquad \mapsto \qquad \left\{ \begin{array}{l} A \leftarrow X \land V \land Y \\ A \leftarrow X \land W \land Y \end{array} \right\}, \ \Sigma \\
8 \quad A \leftarrow X \land (V \to W) \land Y, \ \Sigma \qquad \mapsto \qquad \left\{ \begin{array}{l} A \leftarrow X \land \neg V \land Y \\ A \leftarrow X \land W \land Y \end{array} \right\}, \ \Sigma \\
9 \quad A \leftarrow X \land \exists \overline{x} V \land Y, \ \Sigma \qquad \mapsto \qquad A \leftarrow X \land V \land Y, \ \Sigma$$

10 $A \leftarrow X \wedge \forall \overline{x} V \wedge Y, \Sigma$

 $\mapsto \left\{ \begin{array}{l} A \leftarrow X \land \neg q(\overline{u}) \land Y \\ q(\overline{u}) \leftarrow \neg V \end{array} \right\}, \ \Sigma \cup \{q\}$

Given any initial program P with predicates Σ , a sequence of these transformations must terminate and the resulting program P' (called the normal form of P) is normal [50]. Moreover, if U is a logical consequence of comp(P') and all predicates in U are in Σ , then U is a logical consequence of comp(P), where comp(P) denotes the completion of program P (Appendix A).

As an illustration of the transformation to a normal program, consider the definition of the binary predicate ReverseList (Figure 3.1 where both arguments are lists and each is the reverse of the other. ReversePos is true when the *i*th element of the first list is the same as the element in the *i*th position from the end of the second list (where *i* is the third argument).

Length is a predicate provided by the Gödel system module Lists. The notation $a = \langle i = \langle b \text{ is syntactic sugar for the atom Interval}(a,i,b)$ and is used to generate values for i (or check that the values of i lie) between the bounds a and b. Interval is provided by the system module Integers.

Applying the transformations to the definition of ReverseList gives the normal statements in Figure 3.2, where NotReversed and Match are new predicates and ReversePos is defined as before.

With this transformed program, ReverseList can only be used to check that one list is the reverse of the other. In particular, the goals

```
<- ReverseList([1,2,3,4],z).</pre><- ReverseList(z,[1,2,3,4]).</pre>
```

will both flounder. To see this, consider the first of these goals. The call to Length(xs1, 1) will succeed with 1 bound to 4 and the call to Length(xs2, 1) will then bind xs2 to a list of length 4 with all elements unique variables. Thus, with a safe computation rule, since xs2 is not ground, the call to "NotReversed(xs1, xs2, 1) will flounder. The behaviour will be similar for the second goal, although since the DELAY declaration for Length causes the atom to delay if both arguments are variables, Length(xs2, 1) will be evaluated before Length(xs1, 1).

In this example, the subformula containing the universal quantifier is an implication formula of the form $V \to W$. This is the most natural way to use a universal quantifier in a program, since V restricts the domain of the quantified variables, while W gives the condition that must be satisfied by all the values in the restricted domain.

```
PREDICATE ReverseList : List(a) * List(a).
ReverseList(xs1, xs2) <-
   Length(xs1, 1) &
   Length(xs2, 1) &
   ALL [i] (1 = \langle i = \langle 1 - \rangle \text{ReversePos}(xs1, xs2, i, 1)).
PREDICATE ReversePos :
   List(a) * List(a) * Integer * Integer.
ReversePos(xs1, xs2, i, 1) <-
   Element(xs1, i, x) &
   Element(xs2, 1-i+1, x).
PREDICATE Element :
   List(a) * Integer * a.
Element([x|_], 1, x).
Element([_|xs], i, x) \leftarrow
   i > 1 &
   Element(xs, i-1, x).
```

Figure 3.1: A program for reversing a list

I now outline an alternative computational model for logic programs, SLDQE-resolution, which generalises SLDNF-resolution in that it is capable of evaluating goals such as those above for ReverseList without floundering. This work was published in [16]. Formal definitions for SLDQE-resolution can be found in Appendix A. The quantifier evaluation technique described here was derived from work on bounded quantifications. A bounded quantification [5, 84] is a quantification that ranges over a finite domain [78]. These were introduced to logic programming to enhance expressiveness [5] and enable repetitive computations to be implemented more efficiently by using iteration rather than recursion [8, 1]. It has also been shown that parallel

```
ReverseList(xs1, xs2) <-
   Length(xs1, 1) &
   Length(xs2, 1) &
        " NotReversed(xs1, xs2, 1).

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

NotReversed(xs1, xs2, 1) <-
   1 =< i =< 1 &
        " Match(xs1, xs2, i, 1).

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

Match(xs1, xs2, i, 1) <-
   ReversePos(xs1, xs2, i, 1).</pre>
```

Figure 3.2: The normal form of ReverseList

implementations of bounded quantifications can obtain good speedups over sequential processing [3, 7]. A bounded quantification $\forall \overline{x}V \to W$ restricts the form of V so that the number of iterations required, which is the number of solutions to $\leftarrow V$, can be determined at compile time; no such restriction is needed when the formula is evaluated by SLDQE-resolution.

A formal semantics and computational model (SLDB-resolution) for bounded quantifications was given by Voronkov in [84, 83]. Voronkov's logic has a simple type system, and permits bounded quantifications of the form $(\forall x \in l)V$ or $(\forall x \sqsubseteq l)V$, where l is a list term, V is an arbitrary formula, and \in and \sqsubseteq denote the list membership and list suffix relations respectively.

An alternative transformation technique using unfold/fold transformations to synthesise Horn clause programs from programs containing universally quantified implication formulas is described by Tamaki and Sato in [72]. The precise conditions when the technique works are not explicitly given, making comparisons difficult. However, the theoretical framework for SLDQE-resolution is at least as powerful as the techniques of [72]. Each of the examples given in [72] has been implemented in Gödel.

SLDQE-resolution extends the idea of bounded quantification by providing an evaluation method for formulas of the form $\forall \overline{x}V \to W$, where V can be any formula. Informally, solutions are obtained provided the goal $\leftarrow V$ has a finite number of computed answers that bind all the variables in \overline{x} to ground terms; a more precise condition is given below. Negation as failure is cleanly integrated into the SLDQE model. Further work is needed to clarify the potential gains for parallel implementations of SLDQE-resolution.

A restricted quantification is a formula of the form $\forall \overline{x}(V \to W)$. It is atomic if V and W are atoms. W is called the head and V the body.

Note that a negative literal $\neg A$ is a special case of a restricted quantification since it can be expressed as $\forall \overline{x}(A \to False)$ where \overline{x} is empty.

A quantified-normal (q-normal) formula F is of the form $F_1 \wedge \cdots \wedge F_n$ where each F_i is either

- 1. an atom (called an atomic conjunct of F) or
- 2. an atomic restricted quantification.

A q-normal statement is a statement whose body is q-normal. A q-normal program is a program whose statements are all q-normal.

Since a negative literal is equivalent to an atomic restricted quantification, the definition of a q-normal program includes a normal program as a special case. It is therefore possible to use a set of transformations similar to the Lloyd-Topor transformations above to transform a program into a q-normal program. Changes are required only to transformation 5, to express negation in terms of a restricted quantification, and to transformation 10, which now has two cases. The resulting q-normal program can then be evaluated using SLDQE-resolution. Proofs of termination and correctness for this modified transformations can be found in [16].

$$5a \ A \leftarrow X \land \neg \exists \overline{x} V \land Y, \ \Sigma \qquad \mapsto A \leftarrow X \land \forall \overline{x} (V \to False) \land Y, \ \Sigma$$

$$10a \ A \leftarrow X \land \forall \overline{x} V \land Y, \ \Sigma \qquad \mapsto \begin{cases} A \leftarrow X \land \forall \overline{x} (q(\overline{v}) \to False) \land Y \\ q(\overline{v}) \leftarrow \neg V \end{cases}, \ \Sigma \cup \{q\}$$

$$\overline{v} = \text{fvar}(V)$$

$$10b \ A \leftarrow X \land \forall \overline{x} (V \to W) \land Y, \ \Sigma \mapsto \begin{cases} A \leftarrow X \land \forall \overline{x} (q(\overline{v}) \to r(\overline{w})) \land Y \\ q(\overline{v}) \leftarrow V \end{cases}, \ \Sigma \cup \{q, r\}$$

$$V \text{ or } W \text{ is not atomic}$$

$$\overline{v} = \text{fvar}(V), \ \overline{w} = \text{fvar}(W).$$

In the definition of SLDQE-resolution, a goal statement for a program P is used instead of the usual notation $\leftarrow T$ for a goal for P. A goal statement is a statement. In particular, a goal $\leftarrow T$ can be written $q(\overline{x}) \leftarrow T$ where $\overline{x} = \text{fvar}(T)$ and q is a new predicate. This facilitates a more elegant formulation of the procedure for the restricted quantifications. The use of goal statements is in line with earlier observations in [53] that a resultant encapsulates most of the information about a derivation. An SLDQE-derivation (and tree) for a goal statement and a program is an extension of an SLD-derivation (and tree) for processing the restricted quantifications. The definitions for SLDQE follow closely those for SLDNF in [50]. The difference is that the selected subformula in a goal may now be either an atom or an atomic restricted quantification, and if it is the latter it must be evaluated using a quantifier evaluation step. Formal definitions of SLDQE-trees and SLDQE-refutations can be found in Appendix A¹; a less formal description is given here.

If the body of a goal statement is of the form

$$X \wedge \forall \overline{x}(V \to W) \wedge Y$$

where $\forall \overline{x}(V \to W)$ is the selected formula and \overline{x} is a subset of the free

¹An alternative definition of an SLDNF-tree, more generally applicable that that in [50], is given in [2]. A reformulation of the SLDQE-tree definition in Appendix A along similar lines would be straightforward.

variables in V, the quantifier evaluation step replaces the body by

$$X \wedge W\phi_1 \wedge \cdots \wedge W\phi_m \wedge Y$$

where $\{\phi_1,\ldots,\phi_m\}$ is a finite set of all computed answers for $W\leftarrow V$ with the given program using an SLDQE-derivation. The theoretical requirement for the soundness of quantifier evaluation is that for all $x\in \overline{x}$, $x\phi_i$ is ground and for all $y\in \text{fvar}(V)\backslash \overline{x}$, $y\phi_i=y$ for $1\leq i\leq m$. That is, every substitution ϕ_i must bind all the quantified variables to ground terms, and must not bind any of the free variables in the atomic restricted quantification that appear in the body; if this condition does not hold, the formula cannot be selected. If the selected formula is a restricted quantification $V\to W$, then the answer set for $W\leftarrow V$ has to be obtained using a subsidiary SLDQE-tree. Like an SLDNF-derivation, an SLDQE-derivation flounders when a goal is reached that consists entirely of atomic restricted quantifications that cannot be selected.

In a practical implementation, a stronger condition for the selection of restricted quantifications is normally required, to avoid the inefficiency of computing all the ϕ_i in order to determine that the restricted quantification was not selectable. A suggested condition is that restricted quantification is selectable if $\operatorname{fvar}(V)\backslash \overline{x}=\emptyset$ and, when a restricted quantification is selected, the substitutions ϕ_i are generated and checked to see that they bind every $x\in \overline{x}$ to a ground term. If they do not, the computation is aborted with a run-time error. This is reasonable because, if the set $\operatorname{fvar}(V)\backslash \overline{x}$ is empty, the set of computed answers $\{\phi_i\}$ will not be altered by selecting the formula later in the computation. A computation aborted for this reason is therefore certain to have floundered eventually. It is shown in Section 3.3.2 that with this stronger safeness condition, SLDQE-resolution can be implemented neatly in SICStus Prolog.

Consider again the program and goal statements for ReverseList given above. These are q-normal. Using SLDQE-resolution, neither of the goals flounder and both have the computed answer z = [4,3,2,1].

Pragmatically, the use of a restricted quantification to implement negation via transformation 5a is not a good idea. Let V be an atom, and $\overline{x} = \text{fvar}(V)$. Consider the behaviour of a subformula $\neg \exists \overline{x} V$ appearing in the body of a statement. Using SLDQE-resolution and transformation 5a,

the formula $\forall \overline{x}(V \to False)$ will only be found to fail after all the computed answers to the goal statement $False \leftarrow V$ have been found. In contrast, the implementation via transformation 5 and SLDNF-resolution needs only find one computed answer for $\leftarrow V$ for the subgoal to fail. The Gödel implementation therefore does not use transformation 5a, but is based on a hybrid of SLDNF- and SLDQE-resolution.

Another pragmatic consideration is that the introduction of new predicates is best avoided, because it muddies the relationship between the transformed program and the original. This has implications for debugging and meta-programming. Practically, transformation 5 is not used; instead, the normalised program is required to provide a direct implementation of negated existential quantifiers. New predicates are also introduced by transformations 10a and 10b. It is not necessary that they do so, but the restriction to atomic restricted quantifications simplifies the presentation and proofs in [16]. Gödel therefore uses variants of transformations 10a and 10b that do not introduce new predicates.

Transformations 1, 7 and 8 handle disjunctive formulas in a manner that potentially introduces inefficiency, by duplicating the computation of goals that appear before the disjunctive formula. Again, the simplest solution is to require a direct implementation of disjunction in the normalised program, and the Prolog connective ";" serves this purpose.

Before compilation to Prolog, a Gödel program is transformed to a normalised form in which all subformulas in the body of a statement are of the form $V \wedge W$, $V \vee W$, $\neg \exists \overline{x} W$ or $\forall \overline{x} (V \to W)$, plus conditionals and commits. The transformation replaces any subformulas not in this form with logically equivalent subformulas, and renames all quantified variables so that they are unique within the statement in which they appear.

3.3.2 Translation to Prolog

I now turn to the translation of the ground representation of a Gödel program into a Prolog program that effectively implements the semantics of the Gödel object program. The translation has been simplified for clarity of exposition, and does not deal with all the features of Gödel. In the actual implementation of Gödel, the generated Prolog code is more complex, because it has to deal with additional bookkeeping associated with the Gödel commit, set unification, and the detection of floundering.

The flat names of Gödel symbols, which are represented by terms of type Name, are most easily translated into Prolog symbols that contain all four components of the flat name in some form. Because flat names are unique within the flattened form of a Gödel program, the resulting symbol will then be unique within the resulting Prolog program. The Prolog symbol is generated by first taking one of the characters C, F, O, or P, depending on whether the Gödel symbol is a constant, function, proposition, or predicate², respectively, and appending the arity of the symbol. The result is then prepended with module name and symbol name components of the flat name, separated by ".". For example, the Gödel constant Topper declared in a module called Hats is represented by the Name term Name{"Hats", "Topper", Constant, 0), and this is translated into the Prolog atom 'Hats. Topper. CO' There is one notable exception to this treatment of names: lists in Gödel are translated directly into Prolog lists, because most Prolog implementations have internal optimisations for handling lists, and it is desirable to take advantage of these. Literals such as strings and numbers are also translated into their equivalents in Prolog; Gödel strings are translated to Prolog atoms.

The representation of a Gödel atom can then be translated directly into a Prolog atom with the same structure, by systematically replacing the variable representations with Prolog variables. For example, the representation of the Gödel atom Append([], xs, xs), where Append is declared in module Lists, becomes the Prolog atom 'Lists.Append.P3'([], A, A).

Next, I turn to the translation of the body of a Gödel statement. First, all quantified variables in the body are renamed, so that all variables appearing in the body, whether quantified or not, have unique names within the statement. A sequence of transformations is then applied, as explained in Section 3.3.1, to obtain a formula equivalent to the original body, but in which every subformula contains only conjunctions, disjunctions, negated existentials (of which negative literals are a special case), conditionals, and restricted quantifications. The translation of each kind of subformula is described below. Note that I write about Gödel formulas directly for brevity, although it is of course the ground representation of those formulas that is translated by the Gödel code generator.

²Base types and type constructors never need to be translated into Prolog

In the following, U, V and W denote subformulas in the transformed body of a Gödel statement, and U', V' and W' denote the Prolog code equivalent to U, V and W according to this translation. If \overline{x} is a list or set of Gödel variables, and \overline{x}' is a list of Prolog variables, then $\overline{x}' \leftarrow \overline{x}$ indicates that a mapping from the variables in \overline{x} to those in \overline{x}' is used to translate the variables in \overline{x} into Prolog variables.

A conjunctive Gödel subformula V&W is simply translated into the conjunctive Prolog expression V', W'. Similarly, a disjunctive Gödel subformula $V \setminus W$ is translated into the disjunctive Prolog expression V'; W'.

To translate a Gödel subformula SOME \overline{y} V, it is necessary to make use of the control facilities of SICStus Prolog so that the subformula is not selected while it contains free variables. This will ensure that the implementation of negation is sound. SICStus Prolog provides a predicate freeze(X, Goal) that suspends the execution of Goal until X is instantiated to a non-variable term. The freeze predicate can be used to implement a predicate gfreeze(Xs, Goal), that takes a list of variables Xs and suspends the execution of Goal until every variable in the list is instantiated to a ground term³. The subformula is then translated into the Prolog code gfreeze(\overline{x}' , V'), where $\overline{x}' \leftarrow \text{fvar}(V) \backslash \overline{y}$. Extracting the free variables from the subformula at compile-time saves doing so at run-time.

A simple Gödel conditional IF U THEN V ELSE W is translated to the Prolog code $\operatorname{\sf gfreeze}(\overline{x}', (U' \to V' ; W'))$, where $\overline{x}' \hookleftarrow \operatorname{\sf fvar}(U)$. A full Gödel conditional IF SOME \overline{y} U THEN V ELSE W is translated to the Prolog code $\operatorname{\sf gfreeze}(\overline{x}', \operatorname{\sf if}(U', V', W'))$, where $\overline{x}' \hookleftarrow \operatorname{\sf fvar}(U) \backslash \overline{y}$. Note that the SICStus Prolog predicate $\operatorname{\sf if}(P, Q, R)$ differs from the usual Prolog ifthen-else construct in that it explores all the solutions for P. That is, $\operatorname{\sf if}/3$ provides a $\operatorname{\sf soft}$ or $\operatorname{\sf shallow}$ cut.

A subformula that is a restricted quantification ALL \overline{y} ($V \rightarrow W$) is treated as follows. All variables in $fvar(V)\backslash \overline{y}$ are required to be bound to ground terms before the restricted quantification can be selected. As explained in Section 3.3.1, this condition is more restrictive than the theoretical requirements, but it can be checked much more efficiently. The subformula is translated into the Prolog code

 $^{^3{\}rm The}$ Gödel implementation was developed before the when/2 predicate of SICStus Prolog 3 became available

Figure 3.3: The definition of try

```
gfreeze(\overline{x}', (gsetof(\overline{y}', V', Ps), try(Ps, \overline{y}', W, \overline{z}')))
where \overline{y}' \longleftrightarrow \overline{y}, \overline{x}' \longleftrightarrow \text{fvar}(V) \setminus \overline{y}, and \overline{z}' \longleftrightarrow \text{fvar}(W) \setminus (\text{fvar}(V) \cup \overline{y}).
```

The predicate gsetof/3 is part of the Gödel run-time library, and differs from Prolog's setof/3 in that gsetof(T, G, As) generates a run-time error if any of the terms in As is non-ground.

The predicate try/4 is also from the Gödel run-time library. Its definition in Prolog is given in Figure 3.3.

The call to gsetof will compute Ps, the list of lists of the values representing the computed answers ϕ_1, \ldots, ϕ_m for $W \leftarrow V$. These values appear in the same order as the corresponding variables in the list \overline{y}' . The call to copy_term/2 in the definition of try makes a copy of W with fresh variables, while keeping track of which of those new variables are copies of variables in \overline{x}' , and which are copies of variables in \overline{z}' . The function c is a dummy function that groups the terms together. The two equalities then instantiate the copy of W appropriately.

It can be observed that there is a possibility for coroutining between gsetof and try, that improves efficiency by detecting early failure⁴. That is, as soon as $W\phi_i$ is found to fail for some i, the whole restricted quantification can fail without the need to generate the remaining $\phi_{i+1} \dots \phi_m$. However, the implementation of setof in SICStus Prolog does not support coroutining.

⁴Thanks to Jonas Barklund for this suggestion.

An example of the compilation of Gödel under this scheme is given below. This is the result of compiling the definition of ReverseList from Figure 3.1 into Prolog.

```
'SLDQE.ReverseList.P2'(A, B) :-

'Lists.Length.P2'(B, D),

'Lists.Length.P2'(B, D),

gfreeze([D],

gsetof([E], Integers.Interval.P3'(1, E, D), F),

try(F, [H], 'SLDQE.ReversePos.P4'(A, B, H, D), [A, B])

).
```

The translation of predicate definitions with DELAY declarations is straightforward, and I explain it by means of an example. Using the freeze/1 facility of Prolog, a predicate gdelay/2 can be defined, that takes as first argument a direct translation of the body of a DELAY declaration into Prolog syntax, and as second argument a Prolog goal. Given a delay condition, gdelay/2 suspends the execution of its second argument until the condition in the first argument is satisfied. To ensure that the call pattern matches the head of a DELAY declaration before the condition is evaluated, new auxiliary predicates are introduced.

The predicate Sorted from Figure 2.5 has the following DELAY declaration.

```
DELAY Sorted([]) UNTIL True;
Sorted([_|x]) UNTIL NONVAR(x).
```

Say Sorted is declared in a module M. The Prolog code below is translation of the control part of Sorted.

```
'M.Sorted.P1'(X) :-
gdelay(nonvar(X), 'M.Sorted.P1.1'(X)).

'M.Sorted.P1.0'([]) :-
'M.Sorted.P1.1'([]).

'M.Sorted.P1.0'([X|Y] :-
gdelay(nonvar(Y), 'M.Sorted.P1.1'([X|Y])).
```

The definition of 'M.Sorted.P1.1' is the Prolog translation of the definition

of Sorted. In practice, most real Gödel programs have very few DELAY declarations with structure in the head.

It is worth noting that gdelay/2 incorporates an ad hoc optimisation for arguments of type Program; these arguments are known to be either uninstantiated variables or very large ground terms. There is no need to check that such arguments are ground if they are non-variable, and such a check is potentially expensive.

3.4 Reflective Interpreters

This section is a brief discussion of the implementation of the predicates in the Programs module that provide full-scale interpreters for Gödel object programs, such as Succeed and Compute.

Semantically, these predicates simply invoke interpreters for the ground representation, and could be implemented that way, but it is better to avoid interpreting the ground representation directly. It should be more efficient to reflect the representation of program and goal down to the object level, perform the computation directly to obtain a computed answer, and reflect this answer back up to its meta-level representation. This implementation trick is of course completely transparent to the programmer, and the separation between object program and meta-program is maintained. No reflective inference rules are involved.

The straightforward method of reflecting and compiling the entire object program each time an interpreter is called introduces unacceptable overhead, except when the object-level computation to be performed is very large in comparison with the compilation process, and this is not usually the case. Indeed, for small object-level computations, a direct interpreter would often be faster.

To reduce the overhead of reflection, the system retains the code compiled by the reflection process and only recompiles a module of an object program when necessary. Recompilation is required when the module represented in the object program to be executed is different from a module with the same name that was previously reflected to produce the compiled code. The integer argument to the Code function (see Section 3.2.4) helps the system to keep track of such changes.

Dividing the representation of the object program code into separate

modules allows the reflection operation to recompile module by module only those modules that have changed since the last time they were reflected, rather than having to recompile the whole object program. The implementation in Gödel is unfortunately not completely sound, and can fail to detect the need to recompile an object program module in certain obscure circumstances. It does, however, satisfactorily demonstrate the idea.

A better scheme would be to actively reflect and recompile only predicates whose definition has changed, or even to operate statement by statement. This leads towards a representation for dynamic object theories like that used in MetaProlog [12].

The efficient implementation of such reflective interpreters is problematic, given only the facilities of SICStus Prolog. It requires low-level support, to handle non-destructive updates to the compiled code, and to provide a trailing mechanism to undo changes on backtracking. This line of investigation was not pursued further in Gödel.

Chapter 4

An Analysis of Gödel Meta-programming

This chapter contains an analysis of some important features the Gödel meta-programming library that was described in Section 2.2, with particular emphasis on finding ways to improve the performance of interpreters for Gödel object programs in the ground representation. A substantial part of the work in this chapter was published in [15].

Since the aim of a computation using SLD-resolution is to compute an answer substitution, and the most complex step in the process is the unification of the selected atom with the head of the clause, I consider first the representation of substitutions and how this affects the performance of simulated unification for terms in the ground representation. I then examine an evolutionary series of interpreters for definite Gödel programs, which leads towards a style of meta-programming that avoids some of the inefficiencies inherent in the interpretation of programs in the ground representation. Specialisation of interpreters to particular object programs is an important candidate technique for solving the performance problems. Some experiments in specialising interpreters using the partial evaluator SAGE will be described. SAGE was developed by Gurr [33, 32], using the Gödel metaprogramming library. These experiments lead to some conclusions about the facilities that might be added to the implementation, beyond those found in a typical logic programming system such as SICStus Prolog, to support efficient interpretation of the ground representation. Finally, I describe a breadth-first interpreter, a style of interpreter that presents considerable challenges for the techniques given previously.

For conciseness and clarity, in what follows the bulky terms with the function Name that Gödel uses to represent symbols will usually be omitted, and in their place write the declared name of the represented symbol with a prime. For example, F' is to be read as

```
Name(module, "F", category, arity)
```

where the module, category and arity components will either be obvious or unimportant in the context.

4.1 Substitutions and Unifications

The essential difference between programs that use the ground representation and those that use the traditional, non-ground Prolog style of metaprogramming is that, in simulating object-level operations such as unification, the former must explicitly handle the representation of variables, while the latter can rely on the underlying system to perform this task on the meta-variables that stand in place of object variables.

The complexity of the unification process has been extensively studied, and the most efficient algorithms such as those proposed in [54] and [64] are linear in the size of the input terms, despite performing the occur check. However, these algorithms are not commonly used to implement unification in logic programming languages, because they involve the use of data structures that do not fit easily with other requirements for efficient implementation; instead, most Prolog implementations obtain efficiency by omitting the occur check. The linear unification algorithms are also not directly compatible with meta-level simulations of object-level unification; the Paterson-Wegman algorithm [64] for example relies on destructive update of a graphical term representation. The analysis of the cost of unification in meta-programming that follows in this section concentrates on simple unification algorithms that compare terms subterm-by-subterm. The applicability of more sophisticated unification algorithms to meta-programming is not considered further in this work.

An example of common Prolog meta-programming practice is given by programs that perform some variant of the default unification process, perhaps by adding the occur check. They usually work by using the built-in

```
unify(X,Y) :-
   var(X), var(Y), X = Y.
unify(X,Y) :-
   var(X), nonvar(Y), not_occurs_in(X, Y)
   Y = X.
unify(X,Y) :-
  nonvar(X), var(Y), not_occurs_in(Y, X)
   Y = X.
unify(X, Y) :-
   nonvar(X), nonvar(Y),
   functor(X, F, N), functor(Y, F, N),
   unify_args(N, X, Y).
unify_args(0, X, Y).
unify_args(N, X, Y) :-
   N > 0, N1 is N - 1,
   arg(N, X, XArg), arg(N, Y, YArg),
   unify(XArg, YArg),
   unify_args(N1, X, Y).
```

Figure 4.1: A non-declarative simulation of unification in Prolog

primitives functor/3 and arg/3 to compare the functors and arguments of the input terms recursively, as in Figure 4.1. This type of program is often used for teaching; several examples can be found in Sterling and Shapiro [73]. These programs still rely on, and obtain their efficiency from, the underlying system's treatment of the meta-variables that stand for object variables. It goes without saying that these techniques are not declarative, and one consequence of this is that the programs change their input arguments (by binding the meta-variables they contain) in a way that is seen to be very awkward once a declarative style of meta-programming has become familiar.

The program in Figure 4.1 might properly be viewed as an extension to the Prolog language, providing a procedure that performs unification with the occurs check. Viewed purely as a meta-program, however, the disadvantages inherent in the lack of a declarative meaning become apparent. A brief example will illustrate the clarity and expressiveness gained by emulating unification in the ground representation. Consider the goal (I revert temporarily to Prolog syntax): - unify(U,V) where U is instantiated to p'(f'(X),X), intended to represent the object-level atom p(f(A),A), and V is instantiated to p'(Z,c'), intended to represent the object-level atom p(B,c). After the execution of the goal, both U and V will be instantiated to p'(f'(c),c), so that the object-level atom represented by both U and V has been changed. This is an inconvenient way to return the result of a meta-computation. On the other hand, an analogous call to the Gödel predicate UnifyAtoms in Syntax, using the ground representation, leaves the input representations unchanged, and returns the representation of a substitution. If desired, this substitution can subsequently be applied to either input representation to obtain the representation of a common instance.

Now (returning to Gödel syntax) consider the problem of a meta-program that is to simulate the unification of the representations of the object-level terms F(A,y) and F(x,x). As has been seen, in a non-ground representation, the unification is achieved by simply unifying the representations; that is, by solving an equation such as F'(A',u) = F'(v,v), where F' and A' are the symbols of the meta-language representing F and A respectively. In the process, the meta-variable v is bound to the constant A', and because both occurrences of v are represented internally by the same physical memory location, the binding is automatically propagated to the second occurrence, and thus to v when v and v are bound together. All in all, each term is scanned once, and two memory locations are changed (ignoring any trailing that may be necessary) to obtain the term F'(A',A') representing F(A,A).

Contrast this with the ground approach. Continuing the same example means performing a meta-level simulation of unification on the representations Term(F',[CTerm(A'),Var("y",0)]) and Term(F', [Var("x",0),(Var("x",0)]). Suppose that a simple unification algorithm such as that found in [50] is employed to implement a meta-predicate like GroundUnify below.

```
PREDICATE GroundUnify:

Term  % Representation of a term.

* Term  % Representation of a term.

* Term.  % Representation of the term formed

% by applying the mgu of the terms in

% the first two arguments to the

% first argument.
```

Beginning by comparing the terms from left to right, it soon becomes necessary to record the binding of variable v to constant A. In the nonground representation, this was done by simply instantiating the metavariable standing in place of the representation of v to the constant A' representing A, and all the other occurrences of this same meta-variable representing v were also instantly affected. Is it possible to use an analogous method in the ground case? The idea is to record bindings by everywhere replacing the representation of the bound variable by the representation of the value to which it has been bound. Thus all occurrences of Var("v",0) must be replaced by CTerm(A') in the representations of both terms. However, the representation of v is a ground term, rather than a meta-variable, so this cannot be done destructively as in non-ground. Instead, the process of searching both representations for occurrences of Var("v",0) and replacing each one with CTerm(A') will construct a complete new copy of the representation of each term. Now notice that this has been done simply to record a single variable binding. Scanning and copying both terms every time a variable is bound involves an excessive overhead in space and time. Recording bindings via the explicit representation of substitutions looks to be a much more promising route to efficiency.

Recall that Gödel's Syntax module exports an abstract type TermSubst for the representation of substitutions. The most direct and obvious way to represent substitutions is as lists of bindings.

```
CONSTRUCTOR Binding/1.

FUNCTION
    / : xFx(10) : Term * Term -> Binding(Term);
    Subst : List(Binding(Term)) -> TermSubst.
```

Thus the term $\operatorname{Subst}([\operatorname{Var}(1)/[t_1],\ldots,\operatorname{Var}(n)/[t_n]])$ represents the object-level substitution $\{v_1/t_1,\ldots,v_n/t_n\}$, where $\operatorname{Var}(1),\ldots,\operatorname{Var}(n)$ represent the variables v_1,\ldots,v_n and $[t_1],\ldots,[t_n]$ are the representations of the terms t_1,\ldots,t_n respectively.

During the standard unification process, it is necessary both to look up bindings in the substitution whenever a variable is encountered, and to add new bindings to the substitution as they are made. Adding the binding of variable v_m to term t_m to the substitution θ already constructed is done by forming the composition $\theta\{v_i/t_i\}$. If θ is $\{v_1/t_1,\ldots,v_n/t_n\}$, then all occurrences of v_m in t_1,\ldots,t_n must be replaced by t_m . Looking up a binding in the representation of a substitution means conducting a linear search of the list. Adding a binding to the representation $[Var(1)/[t_1],\ldots,Var(n)/[t_n]]$ will necessarily involve constructing a copy of the list on current Prolog systems. I will return to this problem in the context of interpreters in Section 4.2.2.

Two more possibilities suggest themselves for efficient simulation of unification in the ground representation: structure reuse and reflection. Structure reuse using liveness analysis techniques such as described by Mulkers [60] might, in theory, permit fast unification by allowing destructive assignment to variable representations in one or both of the structures to be unified, but this technology is not mature and there is a long way to go before practical analyses are accurate enough to make this work well, especially in languages like Gödel in which computation does not necessarily proceed left to right. Flexible computation rules complicate abstract analysis and reduce its accuracy. Implementing unification by reflection, that is by actually performing the unification at the object-level, is more promising. However, it suffers from the fact that in purely declarative meta-programming, the reflective implementation must be completely hidden. One possibility involves constructing the object-level structures denoted by the ground representation, unifying them using the efficient unification routine in the underlying

system, and then mapping the result back into the ground representation. However, there is still a cost associated with the translation step, and since the operation is internally non-logical, its logic cannot be made available to general optimisation processes such as partial evaluation which would prevent the effective specialisation of unification by SAGE demonstrated in Section 4.3.2. These are important disadvantages. It is also possible to imagine an alternative reflective implementation, in which object-level structures are maintained in parallel with their representations at the meta level. The difficulty here is that the object-level unification essentially destroys these parallel structures, so copies of the structures to be unified must again be made before unifying them. Performing compile time analysis to determine when this is necessary requires similar analysis to that for structure reuse.

4.2 Simple Interpreters

The greatest computational expense introduced by the ground representation occurs when representations of substitutions are manipulated, as in unification or resolution. In this section we look at interpreters that do no more than implement SLD-resolution, in order to demonstrate how metaprograms which use the ground representation may be made efficient. Such interpreters have the advantage of being both simple and familiar in principle, while at the same time being fully illustrative of the potential inefficiencies of the ground representation.

Even simple SLD-interpreters for the ground representation add value over the non-ground vanilla interpreter, because the result is returned in the form of the ground representation of a computed answer substitution, rather than by instantiating meta-variables, and this is often what is required. In addition, it is straightforward to extend these SLD-interpreters, for example to return a proof term, or to return all the input clauses used in a derivation as part of a knowledge assimilation procedure like those described in [30], [31]. The latter task requires the ground representation if it is to be done declaratively. I have not included enhanced interpreters in the discussion, partly because they are easily constructed using well-known techniques, and partly because the additional complexity might obscure the argument.

While I use only simple interpreters to illustrate the efficiency issues for

declarative programming in this paper, this does not imply that the optimisation techniques proposed are applicable only to such programs. The Gödel meta-programming library described in this thesis has been used in the construction of a range of quite sophisticated meta-programs, including coroutining interpreters, theorem provers, declarative debuggers and program specialisers such as the SAGE partial evaluator.

4.2.1 Instance Demo

In passing, it is worth discussing the interpreter proposed by Hill and Gallagher [36] and based on an idea outlined in [46], called **Instance Demo**. **Instance Demo** is an interpreter for programs in the ground representation, yet it uses meta-variables for unification and relies on the underlying system to manage variable bindings and backtracking in exactly the same way as the vanilla interpreter does in the non-ground representation. Figure 4.2 shows part of this interpreter, written using Gödel's meta-programming facilities. The predicate InstanceOf is intended to be true when its second argument represents an instance of the formula represented by its first argument. However, when called with the second argument unknown, its effect is to return the formula in the first argument with all the representations of variables replaced by meta-variables. An alternative version of InstanceOf might produce representations of all possible instances on backtracking; this would require the presence of a Program argument to provide the alphabet and ensure that only well-typed instances are generated. The predicate IClause uses InstanceOf to find instances of program clauses.

Instance Demo has the nice property that it is possible to remove the calls of IClause by partial evaluation, thus specialising the interpreter for a particular object program. The residual code then looks and operates exactly like a specialised vanilla interpreter, and is equally efficient.

At first sight this looks a very promising way to obtain efficient interpreters for the ground representation. However, because **Instance Demo** is so similar to non-ground based interpreters, it suffers similar limitations. It does not effectively capture the procedural semantics of the object program: it cannot be used to vary the search strategy much from that of the underlying system, for example to conduct a breadth-first search, and it cannot give information about variable bindings. The answer it provides to the object program and goal is some instance of the original goal, but that

```
PREDICATE IDemo :
             % A definite program.
   Program
 * Formula
             % The body of a definite
             % goal for the program.
 * Formula. % An instance of the goal body
             % that is a logical consequence
             % of the program.
IDemo(program, goal, goal_instance) <-</pre>
   InstanceOf(goal, goal_instance) &
   IDemo1(goal_instance, program).
PREDICATE IDemo1 : Formula * Program.
IDemo1(empty, _) <-</pre>
   EmptyFormula(empty).
IDemo1(conjunction, program) <-</pre>
   And(left, right, conjunction) &
   IDemo1(left, program) &
   IDemo1(right, program).
IDemo1(atom, program) <-</pre>
   IClause(program, atom, clause) &
   IsImpliedBy(atom, body, clause) &
   IDemo1(body, program).
PREDICATE IClause : Formula * Formula * Formula
IClause(program, atom, clause_instance) <-</pre>
   PredicateAtom(atom, name, _) &
   DefinitionInProgram(program, _, name, clauses) &
   Member(clauses, clause) &
   InstanceOf(clause, clause_instance).
```

Figure 4.2: The **Instance Demo** interpreter

instance will sometimes be incompletely specified, because it can contain meta-variables. These factors render instance demo significantly less useful than interpreters that handle the substitution of variables directly, although it is certainly applicable in circumstances in which a non-ground interpreter would be appropriate.

4.2.2 An SLD Interpreter using Composition

There are relatively few examples in the literature of interpreters that explicitly represent bindings for variables in the ground representation, and most of those that exist, for example in [11] or [38], are based on explicitly mimicking the standard mechanism of SLD-resolution (as it appears for example in [50]).

Thus, the main loop of a typical interpreter first selects an atom in the current goal, then finds a clause with a matching predicate in the program, renames the variables in the clause, computes an mgu for the selected atom and the head of the renamed clause, composes this mgu with the sequence of mgus previously computed (which result will eventually form the computed answer we want), constructs a new goal from the remains of the current goal and the body of the renamed clause, and finally applies the mgu to the new goal.

Figure 4.3 shows a first attempt at an interpreter for definite programs, based on composition of substitutions. This interpreter uses resultants to keep track of the variables that must be avoided during standardisation apart. It also makes use of a predicate Select which implements a selection rule by selecting the next atom in the body of the resultant to be resolved. The version of Select in Figure 4.4 implements a leftmost selection rule. Figure 4.4 also shows the definition of the Mgu predicate in terms of UnifyAtoms.

All the interpreters in this chapter require a top-level predicate for execution, which is responsible for reading in the object program, setting up suitable arguments for the interpreter, and presenting the result of the computation in human-readable form. The predicate Go in Figure 4.5 is such a test harness for the CDemo program; simple variations on this procedure can be used with any Gödel interpreter. A suitable goal for Go would be

$$\leftarrow$$
 Go("OM", "MyAppend([1,2],[3,4],x)", w)

```
PREDICATE CDemo :
  Program
               % A definite program.
* Formula
               % The head of a resultant.
* Formula
               % The body of the resultant.
* TermSubst
               % A substitution.
* TermSubst.
               % The substitution obtained
               % by composing the term substitution in
               % the fourth argument with the computed
               % answer obtained for this program
               % and resultant.
CDemo(program, rhead, rbody, subst, answer) <-</pre>
   EmptyFormula(rbody) |
   answer = subst.
CDemo(program, rhead, rbody, subst, answer) <-</pre>
   Select(rbody, left, selected, right) |
   StatementMatchAtom(program, _, selected, clause) &
   RenameFormulas([rhead, rbody], [clause], [input_clause]) &
   IsImpliedBy(chead, cbody, input_clause) &
   Mgu(selected, chead, mgu) &
   ComposeTermSubsts(subst, mgu, new_subst) &
   AndWithEmpty(left, cbody, new_rbody1) &
   AndWithEmpty(new_rbody1, right, new_rbody2) &
   ApplySubstToFormula(new_rbody2, new_subst, new_rbody) &
   ApplySubstToFormula(rhead, new_subst, new_rhead) &
   CDemo(program, new_rhead, new_rbody, new_subst, answer).
```

Figure 4.3: An interpreter based on composition of substitutions

```
PREDICATE Select :
                \mbox{\ensuremath{\mbox{\%}}} A (non-empty) conjunction of
   Formula
                % atoms.
 * Formula
                % The conjunction of atoms to
                % the left of the selected atom.
 * Formula
                % An atom selected using a leftmost
                % selection rule.
 * Formula.
                % The conjunction of atoms to
                % the right of the selected atom.
Select(atom, empty_formula, atom, empty_formula) <-</pre>
    Atom(atom) &
    EmptyFormula(empty_formula).
Select(body, left, selected, right) <-</pre>
    And(1, r, body) &
    Select(1, left, selected, r1) &
    AndWithEmpty(r1, r, right).
PREDICATE Mgu :
   Formula
                % An atom.
 * Formula
                % Another atom.
 * TermSubst. % An mgu for the two atoms.
Mgu(v, w, mgu) <-
   EmptyTermSubst(empty) &
   UnifyAtoms(v, w, empty, mgu).
```

Figure 4.4: The predicates Select and Mgu

where the module OM imports the module Lists and contains an appropriate definition of MyAppend. Go uses ProgramCompile from ProgramsIO to read in a representation of the object program, and StringToProgramFormula to parse the concrete syntax of the supplied goal. The object-level goal is also type-checked at this stage, ensuring that it is a valid formula in the language of the given object program. The correctness theorems for Gödel's type system then guarantee that, provided the interpreter is correct, the ensuing interpretation cannot go astray by creating an invalid intermediate goal. When the interpreter completes, RestrictSubstToFormula is used to restrict the resulting substitution to the free variables of the initial goal, in accordance with the definition of a computed answer. The AnswerString predicate is needed to translate the computed answer representation into concrete syntax, since the TermSubst type is an abstract type, and its contents are not otherwise visible to the user. AnswerString is not provided by a Gödel system module, but is easily defined using predicates from Syntax and Programs.

The composition-based interpreter has to do an enormous amount of work in each resolution step. As has been seen, given a simple representation for substitutions as lists of bindings, the unification operation involves linear search within the representation of the partially constructed mgu, and the addition of each new binding constructs a new copy, but the mgu is usually quite small. The explicit Compose operation, however, copies the entire substitution computed up to this point, and this contains bindings for many variables that have been introduced, used and subsequently dropped from the computation. This substitution can become large, and copying its representation is a major overhead. Renaming the variables of the input clause necessitates making a new copy of the clause, and similarly, the application of the mgu to the new goal necessitates copying the goal. All these copying operations are expensive, and this interpreter can be expected to perform very poorly.

In the remainder of Section 4.2, I present some more efficient interpreter designs.

4.2.3 Unify Demo

The first priority is to avoid composing substitutions at all. This is achieved by altering the representation of substitutions, so that they are represented

```
PREDICATE Go:
   String
            % The name of the top-level module of
            % an object program.
            % The body of a goal for the object program,
 * String
            % in concrete syntax.
            % A representation of a computed answer for
 * String.
            % this object program and goal, in concrete
            % syntax.
Go(program_string, goal_string, answer_string) <-</pre>
   ProgramCompile(program_string, program) &
  MainModuleInProgram(program, module) &
   StringToProgramFormula(program, module, goal_string,
      [goal]) &
   EmptyTermSubst(empty_subst) &
   CDemo(program, goal, goal, empty_subst, subst) &
  RestrictSubstToFormula(goal, subst, answer) &
   AnswerString(program, module, answer, answer_string).
```

Figure 4.5: A predicate for testing interpreters

in an effectively uncomposed form. When a new binding is added to a substitution in this representation, it is simply added at the head of the list of existing bindings and the rest of the substitution is unchanged. In this way, reference chains are built up, similar to the reference chains that form on the stack in a standard WAM implementation of Prolog. To determine the result of applying a substitution in this form to some term, it is necessary to follow all reference chains until a result is obtained that contains only variables for which there are no bindings in the substitution. For example, the classical substitution $\{w/z, x/z, y/z\}$ might be represented by the list below.

```
[ Var("w",0)/Var("x",0),
  Var("x",0)/Var("y",0),
  Var("y",0)/Var("z",0)
```

This less literal representation of substitutions differs from the classical representation in an important respect: adding an arbitrary binding could lead to a loop in a reference chain. In fact, only idempotent substitutions can be meaningfully represented in this form, so some generality has been lost, in that we no longer have a representation for every substitution in the sense of [50]. Fortunately, the substitutions generated in an SLD-derivation, by unification of a selected atom with the head of a clause which has been suitably standardised apart, are always idempotent.

It is also not clear how to compose two substitutions in this representation; one way would be to translate them into classical substitutions first, by collapsing all the reference chains, and then use the classical composition algorithm. This is an expensive operation, but the point of this optimisation is to avoid composition entirely. If we limit the operations on substitutions to creating an empty one, unification (with the occur check), and application, this representation is well behaved. In fact, there is a predicate ComposeTermSubsts in the Syntax module, but its implementation is inefficient, and its use is discouraged.

The representation of substitutions in uncomposed form is sufficient to avoid composition during unification, that is within a call of UnifyAtoms, but during interpretation we must also avoid the explicit composition of the new mgu with the answer substitution so far computed. Passing the accumulated substitution to UnifyAtoms as an input argument solves this problem, and this is why UnifyAtoms is designed to expect an input substitution. When the unification process encounters a variable in one of the terms being unified, it first dereferences the variable with respect to the input substitution, before proceeding with the unification. As new bindings are generated during the unification process, they are added to the list of bindings already created by previous unifications. In this way, the direct application of the substitution to the statement at each step of the derivation is also avoided.

Determining the binding of a variable in a given substitution now potentially requires accessing the substitution many times in order to follow the reference chains, whereas in the classical representation it only ever required one access. Each access involves making a linear search of the binding list for the variable concerned. Clearly, there is something to be gained by

improving the efficiency of these accesses.

Recall that in Section 3.1.3 a set of variable representations was introduced in which the variables are uniquely identified by integer indexes. The meta-terms V(1), V(2), V(3), ... are taken to represent the variables v_1, v_2, v_3, ... The root v is simply an arbitrary choice to construct syntactically correct Gödel variables. It would not be practical to expect the authors of object programs to confine their choice of variable names to this limited set, and it is sometimes useful to be able to retain the variable names used in the original program, so these are in addition to the Var("x",0) form, which is still required to represent all other variables. Gödel interpreters can however ensure, when renaming clauses, that all newly introduced variables are of this form.

Now the possibility arises of taking advantage the integer index to obtain faster look up in the substitution, by using a declarative array implementation to represent the binding list. In practice a carefully designed tree structure is used, giving logarithmic access and update times.

The variable indexes also simplify the interpreter considerably, by reducing the information needed for safely renaming the input clause. Instead of carrying around an extra formula containing the variables to be avoided, such as the head of the resultant in the CDemo predicate of Figure 4.3, it is necessary only to remember the smallest variable index that has not previously been used. These considerations lead to the type of interpreter shown in Figure 4.6. The structure of this interpreter is analogous to that of the vanilla interpreter, and it shares the selection rule and search strategy of the underlying system. It still performs explicitly the standard steps of input statement selection, standardisation apart, and unification, but it is much simpler and more efficient than the composition interpreter. However, there is at least one more major optimisation that we can perform, and that is described in the next section.

4.2.4 The Resolve Predicate and SLD Demo

The Resolve predicate, which was introduced in Section 3.1, provides an optimised implementation of a single resolution step. The declaration of Resolve is shown in Figure 4.7, taken from the export part of the Syntax module. When an atom is resolved with respect to some statement in the object program, the statement is first standardised apart, so all the variables

```
PREDICATE UDemo :
               % A definite program.
   Program
 * Formula
               % The body of a definite goal.
 * Integer
               % The first free variable index.
               % No variable with a higher index
               % appears in the goal or the
               % substitution in the fifth argument.
 * Integer
               % The free variable index remaining,
               % after the evaluation of this
               % program and goal.
 * TermSubst
               % A substitution.
 * TermSubst. % The substitution obtained
               % by composing the term substitution in
               % the fifth argument with the computed
               % answer obtained for this program
               % and goal.
UDemo(_, empty_goal, v, v, subst, subst) <-</pre>
   EmptyFormula(empty_goal).
UDemo(program, goal_atom, v_in, v_out, subst, new_subst) <-</pre>
   Atom(goal_atom) &
   StatementMatchAtom(program, goal_atom, statement) &
   StandardiseFormula(statement, v_in, v1, statement1) &
   IsImpliedBy(head, new_goal, statement1) &
  UnifyAtoms(goal_atom, head, subst, subst1).
   UDemo(program, new_goal, v1, v_out, subst1, new_subst).
UDemo(program, goal, v_in, v_out, subst, new_subst) <-</pre>
   And(left, right, goal) &
   UDemo(program, left, v_in, v1, subst, subst1) &
   UDemo(program, right, v1, v_out, subst1, new_subst).
```

Figure 4.6: Unify Demo

```
PREDICATE Resolve :
  Formula
              % Representation of an atom.
* Formula
              % Representation of a statement.
* Integer
              % The first free variable index..
* Integer
              % The free variable index remaining
              % after the resolution step.
* TermSubst
              % Representation of a term substitution.
* TermSubst
              % Representation of the substitution obtained
              % by composing the substitution in the fifth
              % argument with a specific, unique mgu of the
              % atom in the first argument with the
              % substitution in the fifth argument applied,
              % and the head of a renamed version
              % of the statement in the second argument.
* Formula.
              % Representation of the formula obtained by
              % applying the substitution in the sixth
              % argument to the body of the renamed
              % statement.
DELAY Resolve(x, y, z, _, u, _, _) UNTIL
   GROUND(x) & GROUND(y) & GROUND(z) & GROUND(u).
```

Figure 4.7: The Resolve predicate

it contains are replaced with new variables which do not occur elsewhere in the current computation. The implementation of Resolve can take advantage of this knowledge to optimise the resolution process. The fact that these new variables are unique to the renamed statement allows Resolve to simplify many of the necessary unification operations and reduce the number of occur-checks needed. These are essentially the same observations that give rise to the efficient compilation of unification in the WAM [86].

Resolve takes as input arguments an atom (normally the selected atom in the current goal), a statement, a free variable index that can be used to

generate new variables for use in renaming the statement, and a substitution representing the variable bindings created by previous steps in the computation. The renaming is achieved by systematically replacing the variables of the statement by variables with indexes greater than or equal to the third argument. When Resolve succeeds, the fourth argument is instantiated to one more than the highest variable index used in the renaming (or to the value of the third argument, if none is used).

During the resolution step, Resolve generates an mgu by applying the input substitution to the given atom, and unifying the result with the head of the given statement. Notice that the mgu binds a variable in the head to a variable in the selected atom, not the other way around. The bindings in the mgu are added to the input substitution, to form the output substitution in the sixth argument. Thus the substitutions in the fifth and sixth arguments respectively represent the state of the answer substitution before and after the resolution step. The output substitution is applied to the body of the renamed statement to form the new goal in the seventh argument.

Figure 4.8 shows **SLD Demo**, a very simple interpreter for definite programs, based on the Resolve predicate.

SLD Demo is probably the simplest design possible for an interpreter for Gödel's ground representation (with the exception of Instance Demo, which has more limited applicability), while being at the same time built from primitives at a sufficiently low-level to allow the programmer a reasonably useful degree of control over the computation. For example, in SLD Demo, the order in which object-level atoms are selected for resolution depends on the selection rule of the system used to execute the interpreter itself. However by incorporating a Select predicate more sophisticated than that in Figure 4.4, it is easy to alter the selection rule, perhaps to take into account DELAY declarations in the object program. It is also easy to change the computational model completely, and program an interpreter based on Resolve that performs a breadth-first search, as in Figure 4.18. On the other hand, the use of Resolve prohibits modifications of the unification mechanism, such as would be needed to incorporate an extended equality theory.

As an example of a simple extension to **SLD Demo**, and an illustration of the use of the predicate **Compute** from the **Programs** module, the addition of the the clause in Figure 4.9 to the definition of **Demo** will allow it to

```
PREDICATE Demo :
   Program
               % A definite program.
 * Formula
               % The body of a definite goal.
 * Integer
               % The first free variable index.
 * Integer
               % The free variable index remaining
               % after the evaluation of this
               % program and goal
 * TermSubst
               % A substitution.
 * TermSubst. % The substitution obtained
               % by composing the term substitution in
               % the fifth argument with the computed
               % answer obtained for this program
               % and goal.
Demo(_, empty, v, v, subst, subst) <-</pre>
   EmptyFormula(empty).
Demo(program, atom, v_in, v_out, subst, new_subst) <-</pre>
   Atom(atom) &
   StatementMatchAtom(program, _, atom, statement) &
   Resolve(atom, statement, v_in, v1, subst, subst1,
      new_goal) &
  Demo(program, new_goal, v1, v_out, subst1, new_subst).
Demo(program, goal, v_in, v_out, subst, new_subst) <-</pre>
   And(left, right, goal) &
  Demo(program, left, v_in, v1, subst, subst1) &
   Demo(program, right, v1, v_out, subst1, new_subst).
```

Figure 4.8: **SLD Demo**

```
Demo(program, atom, v_in, v_out, subst, new_subst) <-
   Atom(atom) &
   DeclaredInClosedModule(program, _, atom) &
   Compute(program, atom, v_in, v_out, subst, new_subst,
       empty) &
   EmptyFormula(empty).</pre>
```

Figure 4.9: Handling system predicates with Compute

handle atoms in the object goal that have system predicates, that is, any predicate that is declared in the export part of a closed module. Note that the statements defining a system predicate are not accessible to the metaprogram, since such predicates need not be implemented by Gödel code.

The Compute predicate invokes a reflective interpreter (see Section 3.4 for a description of reflective interpreters), and is designed to integrate easily into interpreters based around Resolve. It therefore takes a free variable index and a substitution as input arguments, in addition to representations of an object program and goal, and returns the result of the computation in the form of a new substitution. This substitution is created by adding any new bindings generated by the computation to the input substitution. In its last argument, Compute returns a representation of the body of the last goal reached, in case the computation flounders, or the representation of the empty formula, in case the computation was successful. In the clause above, this last goal is constrained to be empty, so Demo will fail if any computation involving a system predicate flounders.

This clause and the other clause for Demo in Figure 4.8 are mutually exclusive, because StatementMatchAtom will fail for atoms with system predicates.

Figure 4.10 shows the performance of **Unify Demo** and **SLD Demo** interpreting (as object program) the well known *naïve reverse* program on a list of fifty elements. The ground interpreters are compared with the object program executed directly on the Gödel system, and with the non-ground vanilla interpreter interpreting the same program. Since the performance differences are so great, the results from a single benchmark are adequate for

Program	Relative time for naïve reverse
Object program	1
Vanilla	10
Unify Demo	280000
SLD Demo	31000

Figure 4.10: Performance of interpreters

```
MODULE Vanilla.

IMPORT ONaive.

PREDICATE Solve : Formula.

Solve(Empty).

Solve(x And y) <-
Solve(x) &
Solve(y).

Solve(y).

Solve(y).
```

Figure 4.11: A vanilla interpreter in Gödel

the purposes of this discussion. The vanilla interpreter benchmark is shown in Figure 4.11, and the naïve reverse object program for this interpreter is shown in Figures 4.12 and 4.13. The functions ORev and OApp represent the Reverse and Append predicates respectively, and the constant ONil and function OCons represent the usual constant Nil and function Cons for lists.

The table indicates that **SLD Demo** is faster than **Unify Demo** by a factor of almost ten, demonstrating the effectiveness of combining renaming and unification in the **Resolve** predicate and optimising them. However, the interpreter is still three orders of magnitude slower than its non-ground equivalent, and obviously an overhead of this magnitude renders such interpreters almost useless for many practical applications. Note, however,

```
EXPORT ONaive.
IMPORT Integers.
BASE Formula.
CONSTRUCTOR OList/1.
CONSTANT
   Empty : Formula;
   ONil : OList(a).
FUNCTION
         : xFy(110) : Formula * Formula -> Formula;
   And
         : xFx(100) : Formula * Formula -> Formula;
   Ιf
         : OList(Integer) * OList(Integer) -> Formula;
   OApp : OList(Integer) * OList(Integer)
      * OList(Integer) -> Formula;
   OCons : a * OList(a) -> OList(a).
PREDICATE
            Statement: Formula.
```

Figure 4.12: The export part of module ONaive

that the interpreter and the system predicates it depends upon are written almost entirely in Gödel (a very small fraction is written directly in Prolog), and the Gödel code is translated to Prolog by the Gödel compiler. The Prolog system has no built-in support for the ground representation.

A third style of interpreter can be imagined, which makes use of Resolve but carries additional information about the state of the computation by passing a resultant in place of the goal body. Interpreters based on resultants have the advantage that it is easier to eliminate local variables once they have been dropped from the computation. In order to eliminate such intermediate variables, the interpreter must make a pass over the resultant to determine

```
LOCAL ONaive.

Statement(ORev(ONil, ONil) If Empty).

Statement(ORev(OCons(x, xs), ys) If

ORev(xs, zs) And

OApp(zs, OCons(x, ONil), ys)).

Statement(OApp(ONil, xs, xs) If Empty).

Statement(OApp(OCons(x, xs), ys, OCons(x, zs)) If

OApp(xs, ys, zs)).
```

Figure 4.13: The local part of module ONaive

the highest used variable index at each resolution step. It would therefore be expected to have a slightly worse performance than **SLD Demo**, since it differs from **SLD Demo** mainly in this extra pass. The elimination of intermediate variables is discussed further in Sections 4.4.2 and 4.5.

In the next section I consider strategies for improving the performance of ground interpreters, such as compiling away some redundant computation by using partial evaluation to specialise the interpreter to its object program, and by providing in-built support for the ground representation at a low-level in the implementation.

4.3 Optimising SLD Demo

4.3.1 Partial Evaluation

Although partial evaluation is not the subject of this thesis, it must be considered as an important candidate technique for the optimisation of metaprograms. Partial evaluation is a program specialisation technique. A partial evaluator is a tool which takes a program and some partial input data for that program, and produces a version of the program specialised for the input data. A recent overview of partial evaluation and its application to several classes of programming language is given by Jones et al [42]. The partial evaluation of logic programs by unfolding was put on a firm theoret-

ical footing by Lloyd and Shepherdson [53], and as a technical description of partial evaluation is outside the scope of this thesis, I refer to that paper for a formal definition of partial evaluation as it applies to logic programming.

Both SAGE, the Self-Applicable Gödel partial Evaluator, and the design and implementation of the Resolve predicate are the work of Gurr [32]; they are designed to work together. SAGE is a partial evaluator based mainly on finite unfolding. Above all, SAGE is intended to be an effectively self-applicable partial evaluator for a full logic programming language (as opposed to some restricted subset of the language). A self-applicable partial evaluator is one which is capable of specialising itself, and an effectively self-applicable partial evaluator is one which produces significantly improved, efficient residual code upon self-application. Resolve is carefully implemented to leave good residual code when specialised by SAGE.

SAGE performs the partial evaluation of a program with respect to a given goal in four main phases:

- 1. termination analysis;
- 2. partial evaluation;
- 3. optimisation of residual code;
- 4. replacement of original code by specialised code.

The first phase is a static analysis that identifies the predicates that can be unfolded without the risk of infinite unfolding. Those predicates that may unfold infinitely are treated with more caution by the partial evaluation phase.

After partial evaluation, SAGE performs an additional optimisation by removing redundant terms. These appear as input arguments in the head of a clause before specialisation, and are instantiated by the partial goal, but are no longer used in the body of the specialised clause, because the specialisation has removed all references to that particular argument.

For example, a meta-interpreter with top level predicate Demo is specialised to a particular object program by partially evaluating with respect to the partial goal <- Demo($\lceil P \rceil$, query,answer), where $\lceil P \rceil$ is the ground term representing an object program P. After partial evaluation, all predicates in the bodies of the clauses for Demo that access $\lceil P \rceil$

may have been specialised away, because $\lceil P \rceil$ is fully defined before partial evaluation. We may therefore delete this first argument from the specialised clauses for Demo. The deletion is achieved by replacing the ternary predicate Demo with a new binary predicate, Demo_1 say, where any computed answer for Demo_1 (query,answer) is equivalent to that computed for Demo ($\lceil P \rceil$, query,answer).

Note that the partial evaluation of an interpreter with respect to an object program requires the presence of two levels of representation. Let I be an interpreter, and P a program suitable for I to interpret. The partial evaluator itself is the meta-program, and it operates on a representation of the interpreter, $\lceil I \rceil$, which is its object program. A goal for I contains a representation $\lceil P \rceil$ of the interpreter's object program, and when this goal is represented in the partial evaluator, this representation is again represented, so the term $\lceil \lceil P \rceil \rceil$ appears in the representation of the goal given to the partial evaluator. Applications such as partial evaluation of meta-programs therefore render such multi-level representations necessary.

The clean design and implementation of Gödel meta-programming lets this happen in a straightforward manner. In practice, the goal for the interpreter is of the form Go("OM", query, answer) where the predicate Go is a harness for the interpreter like that in Figure 4.5, and "OM" is the name of the top-level module of the object program to be interpreted. Because this is a partial goal, query and answer are uninstantiated variables.

In unfolding Go, the partial evaluator arrives at the atom ProgramCompile("OM", p), which is exported by the closed system module ProgramsIO and so cannot be unfolded. It is, however, sufficiently instantiated to run, and can be executed using the system interpreter Compute (as in the example in Figure 4.9). The call of Compute reflects the representation of the ProgramCompile atom down to the object-level and executes it directly. The actual call of ProgramCompile("OM", p) returns the representation of the program with top-level module OM, and Compute then reflects back upwards, constructing the representation of this representation, finally returning an appropriate binding in the output substitution of the sixth argument.

The present implementation of Gödel constructs double representations such the term $\lceil \lceil P \rceil \rceil$ in a rather naïve way, by simply using the encoding previously described in Sections 3.1 to represent the term $\lceil P \rceil$. Note that

 $\lceil P \rceil$ is the representation of an object program, whereas $\lceil \lceil P \rceil \rceil$ is the representation of an object term. This means that, for example, all the binary tree nodes in $\lceil P \rceil$ are explicitly represented in $\lceil \lceil P \rceil \rceil$. Needless to say, such representations of representations can generate exceedingly large terms, and more compact encodings must be possible, although this has not been investigated.

4.3.2 Specialising SLD Demo

Figure 4.14 illustrates the result of specialising the Demo interpreter of Figure 4.8 with respect to the standard Append program. In this specialised version of the interpreter, three significant changes have been introduced.

- 1. The calls to Resolve have been specialised with respect to the two statements in the object program to produce the third and fourth statements respectively in the new predicate Demo_1.
- 2. Symbols (such as Empty and &'), which are normally hidden because Gödel implements the ground representation as an abstract data type, have been promoted into the specialised program.
- 3. The representation of the object program Append, which is now redundant, has been removed by replacing the predicate Demo/6 by the new predicate Demo_1/5.

Specialising SLD Demo with respect to a given object program yields very considerable performance improvements. The specialised programs typically run forty or more times faster than the original code. There are several sources of this improvement. Unfolding the predicates in Syntax that manipulate the abstract data type is an important one; this reduces the number of procedure calls and enables direct pattern matching on the constants and functions making up the abstract type. It is also significant that many of these predicates have control declarations which are expensive to evaluate at run-time. These control declarations are eliminated when the predicates they guard are unfolded. There is considerable redundant computation involved in repeatedly extracting the input statements from the Program term, where they are stored in a binary tree. For a recursive program such as naïve reverse, the same statement is used many times over;

```
Object program:
Append([],x,x).
Append([a|x],y,[a|z]) <- Append(x,y,z).
Specialised interpreter:
Demo_1(Empty, v, v, subst, subst).
Demo_1(left &' right, v_in, v_out, subst_in, subst_out) <-</pre>
   Demo_1(left,v_in,new_v,subst_in,new_subst) &
   Demo_1(right,new_v,v_out,new_subst,subst_out).
Demo_1(Atom(Append',[arg1,arg2,arg3]),v_in,v_out,
      subst_in,subst_out) <-</pre>
   GetConstant(arg1,CTerm(Nil'),subst_in,s1) &
   GetValue(arg2,arg3,s1,new_subst)
   Demo_1(Empty',v_in,v_out,new_subst,subst_out).
Demo_1(Atom(Append',[arg1,arg2,arg3]),v_in,v_out,
      subst_in,subst_out) <-</pre>
   GetFunction(arg1,Term(Cons',[sub11,sub12]),mode,
      subst_in,s1) &
   UnifyVariable(mode,sub11,v_in,v1) &
   UnifyVariable(mode,sub12,v1,v2) &
   GetFunction(arg3,Term(Cons',[sub21,sub22]),mode1,s1,s2) &
   UnifyValue(mode1,sub11,sub21,s2,new_subst) &
   UnifyVariable(mode1,sub22,v2,new_v) &
   Demo_1(Atom(Append',[sub12,arg2,sub22]),new_v,v_out,
      new_subst,subst_out).
```

Figure 4.14: Specialisation of Demo wrt Append

specialisation completely removes this overhead by promoting all the statements of the object program into the clauses of the specialised interpreter,
and eliminating the program term entirely. Finally, a specialised unification
procedure is generated for the head of each clause in the object program,
expressed in terms of predicates that remain from the unfolding of Resolve.
These predicates form part of the internal implementation of Resolve, and
in the examples in this chapter are normally determined to be unsafe to
unfold by the static analysis phase of SAGE. The names of these predicates
have been chosen (with one exception) to match the names of certain WAM
instructions, to which they bear a functional resemblance. They are therefore referred to collectively as the WAM-like predicates. Thus it can be seen
that the partial evaluation of interpreters is analogous to the compilation of
logic programs into WAM instructions.

Six of the seven WAM-like predicates are analogous to emulators for the WAM instructions **GetValue** (which corresponds to the WAM-like predicate UnifyTerms), **GetConstant**, **GetFunction**, **UnifyValue**, **UnifyVariable** and **UnifyConstant**. A subtle difference between the manner in which the WAM implements the unification of nested terms and the manner in which **Resolve** implements it means that the WAM (as originally defined by Warren [86]) does not have an equivalent for the seventh WAM-like predicate, which is **UnifyFunction**. More complete specifications of the WAM-like predicates can be found in [15] and [32].

4.3.3 An Additional Optimisation

Recall that substitutions are represented as uncomposed sets of bindings, and that, since looking up bindings in the substitution is a frequent occurrence during unification, the speed with which the binding for a particular variable can be located is an important factor in the efficiency of unification. For this reason, integral indices were added to the representation of variables, and the bindings were stored in a binary tree. Why not use a true array to store the bindings, to obtain constant rather than logarithmic access times? The difficulty is, of course, that arrays can only be updated destructively, and this destroys the soundness of the implementation if any references to the original array remain after a destructive update. However the update time is also constant, and fast compared with the typically logarithmic update time for a binary tree, because in updating a node of a

```
Demo(..., subst_in, subst_out, ...) <-
UnifyTerms(..., subst_in, s1) &
GetConstant(..., s1, s2) &
GetFunction(..., s2, s3) &
UnifyVariable(...) &
UnifyFunction(..., s3, s4) &
UnifyValue(..., s4, s5) &
UnifyConstant(..., s5, subst_out).</pre>
```

Figure 4.15: Single-threading of substitutions

binary tree a copy must be made of the branch of the tree from the root to the updated node, and this introduces a significant constant factor.

Figure 4.14 shows that a sequence of WAM-like predicates typically results from specialising the calls to Resolve in SLD Demo. Figure 4.15 has all arguments other than those of type TermSubst removed for clarity, and the pattern in which substitution arguments are passed between the WAM-like predicates can be seen. The sequence of WAM-like predicates is intended to be executed from left-to-right¹, and under this execution order, after each WAM-like predicate updates the substitution data structure, no live references to the original input structure will remain (with the possible exception of subst_in, which may be referred to from outside the Demo context). Data structures treated in this way at the implementation level are referred to as single-threaded, and can often be safely updated destructively.

Take the call of GetConstant(arg3,CTerm(Nil'),s1,s2) from Figure 4.14, as a relatively simple example. If arg3 is instantiated to the representation of a variable, this variable is dereferenced by following its binding chain in the input substitution s1. If the result of dereferencing arg3 is another variable, the binding of this variable to constant Nil is recorded by updating substitution s1 to give the output substitution s2. Since the substitution argument is single-threaded, if there are no other references to subst_in at the time Resolve is called, it will be safe to perform this update destructively, and similarly for updates made by all the other

¹If necessary, DELAY declarations can be added to enforce a particular execution order.

instructions in the sequence. It is assumed that all destructive updates are trailed where necessary, and therefore correctly undone on backtracking. By analysing the program this way by hand, it can be determined that it is safe to experiment with the use of arrays and destructive assignment for the representation of substitutions in both the **SLD Demo** interpreter and in its specialised form.

The SICStus Prolog ² built-in setarg/3 provides a destructive array update suitable for such an experiment. The SICStus Prolog manual [75] entry for setarg/3 states:

setarg(+ArgNo, +CompoundTerm, ?NewArg). Replace destructively argument ArgNo in CompoundTerm by NewArg. The assignment is undone on backtracking. This operation is only safe if there is no further use of the "old" value of the replaced argument. The use of this predicate is discouraged, as the idea of destructive replacement is alien to logic programming.

Thus setarg/3 allows compound terms to be used as arrays that can be updated destructively, and automatically handles the necessary trailing so that such updates are undone appropriately on backtracking. However, it must be noted that terms in SICStus Prolog have a maximum arity of 255, so a nested term structure is required to obtain an array sufficiently large for the experiment. The experiment must therefore be regarded as somewhat crude, and a special-purpose implementation would be certain to perform better.

4.3.4 Results

Figure 4.16 shows the results of some experimentation with the specialisation of interpreters, and the use of arrays to represent substitutions. The same naïve reverse benchmark was used as for the results in Figure 4.10.

The most significant result here is that in the last line of the table, which gives the relative execution time for **SLD Demo**, specialised by SAGE with respect to the naïve reverse object program, and with substitutions represented crudely by fixed size mutable arrays, large enough for this particular

²Actually SICStus Prolog 2.1. The setarg/3 built-in has been discontinued in more recent versions of SICStus Prolog

Program	Relative time for naïve reverse
Unify Demo	280000
Unify Demo, specialised	500000
SLD Demo	31000
SLD Demo, no control	1980
SLD Demo, specialised	612
SLD Demo, specialised, arrays	216

Figure 4.16: Effect of optimising interpreters

computation. It can be seen that the use of arrays to represent substitutions yields a reduction in execution time by almost a factor of three, and comparison with the table in Figure 4.10 shows that the performance has come within an order of magnitude of that of the vanilla interpreter. These timings are exclusive of time spent garbage collecting, so the improved performance is not simply due to a reduction in the number of garbage collections. Nevertheless, the use of a mutable data structure means that less copying of data structures is occurring and so less garbage is generated.

The table entry in Figure 4.16 marked "no control" indicates the speed up obtained by compiling the unspecialised **SLD Demo** with a special Gödel compiler that ignores all control declarations and does not enforce safe negation, so the resulting program simply executes from left to right. The run-time control in the Gödel system by itself costs a factor of 10 at least, and this overhead is not created directly by the use of the ground representation, but is rather a limitation of the present implementation. It may be possible to remove almost all of the control overhead by performing a static analysis of the program at compile time.

Part of the performance gain due to specialisation with SAGE is due to the removal of predicates that have control declarations. For example, Resolve, which has a DELAY declaration, is replaced by a sequence of the WAM-like predicates, which do not. It is therefore fair to compare the performance of the specialised interpreter with that of the plain interpreter compiled with no control. Compared to the execution time obtained without control, the partial evaluation gains a factor of 3 and the use of arrays another factor of 3, bringing the overall performance within a factor of 20 of the speed of the vanilla interpreter.

It is interesting to observe that specialising Unify Demo actually slows

it down by a significant amount. SAGE will not unfold the system predicate StandardiseFormula without knowing the initial free index to be used in the renaming, and this value is unknown during partial evaluation because it depends on the actual computation, not just on the object program, so the residual code contains clauses such as that shown in Figure 4.17, generated from the second clause of Append. The head of the UDemo 1 clause is matched whenever there is a representation of an atom with predicate symbol Append in the first argument, so the Cons and Nil cases are not immediately distinguished. UDemo_1 then builds a complete representation of the second clause of Append on the heap. In contrast, the unspecialised UDemo clause finds a pre-existing representation of a clause for Append in the Program term using the AVL-tree lookup mechanism. Both UDemo and UDemo_1 then rename this clause with a call of StandardiseFormula, before attempting to use UnifyAtoms to unify the goal atom with the the head of the renamed clause. In both cases the call of UnifyAtoms may fail, because the Cons and Nil cases for Append have not been distinguished prior to this point. The precise amount of computation wasted in this way depends on the order of the clauses in UDemo_1, and on the order in which StatementMatchAtom returns clauses for Append in UDemo. There is no reason to suppose that this is the same in both cases³. Building the representation of the clause on the heap before calling StandardiseFormula may also be more expensive than retrieving it from the Program term was in the unspecialised version. This example demonstrates that simple partial evaluation is not an effective optimisation technique for every source program; it works best on source programs that are specifically constructed to be specialised.

4.4 Matters Arising

The results of the previous section demonstrate that the use of mutable arrays to represent substitutions yields a valuable performance improvement. Two questions arise from this observation. Firstly, what techniques are available for the provision of mutable arrays with constant time access in logic programming languages? Secondly, how can the size of substitutions

³The order in which clauses are selected in a computation is undefined in Gödel.

```
UDemo_1(Atom(Append',v_23),v_17,v_2,v_20,v_4) <-
StandardiseFormula(
    Atom(Append',
        [Term(Cons',[Var("x",0),Var("xs",0)]),
        Var("ys",0),
        Term(Cons',[Var("x",0),Var("zs",0)])])
    <-'
    Atom(Append',
        [Var("xs",0),Var("ys",0),Var("zs",0)]),
    v_17,v_18,v_21 <-' v_9) &
UnifyAtoms(Atom(Append',v_23),
    v_21,v_20,v_22) &
UDemo_1(v_9,v_18,v_2,v_22,v_4).</pre>
```

Figure 4.17: Part of Unify Demo specialised to Append

be controlled? Answers to these questions are attempted in this section.

4.4.1 Mutable Data Structures in Gödel

If the use of arrays and destructive assignment is to be a practical tool, some way must be found to ensure that programs remain declarative. For **SLD Demo** and similar programs in which there is just one single-threaded substitution that is updated sequentially there is no difficulty, but the Gödel compiler must be able to make sure all programs are declarative, and do so automatically.

One possibility would be to use an abstract analysis aimed at deriving liveness information for possible structure reuse, or compile-time garbage collection, such as [60]. However, compile-time analysis presents some serious difficulties. Such an analysis is complex to implement and slow to execute, and normally the goal for the program must be supplied before the analysis can be completed. The technology to deal with languages like Gödel that do not execute purely left-to-right is not yet mature, and the flexible computation rule causes an inevitable loss of accuracy. If the accuracy of the analysis cannot be relied upon to discover most of the cases

when destructive updates are allowed, the technique will be of little benefit.

A purely syntactic restriction would be ideal; the informal argument given above for the safety of SLD Demo with mutable arrays hints that such a restriction is possible. In each clause of **SLD Demo**, and of the specialised version, there are no more than two occurrences of each variable of type TermSubst. This suggests something like the two-occurrence restriction of Janus [71]), where variables are limited to two occurrences in a clause, and one occurrence is labelled a producer, and the other a consumer. The two occurrence restriction in Janus is known to be undecidable [25], but this is because Janus has indexed arrays that can contain variables. The Gödel type system could be used to limit the restriction to variables of type TermSubst so that the impracticalities of Janus are largely avoided. If the two-occurrence restriction is violated because a clause with more than two occurrences of a TermSubst variable, the Gödel compiler can generate code to make a duplicate of the entire array at run-time, thus avoiding aliasing problems. While it may be a little heavy-handed, this approach is adequate for interpreters like **SLD Demo** and provides a starting point for developing a more discriminatory analysis.

It is worth noticing at this point that an elegant syntactic restriction that permits mutable data structures exists in higher-order functional languages, in the form of monads [59]. Monads guarantee that certain data structures are single-threaded, and are used in functional languages for declarative input-output, and to provide mutable variables and arrays [67]. Unfortunately, monadic programming requires the use of higher-order functions, and monads do not fit easily into relational languages.

A good solution is an efficient but declarative array implementation, such as the "hairy structures" of Barklund and Millroth [10], which provide mutable data structures with a constant time overhead, and constant time access to the most recent version of the data structure. When a hairy array is updated, the update is performed in-place as in a normal array, and an element containing the updated index and old value is added to an "exception list". Any references to the previous version of the array are made through this exception list, which allows the update to be undone if required. If there are no references to the previous version, the exception list element is garbage and will eventually be discarded. This idea seems to have been unjustly neglected in logic programming systems, perhaps because it can

become inefficient when several very different versions of the same array are in use simultaneously. It is well-suited to the representation of substitutions in interpreters such as **SLD Demo**, however. In addition, hash-tables can be constructed, and used to give faster access to the definitions and declarations within a program representation.

4.4.2 Garbage Collection

During the execution of **SLD Demo**, new variable representations are constantly being created, each new variable having an index which points to a location in the substitution array, where the variable may eventually be bound. The free index value increases as new variables are needed. The substitution array is passed as a parameter to the WAM-like predicates and updated during their execution. The analogy between representations of substitutions in the ground representation and the variable stack in the underlying Prolog system, and between the free variable index and stack pointer, is inescapable. They both implement the answer substitution being accumulated during an SLD computation, and since the stack in the underlying system is designed for maximum efficiency, a promising way to obtain efficiency in interpreters for the ground representation would be to have representations of substitutions emulate the system stack as closely as possible. When viewed in this way, the use of arrays to implement representations of substitutions seems a natural solution to the problem of simulating unification efficiently in the ground representation. It simply involves introducing an extra level of indirection (compared with direct access to ordinary logical variables at the meta-level) between an atom or term and the values to which its variables have been bound during a computation. The meta-program specifies which variable stack representation is to be used to dereference a variable representation, by explicitly mentioning the representation of a substitution as a parameter.

Because an arbitrary number of new variable representations may be generated during the execution of an interpreter like **SLD Demo**, fixed sized arrays are not adequate to represent substitutions in general. The substitution must be able to expand to accommodate more variables as required. This is analogous to the need to expand the variable stack occasionally in Prolog implementations such as SICStus, and this operation sometimes requires copying the stack into a larger region of memory. Sim-

ilarly, substitutions can be expanded by copying the contents into a larger array from time to time, leaving the original substitution to be garbage collected if there are no more references to it. Provided this is not done too frequently, the overhead of expanding substitutions should not be excessive. The Resolve predicate manages renaming of the input statement so that new variable indexes are allocated only for variables that appear in the body of the statement but not in the head; this keeps the number of new variable representations generated to a minimum.

Carrying the analogy between substitutions and stacks still further, interpreters that must be able to deal with large-scale object programs will need to perform a form of garbage collection on representations of substitutions. Garbage collection in this sense means to remove bindings for variables that no longer have a part to play in the representation, and in a compaction phase to perform an appropriate renaming on all occurrences of the remaining variables so that their indexes are small and consecutive. This would prevent the representations of substitutions from growing without limit.

One can envisage such a garbage collection function being provided by a predicate such as GarbageCollect below. In order to determine which variables are "alive", GarbageCollect needs to be given the free variables in the initial goal (these are never renamed) and the entire current goal. It is also given the substitution to be compacted, and returns the compacted substitution (or the original substitution if compaction was unnecessary) and the new free variable index.

```
PREDICATE GarbageCollect:

List(Term) % Free variables in initial goal.

* TermSubst % Substitution to be compacted.

* Formula % The current goal.

* TermSubst % Compacted substitution.

* Formula % Renamed goal.

* Integer. % New free variable index.

DELAY GarbageCollect(x,y,z,_,_,) UNTIL

GROUND(x) & GROUND(y) & GROUND(z).
```

Interpreters should call GarbageCollect immediately before Resolve at

each resolution step. GarbageCollect should apply some simple criteria to determine if a compaction is necessary, and if not, return immediately.

Since GarbageCollect needs the whole of the current goal in order to determine the live variables, it cannot be used with the SLD Demo style of interpreter of Figure 4.8. Instead, it requires an interpreter that keeps the current goal together and uses a Select predicate like that of Figure 4.4. Structuring an interpreter around Select has other advantages too: it makes the computation rule explicit, and extensions such as safe negation and coroutining can be handled as a consequence.

4.5 The Breadth-first Interpreter

I conclude this discussion of interpreters by presenting a Gödel interpreter in a different style. This is **Breadth First**, an interpreter for definite programs that searches an SLD-tree for a given object goal and object program breadth-first, rather than depth first as **SLD Demo** does.

There are several reasons for including this interpreter. Where **SLD Demo** is functionally similar to the vanilla interpreter for the non-ground representation, **Breadth First** provides genuine added value by providing a different computational model for the object program. It is impossible to program a truly declarative and equivalent breadth-first interpreter using the non-ground representation, although it is straightforward to write a non-declarative one in Prolog. Because it is declarative, **Breadth First** is amenable to partial evaluation, and can be specialised effectively by SAGE, improving the performance by a factor of around 3. The Prolog version is not easy to specialise, as the non-declarative aspect must be dealt with. **Breadth First** also demonstrates the flexibility of the **Resolve** predicate, which is useful for constructing many different kinds of interpreter. Finally, and most importantly, Breadth-first presents a pathological counterexample to the effectiveness of the single-threaded substitutions suggested in Section 4.3.3.

The main loop of **Breadth First** is shown in Figure 4.18. The function N is used to represent a node in the SLD-tree; each node contains a goal, the substitution computed on the branch of the tree up to this point, and a free variable index. The interpreter operates on a list of representations of notes that have yet to be expanded. It takes the first such node, selects

```
BASE Node.
FUNCTION N :
                    % A node in the computation.
   Formula
                    % The body of the goal at this node.
 * Integer
                    % Free variable index.
 * TermSubst
                    % The substitution at this node.
-> Node.
PREDICATE Breadth :
   List(Node)
                    % Unexpanded nodes.
 * Program
                    % The object program.
 * List(TermSubst). % The answer substitutions from
                    % successful nodes.
Breadth([], _, []).
Breadth([N(goal, _, subst)|nodes], program,
      [subst|answers]) <-
   EmptyFormula(goal) &
   Breadth(nodes, program, answers).
Breadth([N(goal, v, subst)|nodes], program, answers) <-</pre>
   Select(goal, left, sel, right) &
   PredicateAtom(sel, pred, _) &
  DefinitionInProgram(program, _, predicate, stats) &
   Children(stats, left, sel, right, v, subst, children) &
   Append(nodes, children, new_nodes) &
   Breadth(new_nodes, program, answers).
```

Figure 4.18: The **Breadth First** interpreter

```
% Find all the children of a node in an SLD-tree.
PREDICATE Children :
   List(Formula)
                  % The definition of a predicate.
 * Formula
                   % Selected atom with this predicate.
 * Formula
                   % Subgoal to the left of selected atom.
 * Formula
                   % Subgoal to the right of selected atom.
 * Integer
                   % Free variable index.
                   % The substitution at this node.
 * TermSubst
 * List(Node).
                   % The children of this node.
Children([], _, _, _, _, []).
Children([stat|stats], left, sel, right, v_in, subst,
      children) <-
   ( IF SOME [v_out, subst1, body]
        Resolve(sel, stat, v_in, v_out, subst, subst1, body)
     THEN
        AndWithEmpty(body, right, goal1) &
        AndWithEmpty(left, goal1, goal) &
        children = [N(goal, v_out, subst1)|children1]
     ELSE
        children = children1
   ) &
   Children(stats, left, sel, right, v_in, subst, children1).
```

Figure 4.19: Generating child nodes

an atom in the goal, finds the definition of the predicate symbol in this atom, expands the node through a call to Children, and appends the new child node representations to the back of the list. Note that different search strategies can be obtained by changing the way the new nodes are added to the list: adding them to the front would give a depth-first search, or the list could be sorted according to some heuristic to obtain best-first. The predicate Children (Figure 4.19) uses Resolve to create a child node for each statement in the definition whose head unifies with the selected atom.

Because the interpreter is constructed to store a substitution as part of its representation of each unexpanded node of the tree, the substitutions are not single threaded. Instead, when a node is expanded to produce n child nodes, the single substitution at the node is updated through n distinct unifications to produce n different substitutions, one for each child node, and all n substitutions remain active in the next loop of the computation. As expected, the **subst** variable in the second clause for **Children** has three occurrences in that clause.

If substitution representations are implemented by "hairy" arrays, many bindings in active substitutions will be pushed into the exception list, when the same variable is bound in another branch of the tree. The time taken to dereference a variable degrades as the exception lists grow. Alternatively, the entire substitution might be copied n-1 times to produce the n child nodes, but this is also slow and not space efficient.

The Breadth First interpreter can be formulated differently, in order to avoid sharing substitutions, by representing nodes with resultants instead of substitutions. Each call to Resolve then starts with an empty substitution, and the output substitution form Resolve is applied to the head of the resultant and discarded. This approach is promising, but has two difficulties. Firstly, the application of the substitution to the head of the resultant cannot be specialised. Secondly, either the resultant must be scanned at each step to find a free variable index, or the index of new variables is left to grow monotonically in each branch of the tree. In the latter case, empty substitutions will be required to accept new bindings for variables with arbitrarily high indexes. It is unclear which of these two approaches, either using non single-threaded substitutions or applying the substitution in each child node, will ultimately give the best performance; further research is needed to clarify this issue.

A comparison between **Breadth First** and **SLD Demo** illustrates the fact that the further an interpreter design gets from the computational mechanism used to execute the meta-program, the more difficult it is to make the interpreter efficient.

4.6 Summary

The lessons learned from the work in this chapter can be summarised as follows.

If the simulation of unification for ground representations of terms and atoms is to be efficient, representations of substitutions are required at some level in order to keep track of bindings for object-level variables. Naïve interpreters for the ground representation, such as those found in the literature [38, 11], have been based on composition of substitutions. Composing substitutions is potentially a very expensive operation.

To avoid composition, substitutions are represented in uncomposed form, and an input substitution argument is added to predicates that perform unification. Integer variable indexes simplify the process of standardisation apart, and speed up the search for variable bindings in the substitution structure, which is a tree rather than a list. This leads to a style of interpreter in which substitution representations and free variable indexes are threaded through the computation.

The Resolve predicate lowers the interpretation overhead still further, by combining standardisation apart and unification in a single operation. This also reduces the number of new variables introduced at each resolution step. Importantly, Resolve can be specialised effectively by partial evaluation, when the object program is known at partial evaluation time.

Interpreters designed for partial evaluation are improved significantly by specialisation, giving a performance gain of up to fifty times. Some of this performance gain is due to the removal of Gödel's dynamic mode checking. If the overhead of mode checking is ignored, specialisation improves performance by up to a factor of three. Poorly designed interpreters are unlikely to be specialised effectively.

The execution speed of the residual code from a specialised interpreter can be further improved. It was observed that the substitution representation is passed between the WAM-like predicates in a single-threaded manner, and the resemblance between substitution representations and the heap of the underlying system is unmistakable. Thus, a true mutable array structure is ideal for recording variable bindings, and a crude experiment suggests that a gain of as much as another factor of three can be obtained in this way. The resulting performance is still close to an order of magnitude slower than a non-ground vanilla interpreter, but there is no reason to suppose that the implementation is the most efficient one possible, and the added value of the ground representation compensates to some extent.

If interpreters constructed in the style suggested in this chapter are to scale adequately, a mechanism is required to keep the size of substitution representations under control. Variables that no longer play a role in the computation need to be eliminated. A garbage collection predicate can be provided for this purpose.

Not every interpreter is entirely amenable to the optimisation techniques proposed in this chapter. Breadth-first interpreters are particularly difficult to implement efficiently, because their computational mechanism is so far from that of the underlying system.

Chapter 5

Conclusions

5.1 An Evaluation of Gödel Meta-programming

In this section I give a brief critical evaluation of Gödel meta-programming and the work presented in this thesis, discuss some of the lessons learned, and suggest some likely directions for improvement.

5.1.1 Expressiveness

It is clear that the Gödel meta-programming library is expressive enough for a good many meta-programming tasks. The SAGE partial evaluator is written entirely in Gödel, and SAGE is effectively self-applicable. The library has also been used to implement other sophisticated meta-programs, such as various interpreters, a model elimination theorem prover [24], and knowledge assimilation algorithms such as the example in Section 2.2.4, which involves dynamic meta-programming.

The first drawback of meta-programming in the ground representation is the scale of the programming effort required to implement even a simple interpreter, and the complexity of the resulting program. The approach to meta-programming based on library modules and abstract data types overcomes this drawback satisfactorily. The abstract types enable the programmer to ignore the details of the ground representation, and treat even complex structures such as representations of object programs as simple values.

Interpreters such as **SLD Demo** (Figure 4.8) are not significantly more complex than the equivalent vanilla interpreter for the non-ground represen-

tation, but are considerably more flexible because of the increased expressiveness of the ground representation. Gödel gives a fine degree of control over the object program right down to the level of variable and symbol names. Once the Gödel style of meta-programming has become familiar, Gödel meta-programs are seen to be much easier to understand than non-declarative Prolog meta-programs. Gödel is thus eminently suitable as a language for teaching meta-programming: in my experience, students are often confused by the non-ground representation and the non-declarative features of Prolog. In addition, because Gödel meta-programs are declarative, they are relatively easy to specialise using simple partial evaluation techniques.

Of course, I do not claim that Gödel is the ultimate in expressive metaprogramming; it is unlikely that any general-purpose scheme could be powerful enough to perform well in all applications. Barklund and Hamfelt [4]
give a brief report on an attempt to use Gödel for the representation of
legal knowledge. The legislation they consider is hierarchical, consisting of
schemata and meta-theories that determine how specific legislation can be
derived from the schemata by the application of legal inferences such as
analogia legis. Gödel turned out to be unsuitable for this application, although it is unclear whether for reasons of efficiency or expressiveness; part
of the difficulty may have arisen because the Gödel implementation was in
an early stage at the time. The lack of cyclic dependencies between modules may also have been a factor. In mitigation, it must be said that the
representation of legal knowledge is a very sophisticated application of metaprogramming, and it is perhaps not too surprising that languages designed
for the purpose, such as Alloy [9], are more successful.

Further issues concerning the expressiveness of Gödel as a language for meta-programming are explored under some of the headings below in this section.

5.1.2 Efficiency

Lack of computational efficiency is a major obstacle to effective use of the ground representation, and probably the main reason why no language prior to Gödel has offered support for it on a large scale. In Chapter 4, an analysis of the sources of this overhead in interpreters was conducted. Much of the overhead comes from the management of representations of variables.

Naïve interpreters, which perform explicit renaming, unification and composition of substitutions, are extremely slow. More efficient interpreters can be obtained by representing substitutions in uncomposed form, and combining unification and standardisation apart into a single operation via the Resolve predicate.

Partial evaluation is generally hoped to hold the key to the removal of interpretation overhead, and in many circumstances it does so very well. SAGE gives good speedups, up to fifty times, for interpreters like SLD Demo that use the resolve predicate, which was designed to be specialised. Much of this speed up is due to the removal of control declarations, which can also be compiled away in other ways, for example by abstract interpretation, or a system of precise mode specifications as in the Mercury language [89]. Compared with the same program executed without control, the speed up from partial evaluation is about three times, which is still excellent. With an implementation of substitutions based on mutable arrays, an improvement by another factor of three is obtained. Thus specialised and optimised, the performance of SLD Demo is around ten times slower that that of nonground interpreters. This is remarkably good, but the residual code is still unacceptably slow for many purposes.

These benchmarks must not be taken as the last word on the efficiency of interpreters for the ground representation: the implementation is almost certainly not the most efficient one possible. Performance analysis in systems as complex as a Gödel interpreter running on a Prolog system is very difficult, because there are many factors involved. The WAM-like predicates still perform some mode checking, because they make use of the IF-THEN-ELSE construct; the implementation of control in Gödel can almost certainly be improved, and it is not known whether the SICStus Prolog implementation of freeze can be optimised (it is not a widely used feature). The experimental implementation of substitutions as arrays using setarg/3 is somewhat crude, because terms in SICStus Prolog are limited to 255 arguments, and setarg/3 trails every update so that it can be undone on backtracking. There is good reason to suppose that an optimised low-level implementation of WAM-like predicates would give a further performance improvement. A more compact representation of object-level terms, atoms and formulas is also likely to be beneficial; I will return to this point later.

It is possible to overstate the problem of interpretation overhead. Inter-

preters and other meta-programs obviously give added value over the direct execution of the object program, or they would not be considered an interesting programming technique. Some performance penalty is acceptable in exchange for this added value, although obviously this should be kept to a minimum.

I regard the management of object-level variables as the most crucial efficiency problem of the ground representation, and so have concentrated on that aspect in this work. Other efficiency issues are addressed briefly under the different headings below.

5.1.3 Reflective Interpreters

The performance of reflective interpreters was not explored to any great extent in this work. The main issue is the combination of dynamic metaprogramming with reflective predicates. It is obviously undesirable for a reflective interpreter to recompile the entire object program every time it is invoked, so non-destructive incremental compilation is required. The problem of non-destructive incremental compilation is that links between calls and the procedure definitions they invoke cannot be made destructively. One solution is for the system to keep track of the differences between each object program representation and the state of the compiled code, so that a minimal set of changes can be applied to the compiled code when the interpreter is invoked. Alternatively, a lightweight interpretation mechanism could handle linking by indirection through the Program term. There is a troublesome trade-off between the cost of compilation, which determines the cost of updating a program representation, and the speed of execution of a reflective interpreter. Meta-programs that generate many different object programs but rarely perform heavy reflective computations will suffer if updates are expensive; on the other hand, meta-programs that need to obtain results from complex reflective computations will suffer if compilation does not produce efficient code. There is no satisfactory balance for all applications. Of course, in Prolog, the implementation of assert/1 and retract/1 must also take this trade-off into account.

If the object language has a module system, modules can assist with the efficient implementation of reflective interpreters, by dividing the object program into a set of fixed modules, whose contents cannot be updated, and a set of mutable modules. The fixed modules are compiled once and for all, and the performance degradation from interpretation of the predicates defined in mutable modules then becomes less significant. CLOSED modules serve this purpose in Gödel. A similar approach is taken in SICStus Prolog, where dynamic predicates are interpreted.

5.1.4 Representation

The representation of object-level terms and formulas in Gödel is bulky. The choice of representation stems from the requirement that the logic of the Syntax module be made explicit for partial evaluation; thus, terms belonging to the abstract types Term and Formula are valid Gödel terms. Because of Gödel's strong typing, and because any object language can be encoded in the ground representation, the representation is based on generic data types such as lists and strings.

Flat names also contribute to the size of the representation, because their four components are explicit in terms of type Name. The ability to access these components easily is valuable for the efficient look-up of names in Program terms; the structure of the program representation was designed with this in mind. This precludes the representation of flat names by single strings. However, one might reasonably expect that the representation of an object-level symbol be no bulkier than a meta-level constant.

Undoubtedly, some of the remaining interpretation overhead in the specialised version of **SLD Demo** is a consequence of this bulky representation. Meta-level unification with the head of a clause for <code>Demo_1</code> in Figure 4.14 involves a relatively large amount of computation, because of the depth of nesting of the <code>Atom</code> term. Recall that, in the figure, <code>Append'</code> stands for a name term like <code>Name("M", "Append", Predicate, 3)</code>, and that strings are implemented as Prolog atoms. Because <code>name</code> terms are compound, equality between representations of symbols is not merely a comparison between constants. Thus further performance gains may be obtainable through the use of a more compact representation, as discussed further in Section 5.2.1.

The fact that a term of type Program is a valid Gödel term has the consequence that multiple levels of representation can be achieved without special support in Gödel, by simply representing the representation. Of course, this has the severe disadvantage that the resulting term of type Term is huge. It is difficult to see what can be done about this problem. One possible solution is to exploit the opacity mechanism: terms of type Program

are opaque, so they could be given a special representation. However, it can be argued that Program terms are not fundamentally different from any other large term. A more compact representation for all terms therefore looks to be the best way to reduce the representation size explosion.

I make one final point concerning the representation. Gödel has a minor design flaw in its representation of object-level variables. A variable with a name such as v_1 has three different representations: Var(1) and Var("v", 1) and Var("v_1", 1); this can cause difficulties for metaprograms. In addition, the possible presence of the two-argument form of variable representation in representations of formulas complicates the representation of substitutions. In retrospect, it would have been better to restrict all variable representations to the form with an integer index only. Variable names appearing in the program source need not be lost; they can be preserved by the parser, which might provide a mapping between indexes and original variable names as part of its output when parsing a formula.

5.1.5 Type Checking

It is usual for library procedures to have to check the integrity of their arguments, or behave in an undefined way if given unexpected data. In imperative languages, the second choice is often made, both for efficiency reasons and because it is not possible to check the integrity of many data structures unless designed for the purpose. The philosophy of declarative languages, on the other hand, demands well-defined semantics. Without extensive integrity checking, the precise meaning of complex library functions or predicates becomes heavily dependent on the particular implementation, and too complex to be described intelligibly.

In Gödel's ground representation, a Program term always represents a syntactically valid Gödel program. That is, the object program always satisfies the module conditions and is type correct. It is therefore easy to explain the intended meaning of a predicate such as Succeed, because it is known that the object-level computation cannot go wrong. What's more, Succeed does not have to type-check the object program before compiling the modules that need recompiling. In order to maintain the integrity of the Program term, all predicates that update the object program by adding or removing statements or declarations first ensure that the update yields a valid program. Thus, a statement can be added only if it is well-typed and

satisfies the head condition, a declaration can be deleted only if no other declaration or statement in the program makes use of the declared symbol, a module can be deleted only if it is empty, and so on.

There are two undesirable consequences of this strategy. The first is a loss of efficiency due to redundant type checking. A meta-program that synthesises or updates an object program must be designed to generate type-correct statements from the start, or it will fail unexpectedly when it tries to insert a statement that is not type-correct. Nevertheless, these type-correct statements will be type-checked again on insertion. If there is an application in which a meta-program needs to determine whether or not a statement is type correct, it makes most sense for it to use the predicate FormulaInModule provided for this purpose, rather than relying on the failure of the update, so again the check on insertion will be redundant. The second undesirable consequence is a loss of flexibility: no representation is available for *invalid* programs, and there are applications that start with an invalid program, and create a valid one from it. An example of such an application is a type inference system, which might start with a Gödel program that lacks declarations for some predicates, infer types for those predicates, and insert the resulting declarations. There is no convenient way to construct a type inference system like this in Gödel.

It might be supposed that some redundant type-checking can be removed by an analysis of the meta-program, but this is unlikely to be feasible in general. It also does not solve the problem of flexibility. A better answer would be to relax the integrity restriction on the Program term and provide a predicate to test its validity. Instead of requiring that Succeed check its Program argument, it is preferable to define the meaning of Succeed in terms of the semantics of a free-form language that underlies Gödel, and behaves in a defined way whether or not the object program is a valid Gödel program.

5.1.6 Modules

Modules are an essential programming language feature, and so must figure in object program representations. They convey the important benefit of allowing the object program to be divided into open and closed sections. Yet modules, and the associated requirement to flatten symbol names, are also an inconvenience to meta-programs. The problem is not unique to

Gödel; Prolog's module system also complicates Prolog meta-programming.

One difficulty is that the specialised statements resulting from partial evaluation of an object program may not be valid in the language of any one module of that program. This necessitated the introduction of the Gödel Scripts module; terms of type Script are representations of flattened and module-free Gödel object programs.

The presence of module names as a component of flat names creates another disadvantage, which is that Name terms are not generally portable between programs, even though it may be intuitive that two symbols declared in different programs with the same declared name may be related in some way.

The paper by Brogi and Contiero [18] describes an attempt to use Gödel interpreters to implement a set of composition operators for definite programs. Both non-ground and ground interpreters were tried. A key idea of program composition is that it allows the definition of a predicate to be split between several programs, which are then composed. Brogi and Contiero found that the use of flat names in the ground representation in Gödel created difficulties for this aspect of their work, because functions and predicates with the same declared name, but declared in two or more programs in modules with distinct names, were considered different predicates by the ground representation. They were able to work around this difficulty, somewhat artificially, by making sure that each program to be composed consisted of a single module, always with the same name. These authors made several positive comments about Gödel meta-programming. They found the ground representation was more suitable than non-ground for the implementation of composition operators, because of its greater expressive power. They also reported some success in using SAGE to improve the performance of their ground interpreters, obtaining speedups of around fifty percent.

Determining what significance a symbol declared in one object program should have in another is obviously problematic, but an answer to the problem of portability of names would be useful. If the symbols declared in one distinguished module only, say the top-level module of a program, were known by their declared names rather than their flat names, the uniqueness property of symbol names within programs would still be satisfied. Such symbols could then be used in the context of several different programs. Al-

though this is not a complete solution, it might at least solve the problems reported by Brogi and Contiero.

5.1.7 Partial Evaluation

It is difficult to imagine a more complete specialisation than that achieved by SAGE for **SLD Demo**; the results of partial evaluation can be spectacular. However, I do not believe that partial evaluation is a panacea for the removal of interpretation overhead.

Partial evaluation is not very useful for one-off applications of metaprograms, because it is expensive to perform and the time taken to specialise the meta-program may well exceed the time required to execute the unspecialised program. There is often insufficient partial information available for an effective specialisation, particularly when the object program is modified by the meta-program. In these cases, substantial residual code will remain. Effective meta-programming is possible only if this residual code executes efficiently. The opportunity to use partial evaluation does not therefore free us from the need to optimise the unspecialised portion of the program.

A paper by Leuschel and Martens [47] shows that meta-programming with explicit substitutions prevents partial evaluators from propagating partial information through meta-predicates like Gödel's UnifyTerms, that simulate unification of object-level terms or atoms. This seems to be a limitation inherent in partial evaluation as a technique, rather than an argument against the use of explicit representations of substitutions: the same problem occurs with programs other than meta-programs. The specialisation of interpreters by SAGE does not require this form of information propagation, provided the object-level goal is completely unknown at partial-evaluation time. The requirement largely comes from the application domain considered by Leuschel and Martens, which is integrity checking. The authors offer a substitution-free style of meta-programming for integrity checking, and show that meta-programs in this style can be specialised effectively. Indeed, the paper demonstrates in spectacular fashion that a meta-program using the ground representation can be specialised more than an equivalent non-ground and non-declarative meta-program, leaving much faster residual code.

The work of Leuschel and Martens is confirmation that, for effective specialisation, application programs must be designed to be specialised. The author of a program to be specialised needs to keep in mind which data structures will be known at partial evaluation time, and consider how this partial information will be propagated through the program by the partial evaluator. The nature of the application also plays a part in determining which low-level features will specialise well.

5.2 Future Work

5.2.1 Improving Gödel

Perhaps the most interesting future work to be done with Gödel metaprogramming would be to see how far the ideas presented in this thesis could be pushed, in order to make meta-programming in the ground representation as efficient as possible.

As remarked in the previous section, there is much to be gained from a more compact representation for terms and formulas, especially one that avoids using lists for the arguments of predicates and functions. A more compact representation can be obtained by introducing two infinite families of functions Term/1, Term/2, Term/3, ... of type Term and Atom/1, Atom/2, Atom/3, ... of type Formula. Functions in both families take a term of type Name as the first argument, and succeeding arguments are terms of type Term. Of course, infinite families of functions cannot have explicit declarations in Gödel, but Gödel already has infinite types such as Integer which share this feature. A consequence of the compact representation is that predicates like FunctionTerm in Syntax cannot be given an explicit definition in a finite number of Gödel statements; they must be implemented at a lower level.

It is also desirable to replace the compound Name terms by an infinite family of constants of type Name, so that equality between symbol names is optimally efficient. Access to the components of flat names will still be required. The Program term could contain a table mapping between Name constants and their flat name components. Alternatively, there may be advantages to storing such a mapping in a global table, like the atom table in a Prolog system; the presence of a program representation would not then be required to create or decode a name.

This compact scheme does not preclude partial evaluation, since specialised statements into which explicit term and formula representations have been promoted will still be valid formulas in the language of the meta-

program, if not in the language of any one module. Special provision has already been made for representing specialised programs in the form of the Script type and its compiler.

It would be worthwhile trying to give selected predicates in Syntax and Programs an optimised low-level implementation. These include at least the WAM-like predicates, StatementMatchAtom, and Resolve. Optimised implementations of these predicates would make use of declarative arrays, both for representing substitutions and to replace the balanced tree structures in the Program term with hash tables. Arrays with exception lists [10] seem promising for this application. Recall that Resolve is designed to be specialised effectively, rather than to execute efficiently. For such predicates, it would help to provide two implementations, one optimised, and one written in explicit logic designed for partial evaluation.

A comparison between the work done by a vanilla interpreter (such as that in Figure 4.11 and by the unspecialised SLD Demo 4.8 with all the optimisations above is interesting. The clauses that terminate the computation when the object-level goal has an empty body, and the clauses that handle conjunctive goals are very similar in each case. The vanilla interpreter can pattern match, where SLD Demo cannot because of the abstract data type. SLD Demo also has additional arguments. Some overhead originates from these sources. The major difference between the two interpreters is in the clause that handles an atomic goal body. The call to Statement in the vanilla interpreter presumably uses first argument indexing to locate an object-level statement with a matching predicate in the head. The call to StatementMatchAtom can similarly use a hashing technique to locate a statement, albeit going through two levels of nested hash tables in order to locate the appropriate module first. Statement then performs head unification, and constructs a copy of the instantiated body of the input clause on the heap. This is the function of Resolve in SLD Demo. Of course, Statement is compiled¹, whereas Resolve must interpret the input clause; nevertheless, the operations performed by Resolve are similar to those performed by the code of Statement, given an array-based representation of substitutions. Additional overhead in Resolve comes from array indexing and the generation of exception list elements. Leaving aside the overhead

¹In Prolog meta-programming, the predicate clause/1 is often used in place of Statement to access dynamic clauses, which are not compiled.

of checking control declarations, there seems no reason to suppose that an optimised implementation could not improve significantly on the present performance of **SLD Demo**, which is already within two orders of magnitude of the performance of the vanilla interpreter.

Unfortunately, it has not yet been possible to try out the optimisations proposed above. Gödel is a complex language, and its implementation requires a very full-featured logic programming system, such as SICStus Prolog. There is no system currently available that would support Gödel and permit sufficient low-level access for the necessary modifications to be made, and resources were not available to build one.

Other directions for future work might aim at ways to enhance the expressiveness of Gödel meta-programming, along the lines suggested in Section 5.1. It is perhaps more appropriate to consider the declarative languages that will succeed Gödel, and that is the topic of the next section.

5.2.2 Meta-programming in Escher

Many of the ideas in this thesis can be extended to other declarative languages, particularly languages with the modern features of types and modules. Declarative languages of the future will surely have support for powerful declarative meta-programming; as has been argued, meta-programming is one of the great strengths of the declarative formalism, and drives the choice of a declarative language for many applications. It is likely that the future of declarative programming rests with languages that integrate functional and logic (relational) programming [51]. Curry [34] is an example of such a language; Escher [52] is another. In this section, I speculate briefly about the design of meta-programming facilities for Escher.

The Escher language is designed by Lloyd. It is based on the syntax of Haskell [66], and extends Haskell syntax slightly through the addition of universal and existential quantifiers, which permit logical variables to appear in expressions. Escher inherits the higher-order features and lazy procedural semantics of Haskell, and also subsumes Gödel, in the sense that any Gödel program can be translated fairly directly into an equivalent Escher program. The underlying logic of Escher is based on Church's simple theory of types [21]. The Escher language is still under development, and efficient implementation techniques are the subject of ongoing research. It is not possible to give a full description of Escher here; the interested reader is

referred to [52]. The ideas concerning Escher meta-programming presented in this section are mine alone; the language designer bears no responsibility for any miscomprehensions.

An example Escher program is shown in Figure 5.1. This program implements a relation sort, using the merge sort algorithm. The sort relation is higher-order, and expects a predicate as its first argument; this latter predicate should be true when its first two arguments are sorted lists, and its third argument is the sorted list obtained by merging the lists in the first two arguments. The predicate intMerge satisfies this requirement for lists of integers. The functions firsthalf and secondhalf which divide a list are written in a functional style. This program is not meant to be an efficient implementation of the merge sort algorithm, but demonstrates the main features of Escher: the mixture of functional and relational programming, and the use of higher-order functions and predicates. A query for the MergeSort program might be sort intMerge [2,3,1] xs.

If W is an expression of type Bool, Escher syntax permits existential quantifiers in the form exists $\v_1 \ldots v_n W$, which stands for $\exists v_1 \ldots \exists v_n W$, and universal quantifiers in the form forall $\v_1 \ldots v_n W$, which stands for $\forall v_1 \ldots \forall v_n W$. Boolean functions for all the standard connectives, &&, ||, not, and ==> (implication) are provided.

The procedural model of Escher is based on rewriting. In addition to the standard rewrite rules of functional languages, such as β -reduction, Escher has a comprehensive set of rewrite rules for the boolean connectives and quantifiers, in the form of equations. An example of one of the most important rules is shown in Figure 5.2; this rule is responsible for propagating bindings for existentially quantified variables within a conjunction.

An Escher expression is rewritten according to a leftmost-outermost reduction strategy, until no further rewrites are applicable. The expression is then said to be in *normal form*.

In some respects, Escher is more suitable than Haskell as an object language. The procedural model permits computations with incomplete programs, and because Escher subsumes Gödel, Gödel-style interpreters for relational object programs can be expressed. Here though, I am more concerned with a native meta-programming style for Escher, and it is interesting to see how some of the experience of Gödel meta-programming might be applied to meta-programming in Escher. Much of what appears below could

```
module MergeSort where
import List(head, tail, length, take, drop)
sort :: ([a] -> [a] -> [a] -> Bool) -> [a] -> [a] -> Bool
sort merge xs ys =
   if length xs < 2 then ys == xs
   else exists \us vs ->
      sort merge (firsthalf xs) us &&
      sort merge (secondhalf xs) vs &&
      merge us vs ys
intMerge :: [Int] -> [Int] -> Bool
intMerge [] ys zs = zs == ys
intMerge xs [] zs = zs == xs
intMerge (x:xs) (y:ys) zs =
   if (x > y) then
      exists \us -> intMerge (x:xs) ys us && zs == y:us
   else
      exists \vs -> intMerge xs (y:ys) vs && zs == x:vs
firsthalf, secondhalf :: [a] -> [a]
firsthalf xs = take (length xs 'div' 2) xs
secondhalf xs = drop (length xs 'div' 2) xs
```

Figure 5.1: An Escher program

```
exists \x1 ... xn -> x && (xi == u) && y =
    exists \x1 ... xi-1 xi+1 ... xn -> x{xi/u} && y{xi/u}

-- where xi is not free in u; u is free for xi in x and y;
-- and x or y (or both) may be absent.
-- If n=1, then the RHS is x{x1/u} && y{x1/u}.
-- If both x and y are absent, then the RHS is True.
```

Figure 5.2: An Escher rewrite rule

equally well form part of a meta-programming library for Haskell.

Escher expressions consist of variables, constants, function applications, lambda-abstractions, case statements, quantified formulas, and let-expressions, which introduce local definitions. Haskell permits higher-level constructs, such as list comprehensions, but all such constructs are syntactic sugar for expressions in this basic set. There are essentially only two categories of declared symbol: type constructors, of which base types are a special case, and functions, of which constants are a special case. This is much simpler than Gödel, which has six categories and treats the logical connectives as special symbols; in Escher the connectives are just functions. Thus the multiplicity of syntactic entities encountered in Gödel metaprogramming is much reduced in Escher meta-programming.

It is convenient to use a single abstract data type Expression to represent Escher expressions. This type should have a compact implementation, especially for function applications, as described in Section 5.2.1.

As in Gödel, a Syntax library is provided for the manipulation of elements of the Expression type. If a functional style, rather than a relational style, is adopted for Escher meta-programming the functions in Syntax come in groups of three: a boolean test, and constructor and destructor operations. For example, variable representations are distinguished by integer indexes only, so there is a group of functions

```
isVar :: Expression -> Bool
var :: Int -> Expression
varIndex :: Expression -> Int
```

for testing if an expression is a variable, constructing a variable, and extracting the index from a variable.

As argued in [69], it is very convenient to be able to mix types and expressions when compiling a functional language, in order to annotate a subexpression with its type. Types can also usefully contain λ -abstractions and universal quantifiers. The Expression type therefore also serves as a representation of types in Escher.

The simplest Escher expression is a symbol. For maximum flexibility, a symbol representation can be constructed from any string using the function

```
sym :: String -> Expression
```

and the string name extracted from a symbol representation using the function

```
name :: Expression -> String
```

The intention is that sym stores symbol names in a centralised global table, so that symbol representations occupy a single heap cell and equality between symbol representations is fast. Thus, the functions sym and name are analogous to the two modes of the Prolog predicate name/2. The function

```
apply :: Expression -> [Expression] -> Expression
```

constructs function applications. The representation of an Escher expression f A x (where x is a variable) is then easily created by evaluating

```
apply (sym "f") [sym "A", var 1]
```

The Escher Syntax module exports a complete set of tests, constructors and destructors for every kind of Escher expression.

Leaving the Escher module system aside for a moment, an Escher program is a set of declarations and equations. The Equation type represents equations, and has the following constructor and destructors.

```
equals :: Expression -> Expression -> Equation
```

lhs :: Equation -> Expression
rhs :: Equation -> Expression

A Programs module and Program type are provided for the representation of Escher programs. Of course, owing to the generality of the Expression

type, it is possible to construct representations of type Equation that do not represent syntactically valid Escher equations. The facilities of the Escher Programs module determine which symbols, expressions and equations are valid in an object program. The Program type also includes a representation of the Escher module system; the details of this have yet to be worked out. Ideally, the restriction that an instance of the Program type always represents a type-correct Escher program is dropped; instead, the semantics of any reflective interpreters provided by Programs is defined in terms of an underlying untyped λ -calculus.

Because of it is higher-order, Escher provides the opportunity to try monads [59, 67] for meta-programming tasks that benefit from single-threaded data-structures. A monad consists of a type constructor, M say, together with the three functions below².

```
(>>=) :: M a -> (a -> M b) -> M b
(>>) :: M a -> M b -> M b
return :: a -> M a
```

Informally, the instances of the type M a can be thought of as computations or operations that return a result of type a. The function >>= is a sequencing operator: it passes the result of the first operation as an argument to the second. The >> function is similar, but used when the second operation does not require the result of the first. The function return is the trivial operation that simply returns its argument.

As an example of the way monads might be used for meta-programming, I define a simple reflective interpreter for Escher in the form of a monad Computation a. An instance of the type Computation a is an operation that takes a representation of an Escher program and a representation of an Escher expression, and performs some operation such as evaluating the expression with respect to the program. Practically, Computation a is an abstract data type, but its type can be imagined to be defined as follows.

```
type CState = (Program, Expression)
type Computation a = CState -> (a, CState)
```

In this example, the Computation monad has the following three operations:

²These functions must satisfy certain algebraic laws [66].

```
go :: Program -> Expression -> Expression
go prog exp =
   compute prog exp (runComputation >> getExpression)
```

Figure 5.3: A trivial monadic interpreter

```
runComputation :: Computation ()
stepComputation :: Integer -> Computation ()
getExpression :: Computation Expression
```

The operation runComputation evaluates the expression completely; the stepComputation operation performs n reduction steps, where n is the value of its argument, and getExpression returns the current expression.

The function compute below takes the representation of a program and the representation of an expression, reflects them down to the object level (compiling the program), executes a sequence of operations, and returns the result.

```
compute :: Program -> Expression -> Computation a -> a
```

The simplest reflective interpreter that can be defined using the computation monad is that of Figure 5.3; it just evaluates the expression and returns the result.

More advanced interpreters can be defined, because the monad permits interaction with the object-level computation. To define a depth-bounded interpreter, an operation

```
fail :: Computation a
```

is needed. The fail operation represents a failed computation; it is a zero for the Computation monad, satisfying the laws

```
m >> fail = fail
fail >>= m = fail
```

for any m of type Computation a.

The function try in Figure 5.4 defines a depth-bound interpreter. In this program, the predicate good is intended to be true when the expression satisfies some criterion for terminating the computation.

```
try :: Program -> Expression -> Expression
try prog exp = compute prog exp (tryComputation 1000)

tryComputation :: Integer -> Computation Expression
tryComputation n =
   if n <= 0
   then fail
   else
      stepComputation 1 >>
      getExpression >>=
      \exp -> if (good exp) then
      return exp
      else tryComputation (n-1)
```

Figure 5.4: A monadic depth-bounded interpreter

In this example, the monad is required to guarantee that computations are single-threaded, because in performing a computation step at the object level destructively changes the expression.

The Computation monad is only meant to demonstrate the idea that monads might be useful for meta-programming. More complex monads can be devised, for example to enable dynamic meta-programming. To my knowledge, the application of monads to meta-programming has not been explored to any great extent by the functional programming community. On the other hand, the enforced sequencing of operations imposed by monadic programming is very restrictive, and it remains to be seen if this style of meta-programming can be made sufficiently expressive.

Higher-order languages present many opportunities for expressive metaprogramming. For example, it is difficult to use the meta-programming facilities of Gödel to extend the language through the addition of new syntactic constructs, such as meta-annotations to guide a parallel implementation. In a higher-order language, it is possible to provide an extensible parser, which takes a user-defined parsing function as an argument, and invokes it when no other rules are applicable. Meta-programming in Escher is potentially both simpler and more expressive than meta-programming in Gödel.

5.3 Contributions

The contributions of this thesis are:

- 1. the design of a comprehensive library for meta-programming in Gödel, comprising the export parts of system modules Syntax and Programs;
- 2. the design of a naming scheme for the representation of Gödel object programs;
- 3. an implementation of the Gödel meta-programming facilities;
- 4. an analysis of the sources of excessive overhead in common interpreter designs, and the discovery of an idiom for the construction of efficient interpreters;
- 5. an investigation of the efficacy of partial evaluation for improving the efficiency of interpreters;
- 6. a proposal for further improving the efficiency of partially evaluated interpreters;
- 7. the introduction of SLDQE-resolution, a novel computational model for the evaluation of universally quantified implication formulas.

Contribution 1 was made in collaboration with Hill and Lloyd; their design of a meta-programming library for Gödel evolved significantly as a result of the work presented in this thesis. The final form of this library is given in Appendix B, which contains a listing of the export parts of the Syntax and Programs modules.

Contributions 2 and 3 culminated in a stable and substantially complete implementation of the Gödel programming language, which has been used by independent researchers for investigations in meta-programming.

Contributions 4, 5 and 6 were made in collaboration with Gurr, who designed and implemented the *SAGE* partial evaluator. This work was published in [15]. The central idea is that efficient interpreters are best constructed around threaded substitutions, and that the representation of substitutions is the key to the efficiency of interpreters. Partial evaluation

is effective at removing a great deal of interpretation overhead, but the efficiency of the residual code depends to a large extent on the cost of recording bindings for object variables.

Contribution 7 was made in collaboration with Hill. SLDQE-resolution is able to evaluate universally quantified implication formulas that would flounder when implemented using the Lloyd-Topor transformations, and so adds to the power and expressiveness of Gödel. SLDQE-resolution subsumes SLDNF-resolution, and has a soundness theorem. This work was published in [16].

No programming language prior to Gödel has attempted to provide support for meta-programming in the ground representation on the scale achieved in this work. Gödel's support for meta-programming is powerful enough for the construction of sophisticated meta-programs such as SAGE, and it is fully declarative.

5.4 Concluding Remarks

I claim that the first goal of this research, which was

To develop a comprehensive library of predicates for the manipulation of Gödel object programs, sufficiently expressive for a wide variety of meta-programming tasks.

has substantially been met. Gödel's meta-programming library is certainly applicable to a wide variety of meta-programming tasks, and conquers the objection that meta-programs using the ground representation are excessively laborious to create.

The second goal, which was

To investigate the efficiency of declarative meta-programs, and discover techniques that reduce the computational overhead sufficiently to make declarative meta-programming practical.

has clearly not been met in full; efficiency is still an obstacle to metaprogramming in the ground representation. This work has, however, taken several big strides in the right direction. The combination of good interpreter design, partial evaluation, and specific low-level support promises that declarative meta-programming can be made practical. In this thesis, I have argued that for expressiveness and simple declarative semantics there is no alternative to the ground representation. The efficiency problem is a difficult one, but deserves to be taken seriously. I hope this work demonstrates that providing support for meta-programming in the ground representation is worthwhile, because declarative meta-programming is vital.

Appendix A

Definitions and Theoretical Background

A.1 Polymorphic Many-sorted Logic

The concept of a formula in polymorphic many-sorted logic defines what it means for a Gödel statement to be well-typed. The following definitions are taken from [39].

The alphabet of a polymorphic many-sorted language contains parameters, bases, constructors, variables, constants, functions, propositions, predicates, connectives, and quantifiers. Parameters are type variables, bases are types, and constructors have an arity and are used to construct new types. There is a single polymorphic universal quantifier \forall and a single polymorphic existential quantifier \exists . Also variables do not have fixed types but have their types inferred from the context in which they occur. It is assumed that there are denumerably many variables v^1, v^2, \ldots .

Definition A type is defined inductively as follows:

- 1. A parameter is a type.
- 2. A base is a type.
- 3. If c is a constructor of arity n and τ_1, \ldots, τ_n are types, then $c(\tau_1, \ldots, \tau_n)$ is a type.

A ground type is a type not containing parameters.

In a polymorphic many-sorted language, constants have types such as τ , functions have types of the form $\tau_1 \times \ldots \times \tau_n \to \tau$, and predicates have types of the form $\tau_1 \times \ldots \times \tau_n$ (where τ , τ_1 , ..., τ_n are types according to the preceding definition). A symbol is *polymorphic* if its type contains a parameter; otherwise, it is *monomorphic*.

The concept of a term t of type τ is defined so that each subterm of t has a type in t and multiple occurrences of a variable in t all have the same type in t.

Definition A term is defined inductively as follows:

- 1. A variable v is a term of type a, where a is a parameter, and the subterm v has the type a in v.
- 2. A constant c of type τ is a term of type τ and the subterm c has type τ in c.
- 3. Let f be a function of type τ₁×...×τ_n → τ and let t_i be a term of type σ_i, for i = 1,..., n. Suppose that the parameters in τ₁ ×...×τ_n → τ and the parameters of each σ_i, taken together with the parameters in the types in t_i of each of the subterms of t_i, are standardised apart. Consider the set of equations

$$\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$$

augmented with equations of the form

$$\rho_{i_1} = \rho_{i_2} = \ldots = \rho_{i_k}$$

for each variable having an occurrence in the terms t_{i_1}, \ldots, t_{i_k} $(\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}, k > 1)$, say, and where the variable is assigned the type ρ_{i_j} in t_{i_j} $(j = 1, \ldots, k)$. Then $f(t_1, \ldots, t_n)$ is a term if and only if this set of equations has a most general unifier θ , say. In this case, $f(t_1, \ldots, t_n)$ has type $\tau\theta$ and the subterm $f(t_1, \ldots, t_n)$ has type $\tau\theta$ in $f(t_1, \ldots, t_n)$. A strict subterm of $f(t_1, \ldots, t_n)$, which must be a subterm of some t_i and has type σ , say, in t_i , has type $\sigma\theta$ in $f(t_1, \ldots, t_n)$.

Note that multiple occurrences of a variable in $f(t_1, ..., t_n)$ all have the same type in $f(t_1, ..., t_n)$. Also the type of $f(t_1, ..., t_n)$ and the type in

 $f(t_1,\ldots,t_n)$ of each of its subterms is unique up to variants.

The concept of an atom A is defined so that each subterm of A has a type in A and multiple occurrences of a variable in A all have the same type in A.

Definition An *atom* is defined as follows:

- 1. A proposition p is an atom.
- 2. Let p be a predicate with type $\tau_1 \times \ldots \times \tau_n$ and let t_i be a term of type σ_i , for $i = 1, \ldots, n$. Suppose that the parameters in $\tau_1 \times \ldots \times \tau_n$ and the parameters of each σ_i , taken together with the parameters in the types in t_i of each of the subterms of t_i , are standardised apart. Consider the set of equations

$$\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$$

augmented with equations of the form

$$\rho_{i_1} = \rho_{i_2} = \ldots = \rho_{i_k}$$

for each variable having an occurrence in the terms t_{i_1}, \ldots, t_{i_k} $(\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}, k > 1)$, say, and where the variable has the type ρ_{i_j} in t_{i_j} $(j = 1, \ldots, k)$. Then $p(t_1, \ldots, t_n)$ is an atom if and only if this set of equations has a most general unifier θ , say. In this case, a subterm of $p(t_1, \ldots, t_n)$, which must be a subterm of some t_i and has type σ , say, in t_i , has the type $\sigma\theta$ in $p(t_1, \ldots, t_n)$.

Note that multiple occurrences of a variable in $p(t_1, ..., t_n)$ all have the same type in $p(t_1, ..., t_n)$. Also the type in $p(t_1, ..., t_n)$ of each of its subterms is unique up to variants.

Now the definition of a polymorphic many-sorted formula F can be given so that each subterm of F has a type in F and multiple occurrences of a variable in F all have the same type in F.

Definition A polymorphic many-sorted formula is defined inductively as follows:

1. An atom is a formula. Each subterm of the atom having type τ in the atom has the type τ in the formula.

- 2. If F is a formula, then so is $\sim F$. Each subterm of F having type τ in F has type τ in $\sim F$.
- 3. Let F and G be formulas, whose common variables are free in both formulas. Suppose that the parameters in types of subterms of F are standardised apart from the parameters in types of subterms of G. For each variable in common with F and G, form the equation

$$\rho = \sigma$$

where ρ is the type assigned to the variable in F and σ is the type assigned to the variable in G. Then $F \wedge G$ (resp., $F \vee G$, $F \to G$, $F \leftarrow G$, $F \leftrightarrow G$) is a formula if and only if the set of equations has a most general unifier θ , say. In this case, a subterm of $F \wedge G$ (resp., $F \vee G$, $F \to G$, $F \leftarrow G$, $F \leftrightarrow G$), which must be a subterm of either F or G and has type ρ in F or G, has type $\rho\theta$ in $F \wedge G$ (resp., $F \vee G$, $F \to G$, $F \leftarrow G$, $F \leftrightarrow G$).

- 4. Let F be a formula containing a free variable v. Then $\forall v \ F$ is a formula and every subterm of $\forall v \ F$ has the same type in this formula as it has in F.
- 5. Let F be a formula containing a free variable v. Then $\exists v \ F$ is a formula and every subterm of $\exists v \ F$ has the same type in this formula as it has in F.

Note that multiple occurrences of a variable in a formula have the same type in the formula. Also the types of subterms of a formula are unique up to variants.

Definition The *polymorphic many-sorted language* given by an alphabet consists of the set of all polymorphic many-sorted formulas constructed from the symbols of the alphabet.

Definition A polymorphic many-sorted theory consists of a polymorphic many-sorted language and a set of axioms which is a designated subset of closed formulas in the language of the theory.

A.2 A Program and its Completion

This section gives the definitions of the basic logic programming concepts of program, completion, and correct answer in the context of polymorphic many-sorted logic with equality. It is assumed that function declarations are transparent and that statements satisfy the head condition. These assumptions are needed to ensure that the completion of a program is well-defined. Again, these definitions are taken from [39].

Definition A statement is a polymorphic many-sorted formula of the form

$$A \leftarrow W$$

where A is an atom and W is a polymorphic many-sorted formula. The formula W may be absent. Any variables in A and any free variables in W are assumed to be universally quantified at the front of the statement. A is called the head and W the body of the statement.

Definition A *program* is a finite set of statements.

Definition A goal is a polymorphic many-sorted formula of the form

$$\leftarrow W$$

where W is a polymorphic many-sorted formula and any free variables of W are assumed to be universally quantified at the front of the goal.

Definition The *definition* of a proposition or predicate p appearing in a program P is the set of all statements in P which have p in their head.

Definition Suppose the definition of a proposition or predicate p of arity $n \geq 0$ in a program is

$$A_1 \leftarrow W_1$$

$$A_k \leftarrow W_k$$

Then the *completed definition* of p is the formula

$$\forall x_1 \dots \forall x_n (p(x_1, \dots, x_n) \leftrightarrow E_1 \vee \dots \vee E_k)$$

where E_i is $\exists y_1 \ldots \exists y_d ((x_1 = t_1) \land \ldots \land (x_n = t_n) \land W_i)$, A_i is $p(t_1, \ldots, t_n)$, y_1, \ldots, y_d are the variables in A_i and the free variables in W_i , and x_1, \ldots, x_n are variables not appearing anywhere in the definition of p.

Definition Let P be a program and p a proposition or predicate occurring in P. Suppose there is no program statement in P with p in its head. Then the *completed definition* of p is the formula

$$\forall x_1 \dots \forall x_n \sim p(x_1, \dots, x_n).$$

Definition The *equality theory* for a program consists of all axioms of the following form (where $s \neq t$ is an abbreviation for $\sim (s = t)$):

- 1. $c \neq d$, for all pairs c, d of distinct constants whose types have a common instance.
- 2. $\forall (f(x_1,\ldots,x_n)\neq g(y_1,\ldots,y_m)), \text{ for all pairs } f,g \text{ of distinct functions}$ whose range types have a common instance.
- 3. $\forall (f(x_1,\ldots,x_n)\neq c)$, for each constant c and function f such that the type of c and the range type of f have a common instance.
- 4. $\forall (t[x] \neq x)$, for each term t[x] containing x and different from x.
- 5. $\forall ((x_1 \neq y_1) \lor \ldots \lor (x_n \neq y_n) \rightarrow f(x_1, \ldots, x_n) \neq f(y_1, \ldots, y_n)),$ for each function f.
- 6. $\forall x(x=x)$.
- 7. $\forall ((x_1 = y_1) \land \dots \land (x_n = y_n) \rightarrow f(x_1, \dots, x_n) = f(y_1, \dots, y_n)),$ for each function f.
- 8. $\forall ((x_1 = y_1) \land \ldots \land (x_n = y_n) \rightarrow (p(x_1, \ldots, x_n) \rightarrow p(y_1, \ldots, y_n))),$ for each predicate p (including =).

Definition Let P be a program. The *completion* of P, denoted by comp(P), is the collection of completed definitions of propositions and predicates in P together with the above equality theory.

Definition Let P be a program, G a goal $\leftarrow W$, and θ an answer for $P \cup \{G\}$. We say θ is a correct answer for $comp(P) \cup \{G\}$ if $\forall (W\theta)$ is a logical consequence of comp(P).

The concept of a correct answer gives a declarative description of the desired output from a goal and program.

A.3 SLDQE-resolution

In this section, I give the formal definitions and theorems relating to SLDQE-resolution, which was introduced in Section 3.3.1 and [16].

The set of free variables in a formula F is denoted by $\operatorname{fvar}(F)$. The notation \overline{x} denotes the sequence or set (depending on context) of variables x_1, \ldots, x_n for some n. $\forall \overline{x}F$ (resp., $\exists \overline{x}F$) means that all the variables in \overline{x} are universally (resp., existentially) quantified in front of F. $S \ll_V P$ denotes a variant S of a statement in P with $\operatorname{fvar}(S) \cap V = \emptyset$.

A goal statement for a program P, which is a statement, is used instead of the usual notation $\leftarrow T$ for a goal for P. In particular, a goal $\leftarrow T$ can be written $q(\overline{x}) \leftarrow T$ where $\overline{x} = \text{fvar}(T)$ and q is a new predicate.

Definition A restricted quantification is a formula of the form $\forall \overline{x}(V \to W)$. It is atomic if V and W are atoms. W is called the head and V the body.

Definition A quantified-normal (q-normal) formula F is of the form $F_1 \wedge \cdots \wedge F_n$ where each F_i is either

- an atom (called an atomic conjunct of F) or
- an atomic restricted quantification.

An atom A occurs positively in F if, for some i, either $A = F_i$, or F_i is an atomic restricted quantification whose head is A. An atom A occurs negatively in F if, for some i, A is the body of the atomic restricted quantification F_i . A variable $x \in \text{fvar}(F)$ occurs positively (resp., negatively) in F if it occurs in an atom that occurs positively (resp., negatively) in F.

A q-normal statement is a statement whose body is q-normal. A q-normal program is a program whose statements are all q-normal.

A q-normal goal statement for a program P is a q-normal statement.

Definition If P is a program, G is the goal statement $U \leftarrow T$, and θ a

substitution for the variables in U then θ is a correct answer for P and G if $comp(P) \models \forall (U \leftarrow T) \rightarrow \forall U\theta$.

Definition Let P be a q-normal program and G a q-normal goal statement. An SLDQE-refutation of rank k for P and G consists of sequences $G = G_0, G_1, \ldots, G_n$ and G_1, \ldots, G_n of statements such that $G_n = U\theta_n \leftarrow True$ and, for each $i \in \{0, \ldots, n-1\}$ either:

- 1. $G_i = U\theta_i \leftarrow X \land A \land Y$, A is the selected atom, $H \leftarrow B \ll_{\text{fvar}(G_i)} P$, $\text{mgu}(\{A, H\}) = \theta'_i$, $C_{i+1} = H \leftarrow B$, $\theta_{i+1} = \theta_i \theta'_i$ and $G_{i+1} = U\theta_{i+1} \leftarrow X\theta'_i \land B\theta'_i \land Y\theta'_i$; or
- 2. k > 0, $G_i = U\theta_i \leftarrow X \land \forall \overline{x}(V \to W) \land Y$, the selected formula in the body of G_i is $\forall \overline{x}(V \to W)$, and $\{\phi_1, \dots, \phi_m\}$ is a ground answer-set with domain \overline{x} for an SLDQE-tree of rank k-1 for P and $W \leftarrow V$. In this case, $C_{i+1} = W \leftarrow V$, $\theta_{i+1} = \theta_i$ and $G_{i+1} = U\theta_{i+1} \leftarrow X \land W\phi_1 \land \dots \land W\phi_m \land Y$.

In both cases, G_{i+1} is said to be derived from G_i and C_{i+1} .

The substitution θ_n restricted to the variables of U is a computed answer for P and G.

Definition Let P be a q-normal program and G a q-normal goal statement. A completed SLDQE-tree of rank k for P and G is a finite tree satisfying:

- 1. each node of the tree is a q-normal goal statement;
- 2. the root node is G;
- 3. Suppose $G' = U\theta \leftarrow X \land S \land Y$ is a non-leaf node in the tree and that S is the selected formula, then either
 - (a) S is an atom and, for each statement $H \leftarrow B \ll_{\text{fvar}(G')} P$ where there exists $\text{mgu}(\{S, H\}) = \theta'$, the node has a child $U\theta\theta' \leftarrow X\theta' \wedge B\theta' \wedge Y\theta'$;
 - (b) k > 0 and S is of the form $\forall \overline{x}(V \to W), \{\phi_1, \dots, \phi_m\}$ is a ground answer-set with domain \overline{x} for an SLDQE-tree of rank k-1 for P and $W \leftarrow V$, and there is a single child $U\theta \leftarrow X \wedge W\phi_1 \wedge \cdots \wedge W\phi_m \wedge Y$;

- 4. Suppose $G' = U\theta \leftarrow X \land S \land Y$ is a leaf node in the tree and that S (distinct from True) is the selected formula, then S is an atom and there does not exist $H \leftarrow B \ll_{\mathbf{fvar}(G')} P$ so that $\{H, S\}$ unifies;
- 5. $U\theta \leftarrow True \text{ is a } successful \text{ leaf node with } answer \theta \rightharpoonup \text{fvar}(U).$

The set of answers at successful leaf nodes is an answer-set for P and G. If each answer is ground, then the answer-set is ground.

An SLDQE-refutation (completed SLDQE-tree) for P and G is an SLDQE-refutation (completed SLDQE-tree) of rank k for P and G for some k.

Proofs of the following two theorems are given in [16].

Theorem Let P be a q-normal program and $\forall \overline{y}(T \to U)$ an atomic restricted quantification. If $\{\theta_1, \ldots, \theta_l\}$ is a ground answer-set with domain \overline{y} for P and $U \leftarrow T$, then $\text{comp}(P) \models \forall (\forall \overline{y}(T \to U) \leftarrow U\theta_1 \land \cdots \land U\theta_l)$.

Theorem Let P be a q-normal program and G a q-normal goal statement. Then every computed answer for P and G is a correct answer for $comp(P) \cup \{G\}$.

The definitions above closely follow the definitions given for SLDNF-resolution in [50]. These do not define an SLDNF-tree for every normal program and normal goal; for example, $\{P \leftarrow \neg P\} \cup \{\leftarrow P\}$ does not have an SLDNF-tree. Although adequate for the purpose of proving the soundness of SLDQE-resolution, the definitions given here similarly do not define an SLDQE-tree for every q-normal program and q-normal goal. An alternative definition of SLDNF-resolution that defines an SLDNF-tree for every normal program and normal goal was given by Apt and Doets in [2]; a reformulation of the definition of an SLDQE-tree according to the pattern provided in that work would be straightforward.

Appendix B

Gödel Meta-programming Facilities

B.1 The Syntax Module

```
EXPORT Syntax.
```

```
% Module providing a number of abstract data types for representing % object-level expressions (using a ground representation) and % predicates for processing terms of these types.
```

```
IMPORT
           Strings.
BASE
                           % Type of a term representing the name of
           Name,
                           % a symbol.
           Type,
                           % Type of a term representing a type.
                           % Type of a term representing a term.
           Term,
           Formula,
                           % Type of a term representing a formula.
           TypeSubst,
                           % Type of a term representing a type
                            % substitution.
           TermSubst,
                           % Type of a term representing a term
                            % substitution.
           FunctionInd,
                           % Type of a term representing a function
                            % indicator.
           PredicateInd,
                           % Type of a term representing a predicate
                            % indicator.
           VarTyping.
                            % Type of a term representing a variable
                            % typing.
CONSTANT
           NoPredInd: PredicateInd.
```

% Constant stating that a predicate has no indicator.

```
CONSTANT
            ZPZ, ZP, PZ : PredicateInd.
% Constants representing the predicate indicators zPz, zP, and Pz
% (resp.).
CONSTANT
            NoFunctInd: FunctionInd.
% Constant stating that a function has no indicator.
            XFX, XFY, YFX, XF, FX, YF, FY : Integer -> FunctionInd.
FUNCTION
% Functions representing the function indicators xFx, xFy, yFx, xF, Fx,
% yF, and Fy (resp.).
PREDICATE And:
  Formula
                   % Representation of a formula W.
* Formula
                   % Representation of a formula V.
* Formula.
                   \% Representation of the formula \mbox{W} & \mbox{V}\,.
PREDICATE AndWithEmpty:
  Formula
                   % Representation of possibly empty formula W.
* Formula
                   \mbox{\ensuremath{\mbox{\%}}} Representation of possibly empty formula \mbox{\ensuremath{\mbox{V}}}.
                   \% Representation of W & V, if W and V are non-empty;
* Formula.
                   % W, if V is the empty formula; and V, if W is the
                   % empty formula.
DELAY
            AndWithEmpty(x,y,_) UNTIL GROUND(x) & GROUND(y).
PREDICATE Or :
  Formula
                   % Representation of a formula W.
* Formula
                   % Representation of a formula V.
                   \% Representation of the formula W \/ V.
* Formula.
PREDICATE Not:
  Formula
                   % Representation of a formula W.
                   \% Representation of the formula \tilde{\ } W.
* Formula.
```

PREDICATE Implies :

```
Formula % Representation of a formula W.

* Formula % Representation of a formula V.
```

PREDICATE IsImpliedBy :

Formula % Representation of a formula W.

* Formula % Representation of a formula V.

PREDICATE Equivalent :

Formula % Representation of a formula W.

* Formula % Representation of a formula V.

PREDICATE Some :

List(Term) % List of representations of variables.

* Formula % Representation of a formula.

 $\mbox{\%}$ variables in the first argument of the formula in

% the second argument.

PREDICATE All:

List(Term) % List of representations of variables.

* Formula % Representation of a formula.

* Formula. % Representation of the formula obtained by taking

% the universal quantification over the set of

% variables in the first argument of the formula in

% the second argument.

PREDICATE IfThen:

Formula % Representation of a formula, Condition.

* Formula % Representation of a formula, Formula.

% the form IF Condition THEN Formula.

PREDICATE IfSomeThen:

```
List(Term) % List of representations of variables, v1,...,vn.
```

- * Formula % Representation of a formula, Condition.
- * Formula % Representation of a formula, Formula.
- - % the form IF SOME [v1,...,vn] Condition THEN Formula.

PREDICATE IfThenElse:

- Formula % Representation of a formula, Condition.
- * Formula % Representation of a formula, Formula1.
- * Formula % Representation of a formula, Formula2.
- - % has the form IF Condition THEN Formula1 ELSE
 - % Formula2.

PREDICATE IfSomeThenElse:

- List(Term) % List of representations of variables, v1,...,vn.
- * Formula % Representation of a formula, Condition.
- * Formula % Representation of a formula, Formula1.
- * Formula % Representation of a formula, Formula2.
- - % has the form IF SOME [v1,...,vn] Condition THEN
 - % Formula1 ELSE Formula2.

PREDICATE Commit:

- Integer % Commit label.
- * Formula % Representation of a formula.
- - % the formula in the second argument using a commit
 - % with this label.

PREDICATE IntensionalSet:

- Term % Representation of a term T.
- * Formula % Representation of a formula W.

${\tt PREDICATE} \quad {\tt Parameter} \,: \,$

Type. % Representation of a parameter.

DELAY Parameter(x) UNTIL GROUND(x).

PREDICATE ParameterName :

```
Type % Representation of a parameter.
```

- * String % The root of the name of the parameter.
- * Integer. % The index of the parameter.

PREDICATE TypeMaxParIndex:

```
List(Type) % List of representations of types.
```

- * Integer. % One more than the maximum index of parameters
 - % appearing in these types. (If there are no such
 - % parameters, this argument is 0.)

DELAY TypeMaxParIndex(x,_) UNTIL GROUND(x).

PREDICATE Variable:

Term. % Representation of a variable.

DELAY Variable(x) UNTIL GROUND(x).

PREDICATE VariableName :

- Term % Representation of a variable.
- * String % The root of the name of the variable.
- * Integer. % The index of the variable.

PREDICATE TermMaxVarIndex :

- List(Term) % List of representations of terms.
- st Integer. % One more than the maximum index of variables
 - $\mbox{\ensuremath{\mbox{\%}}}$ appearing in these terms. (If there are no such
 - % variables, this argument is 0.)

DELAY TermMaxVarIndex(x,_) UNTIL GROUND(x).

PREDICATE FormulaMaxVarIndex:

List(Formula) % List of representations of formulas.

* Integer. % One more than the maximum index of variables

% appearing in these formulas. (If there are no such

% variables, this argument is 0.)

DELAY FormulaMaxVarIndex(x,_) UNTIL GROUND(x).

PREDICATE TypeParameters :

Type % Representation of a type.

% parameters occurring in this type.

DELAY TypeParameters(x,_) UNTIL GROUND(x).

PREDICATE TermVariables:

Term % Representation of a term.

% (free) variables occurring in this term.

DELAY TermVariables(x,_) UNTIL GROUND(x).

PREDICATE FormulaVariables:

Formula % Representation of a formula.

% free variables occurring in this formula.

DELAY Formula $Variables(x, _)$ UNTIL GROUND(x).

PREDICATE EmptyFormula:

Formula. % Representation of the empty formula.

PREDICATE EmptyTypeSubst:

TypeSubst. % Representation of the empty type substitution.

PREDICATE EmptyTermSubst:

TermSubst. % Representation of the empty term substitution.

PREDICATE EmptyVarTyping:

VarTyping. % Representation of the empty variable typing.

PREDICATE NonParameter:

Type. % Representation of a non-parameter type.

DELAY NonParameter(x) UNTIL GROUND(x).

PREDICATE NonVariable:

Term. % Representation of a non-variable term.

DELAY NonVariable(x) UNTIL GROUND(x).

PREDICATE Atom:

Formula. % Representation of an atom.

DELAY Atom(x) UNTIL GROUND(x).

PREDICATE ConjunctionOfAtoms:

Formula. % Representation of a conjunction of atoms.

DELAY ConjunctionOfAtoms(x) UNTIL GROUND(x).

PREDICATE Literal:

Formula. % Representation of a literal.

DELAY Literal(x) UNTIL GROUND(x).

${\tt PREDICATE} \quad {\tt ConjunctionOfLiterals} \ :$

Formula. % Representation of a conjunction of literals.

DELAY ConjunctionOfLiterals(x) UNTIL GROUND(x).

PREDICATE CommitFreeFormula:

Formula. % Representation of a commit-free formula.

DELAY CommitFreeFormula(x) UNTIL GROUND(x).

PREDICATE GroundType :

Type. % Representation of a parameter-free type.

DELAY GroundType(x) UNTIL GROUND(x).

PREDICATE GroundTerm:

Term. % Representation of a term with no free variables.

DELAY GroundTerm(x) UNTIL GROUND(x).

PREDICATE GroundAtom:

Formula. % Representation of an atom with no free variables.

DELAY GroundAtom(x) UNTIL GROUND(x).

PREDICATE ClosedFormula:

Formula. % Representation of a formula with no free variables.

DELAY ClosedFormula(x) UNTIL GROUND(x).

PREDICATE Body:

Formula. % Representation of a standard body.

DELAY Body(x) UNTIL GROUND(x).

${\tt PREDICATE} \quad {\tt NormalBody} \,: \,$

Formula. % Representation of a normal body.

DELAY NormalBody(x) UNTIL GROUND(x).

PREDICATE DefiniteBody :

DELAY DefiniteBody(x) UNTIL GROUND(x).

PREDICATE Goal:

DELAY Goal(x) UNTIL GROUND(x).

PREDICATE NormalGoal:

Formula. % Representation of a normal goal.

DELAY NormalGoal(x) UNTIL GROUND(x).

PREDICATE DefiniteGoal:

Formula. % Representation of a definite goal.

DELAY DefiniteGoal(x) UNTIL GROUND(x).

PREDICATE Resultant :

Formula. % Representation of a standard resultant.

DELAY Resultant(x) UNTIL GROUND(x).

PREDICATE NormalResultant:

Formula. % Representation of a normal resultant.

DELAY NormalResultant(x) UNTIL GROUND(x).

PREDICATE DefiniteResultant :

Formula. % Representation of a definite resultant.

DELAY DefiniteResultant(x) UNTIL GROUND(x).

PREDICATE Statement:

Formula. % Representation of a standard statement.

DELAY Statement(x) UNTIL GROUND(x).

PREDICATE NormalStatement:

Formula. % Representation of a normal statement.

DELAY NormalStatement(x) UNTIL GROUND(x).

PREDICATE DefiniteStatement:

Formula. % Representation of a definite statement.

DELAY DefiniteStatement(x) UNTIL GROUND(x).

PREDICATE BaseType :

Type % Representation of a non-opaque base.

* Name. % Representation of the name of this base.

PREDICATE ConstructorType :

Type % Representation of a non-opaque type with a

% constructor at the top level.

* Name % Representation of the name of this constructor.

* List(Type). % List of representations of the top-level subtypes

% of this type.

PREDICATE ConstantTerm :

Term % Representation of a non-opaque constant.

* Name. % Representation of the name of this constant.

PREDICATE FunctionTerm :

Term % Representation of a non-opaque term with a function

% at the top level.

* Name % Representation of the name of this function.

% of this term.

PREDICATE PropositionAtom:

```
Formula % Representation of a non-opaque proposition.

* Name. % Representation of the name of this proposition.
```

PREDICATE PredicateAtom:

Formula % Representation of a non-opaque atom with a predicate

% at the top level.

* Name % Representation of the name of this predicate.

% this atom.

PREDICATE OpaqueType:

Type. % Representation of an opaque type.

DELAY OpaqueType(x) UNTIL GROUND(x).

PREDICATE OpaqueTerm:

Term. % Representation of an opaque term.

DELAY OpaqueTerm(x) UNTIL GROUND(x).

PREDICATE OpaqueAtom:

Formula. % Representation of an opaque atom.

DELAY OpaqueAtom(x) UNTIL GROUND(x).

PREDICATE ApplySubstToType :

Type % Representation of a type.

* TypeSubst % Representation of a type substitution.

% this substitution to this type.

DELAY ApplySubstToType(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE ApplySubstToTerm:

Term % Representation of a term.

* TermSubst % Representation of a term substitution.

DELAY ApplySubstToTerm(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE ApplySubstToFormula:

```
Formula % Representation of a formula.
```

- * TermSubst % Representation of a term substitution.
- * Formula. % Representation of the formula obtained by applying

% this substitution to this formula.

DELAY ApplySubstToFormula(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE ComposeTypeSubsts:

```
TypeSubst % Representation of a type substitution.

* TypeSubst % Representation of a type substitution.
```

- st TypeSubst. % Representation of the substitution obtained by

 - % that they appear as arguments).

DELAY ComposeTypeSubsts(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE ComposeTermSubsts:

- TermSubst % Representation of a term substitution.
- * TermSubst % Representation of a term substitution.
- * TermSubst. % Representation of the substitution obtained by
 - % composing these two substitutions (in the order
 - % that they appear as arguments).

DELAY ComposeTermSubsts(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE CombineVarTypings :

```
VarTyping % Representation of a variable typing.
```

- * VarTyping % Representation of a variable typing.

% combining these two variable typings.

DELAY CombineVarTypings(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE RestrictSubstToType :

```
Type % Representation of a type.
```

- * TypeSubst % Representation of a type substitution.
- - % restricting this type substitution to the
 - % parameters in this type.

DELAY RestrictSubstToType(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE RestrictSubstToTerm :

Term % Representation of a term.

- * TermSubst % Representation of a term substitution.
- * TermSubst. % Representation of the substitution obtained by
 - % restricting this term substitution to the variables
 - % in this term.

DELAY RestrictSubstToTerm(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE RestrictSubstToFormula:

- Formula % Representation of a formula.
- * TermSubst. % Representation of the substitution obtained by
 - % restricting this term substitution to the free
 - % variables in this formula.

DELAY RestrictSubstToFormula(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE BindingToTypeSubst:

- Type % Representation of a parameter.
- * Type % Representation of a type.
- * TypeSubst. % Representation of the type substitution containing
 - % just the binding in which this parameter is bound
 - % to this type.

DELAY BindingToTypeSubst(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE BindingToTermSubst:

Term % Representation of a variable.

- * Term % Representation of a term.
- st TermSubst. % Representation of the term substitution containing
 - % just the binding in which this variable is bound

% to this term.

DELAY BindingToTermSubst(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE BindingToVarTyping:

- Term % Representation of a variable.

 * Type % Representation of a type.
- - % just the binding in which this variable is bound
 - % to this type.

DELAY BindingToVarTyping(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE BindingInTypeSubst:

- TypeSubst % Representation of a type substitution.
- * Type % Representation of a parameter in the first
 - % component of a binding in this substitution.
- - % is bound in this substitution.

DELAY BindingInTypeSubst(x,_,_) UNTIL GROUND(x).

PREDICATE BindingInTermSubst:

- TermSubst % Representation of a term substitution.
- * Term % Representation of a variable in the first
 - % component of a binding in this substitution.
- - % is bound in this substitution.

DELAY BindingInTermSubst(x,_,_) UNTIL GROUND(x).

PREDICATE BindingInVarTyping:

- VarTyping % Representation of a variable typing.
- - % this variable typing.
- - % is bound in this variable typing.

DELAY BindingInVarTyping(x,_,_) UNTIL GROUND(x).

PREDICATE DelBindingInTypeSubst:

```
TypeSubst
                                                                                     % Representation of a type substitution.
* Type
                                                                                     % Representation of a parameter.
 * Туре
                                                                                     % Representation of a type.
                                                                                     % Representation of the type substitution obtained
 * TypeSubst.
                                                                                     % from the first argument by deleting the binding of
                                                                                     % this parameter to this type.
                                                     DelBindingInTypeSubst(x,_,_,_) UNTIL GROUND(x).
DELAY
PREDICATE DelBindingInTermSubst:
          TermSubst
                                                                                     % Representation of a term substitution.
 * Term
                                                                                     % Representation of a variable.
 * Term
                                                                                     % Representation of a term.
 * TermSubst.
                                                                                     % Representation of the term substitution obtained
                                                                                     % from the first argument by deleting the binding of
                                                                                     % this variable to this term.
                                                    DelBindingInTermSubst(x,_,_,_) UNTIL GROUND(x).
DELAY
PREDICATE DelBindingInVarTyping:
                                                                                     % Representation of a variable typing.
          VarTyping
* Term
                                                                                     % Representation of a variable.
 * Type
                                                                                     % Representation of a type.
                                                                                     % Representation of the variable typing obtained
 * VarTyping.
                                                                                     % from the first argument by deleting the binding
                                                                                     % = % \frac{1}{2} \left( \frac{1}{2} \right) 
DELAY
                                                     DelBindingInVarTyping(x,_,_,_) UNTIL GROUND(x).
PREDICATE UnifyTypes:
```

```
Type % Representation of a type.

* Type % Representation of a type.

* TypeSubst % Representation of a type substitution.

* TypeSubst. % Representation of the type substitution obtained by % composing the type substitution in the third % argument with a specific, unique mgu for the types % which are obtained by applying the type substitution % in the third argument to the types in the first two % arguments. (The mgu binds a parameter in the first % argument to a parameter in the second argument, not % the other way around.)
```

```
DELAY UnifyTypes(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).
```

PREDICATE UnifyTerms :

DELAY UnifyTerms $(x,y,z,_)$ UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE UnifyAtoms:

DELAY UnifyAtoms(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE RenameTypes:

```
List(Type)  % List of representations of types.

* List(Type)  % List of representations of types.

* List(Type).  % List of representations of the types obtained by  % renaming the parameters of the types in the second  % argument by a specific, unique type substitution  % such that they become distinct from the parameters  % in the types in the first argument.
```

DELAY RenameTypes(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE RenameTerms :

```
List(Term) % List of representations of terms.

* List(Term) % List of representations of terms.

* List(Term). % List of representations of the terms obtained by % renaming the (free) variables of the terms in the % second argument by a specific, unique term % substitution such that they become distinct from % the (free) variables in the terms in the first % argument.
```

DELAY RenameTerms($x,y,_$) UNTIL GROUND(x) & GROUND(y).

PREDICATE RenameFormulas:

```
List(Formula) % List of representations of formulas.

* List(Formula) % List of representations of formulas.

* List(Formula). % List of representations of the formulas obtained % by renaming the free variables of the formulas in % the second argument by a specific, unique term % substitution such that they become distinct from % the free variables in the formulas in the first % argument.
```

DELAY RenameFormulas($x,y,_$) UNTIL GROUND(x) & GROUND(y).

PREDICATE VariantTypes :

```
List(Type) % List of representation of types.

* List(Type). % List of representations of types which are variants % of the types in the first argument.
```

DELAY VariantTypes(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE VariantTerms :

```
List(Term) % List of representation of terms.

* List(Term). % List of representations of terms which are variants % of the terms in the first argument.
```

DELAY VariantTerms(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE VariantFormulas :

```
List(Formula) % List of representations of formulas.

* List(Formula). % List of representations of formulas which are
```

% variants of the formulas in the first argument.

DELAY VariantFormulas(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE StandardiseFormula:

Formula % Representation of a formula.

* Integer % A non-negative integer. * Integer % A non-negative integer.

st Formula. % Representation of the formula obtained by

% systematically replacing the variables of the % formula in the first argument by variables with

% indexes greater than or equal to the second

% argument and strictly less than the third argument.

DELAY StandardiseFormula($x,y,_{,-}$) UNTIL GROUND(x) & GROUND(y).

PREDICATE Derive :

Formula % Representation of the head of a resultant.

* Formula % Representation of the body to the left of the

% selected atom of this resultant.

* Formula % Representation of the selected atom in this

% resultant.

* Formula % Representation of the body to the right of the

% selected atom of this resultant.

* Formula % Representation of a statement whose head unifies

% with the selected atom in the resultant.

* TermSubst % Representation of a specific, unique mgu of the

% head of the selected atom and the head of this

% statement. (The mgu binds a variable in the head to

% a variable in the selected atom, not the other way

% around.)

% from this resultant and this statement using this

% mgu.

PREDICATE Resolve :

Formula % Representation of an atom.

```
* Formula
                  % Representation of a statement.
                  % A non-negative integer.
* Integer
* Integer
                  % A non-negative integer.
* TermSubst
                  % Representation of a term substitution.
                  % Representation of the substitution obtained by
* TermSubst
                  % composing the substitution in the fifth argument
                  % with a specific, unique mgu of the atom in the
                  % first argument with the substitution in the fifth
                  \% argument applied and the head of a renamed version
                  % of the statement in the second argument. The
                  % renaming is achieved by systematically replacing
                  % the variables of the statement by variables with
                  % indexes greater than or equal to the third argument
                  % and strictly less than the fourth argument. (The
                  % mgu binds a variable in the head to a variable in
                  % the atom in the first argument with the substitution
                  % applied, not the other way around.)
* Formula.
                  % Representation of the formula obtained by applying
                  % the mgu in the sixth argument to the body of the
                  % renamed statement.
           Resolve(x,y,z,_,u,_,_) UNTIL GROUND(x) & GROUND(y) &
DELAY
                                                 GROUND(z) & GROUND(u).
```

PREDICATE ResolveAll:

```
Formula
                  % Representation of an atom.
* List(Formula)
                 % List of representations of statements.
* Integer
                 % A non-negative integer.
* Integer
                 % A non-negative integer.
                 % A non-negative integer.
* Integer
                 % A non-negative integer.
* Integer
* TermSubst
                 % Representation of a term substitution.
* List(TermSubst) % List of representations of the substitutions
                 \% obtained by composing the substitution in the
                 % seventh argument with a specific, unique mgu of the
                 \% atom in the first argument with the substitution in
                 % the seventh argument applied and the head of a
                 % renamed version of a corresponding statement in the
                 % second argument. The renaming is achieved by
                 % systematically replacing the variables of the
                 % statement by variables with indexes greater than or
                 % equal to the third argument and strictly less than
                 \% the fourth argument, and commit labels of the
                 % statement by commit labels greater than or equal
                 \% to the fifth argument and strictly less than the
                 % sixth argument. (Each mgu binds a variable in the
                  % head to a variable in the atom in the first
```

```
% argument with the substitution applied, not the
                  % other way around.)
* List(Formula).
                 % List of representations of the formulas obtained by
                  % applying the corresponding mgu in the eighth
                  \% argument to the body of the corresponding renamed
                  % statement.
           ResolveAll(x,y,z,_,u,_,v,_,_) UNTIL GROUND(x) & GROUND(y) &
DELAY
                                      GROUND(z) & GROUND(u) & GROUND(v).
B.2
       The Programs Module
CLOSED
           Programs.
% Module providing the abstract data type Program for representing
\% Goedel programs (using a ground representation) and predicates for
% processing terms of type Program.
IMPORT
           Syntax.
BASE
           Program,
                            \% Type of a term representing (the flat form
                            % of) a program.
           ModulePart,
                            % Type of constants representing the keywords
                            % EXPORT, LOCAL, CLOSED, and MODULE.
           Condition.
                            % Type of a term representing a condition
                            % in a DELAY declaration.
CONSTANT
           Export, Local, Closed, Module: ModulePart.
%
% Constants representing the keywords EXPORT, LOCAL, CLOSED, and MODULE
% (resp.).
PREDICATE TypeInProgram:
  Program
                  % Representation of a program.
                  % Representation of a type in the flat language of
* Type.
                  % this program.
           TypeInProgram(x,y) UNTIL GROUND(x) & GROUND(y).
DELAY
PREDICATE TermInProgram :
                  % Representation of a program.
  Program
                  % Representation of a variable typing in the flat
* VarTyping
```

% language of this program.

```
* Term % Representation of a term in the flat language of
```

% this program.

% to this variable typing.

* VarTyping. % Representation of the variable typing obtained by

% combining the variable typing in the second argument

% with the types of all free variables occurring in the

% term.

DELAY TermInProgram(x,y,z,_,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE FormulaInProgram :

Program % Representation of a program.

* VarTyping % Representation of a variable typing in the flat

% language of this program.

% body or standard resultant, in the flat language of

% this program.

* VarTyping. % Representation of the variable typing obtained by

% combining the variable typing in the second argument

% with the types of all free variables occurring in

% the formula.

DELAY FormulaInProgram(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE TypeInModule:

Program % Representation of a program.

* String % Name of a module in this program.

* ModulePart % Representation of a part keyword of this module.

* Type. % Representation of a type in the flat language of

% this module, if the keyword is LOCAL or MODULE, or

 $\mbox{\ensuremath{\mbox{\%}}}$ the flat export language of this module, if the

% keyword is EXPORT or CLOSED.

DELAY TypeInModule(x,y,z,u) UNTIL GROUND(x) & GROUND(y) & GROUND(z) & GROUND(u).

PREDICATE TermInModule :

Program % Representation of a program.

* String % Name of a module in this program.

% this module, if the keyword is LOCAL or MODULE, or % the flat export language of this module, if the

% keyword is EXPORT or CLOSED.

* Type % Representation of the type of this term with

% respect to this variable typing.

% combining the variable typing in the fourth argument

 $\mbox{\ensuremath{\mbox{\%}}}$ with the types of all variables occurring in the

% term.

PREDICATE FormulaInModule :

${\tt Program}$	% Representation of a program.
* String	% Name of a module in this program.
* ModulePart	% Representation of a part keyword of this module.
st VarTyping	% Representation of a variable typing in the flat
	% language of this module, if the keyword is LOCAL
	% or MODULE, or the flat export language of this
	% module, if the keyword is EXPORT or CLOSED.
* Formula	% Representation of a formula, which is a standard body
	% or standard resultant, in the flat language of this
	% module, if the keyword is LOCAL or MODULE, or the
	% flat export language of this module, if the keyword
	% is EXPORT or CLOSED.
st VarTyping.	% Representation of the variable typing obtained by
	% combining the variable typing in the fourth argument
	% with the types of all free variables occurring in
	% the formula.

PREDICATE StringToProgramType :

```
% program whose string representation is the third
% argument.
```

DELAY StringToProgramType(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE StringToProgramTerm:

```
% Representation of a program.
 Program
* String
                  % Name of a module in this program.
```

* String % A string.

* List(Term). % List (in a definite order) of representations of % terms in the flat language of this module wrt this

% program whose string representation is the third

% argument.

DELAY StringToProgramTerm(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE StringToProgramFormula:

```
Program
                  % Representation of a program.
* String
                  % Name of a module in this program.
```

* String % A string.

% List (in a definite order) of representations of * List(Formula).

> % formulas, which are standard bodies or standard % resultants, in the flat language of this module wrt

% this program whose string representation is the

% third argument.

DELAY StringToProgramFormula(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE ProgramTypeToString:

```
Program
                  % Representation of a program.
                  % Name of a module in this program.
* String
```

* Туре % Representation of a type.

% The string representation of this type. (Subtypes * String. % of the type not in the flat language of the module

% do not appear.)

ProgramTypeToString(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & DELAY GROUND(z).

PREDICATE ProgramTermToString:

```
Program % Representation of a program.
```

- * String % Name of a module in this program.
- * Term % Representation of a term.
- - % of the term not in the flat language of the module
 - % do not appear.)

DELAY ProgramTermToString(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE ProgramFormulaToString:

- Program % Representation of a program.
- * String % Name of a module in this program.
- - % body or standard resultant.
- * String. % The string representation of this formula. (Terms
 - $\mbox{\ensuremath{\mbox{\%}}}$ of the formula not in the flat language of the
 - % module do not appear.)

DELAY ProgramFormulaToString(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE ProgramBaseName :

- Program % Representation of a program.
- * String % Name of a module in this program.
- * String % Name of a base.
- - % this base.

DELAY ProgramBaseName(x,y,z,u) UNTIL GROUND(x) &

((GROUND(y) & GROUND(z)) \/ GROUND(u)).

PREDICATE ProgramConstructorName:

- Program % Representation of a program.
- * String % Name of a module in this program.
- * String % Name of a constructor.
- * Integer % Arity of this constructor.
- - % this constructor.

DELAY ProgramConstructorName(x,y,z,u,v) UNTIL GROUND(x) &

```
PREDICATE ProgramConstantName :
                  % Representation of a program.
  Program
                  % Name of a module in this program.
* String
* String
                  % Name of a constant.
* Name.
                  \% Representation of the corresponding flat name of
                  % this constant.
           ProgramConstantName(x,y,z,u) UNTIL GROUND(x) &
DELAY
                                ((GROUND(y) & GROUND(z)) \/ GROUND(u)).
PREDICATE ProgramFunctionName:
  Program
                  % Representation of a program.
* String
                  % Name of a module in this program.
* String
                  % Name of a function.
                  % Arity of this function.
* Integer
                  % Representation of the corresponding flat name of
* Name.
                  % this function.
           ProgramFunctionName(x,y,z,u,v) UNTIL GROUND(x) &
DELAY
                   ((GROUND(y) \& GROUND(z) \& GROUND(u)) \setminus / GROUND(v)).
PREDICATE ProgramPropositionName:
  Program
                  % Representation of a program.
* String
                  \% Name of a module in this program.
* String
                  % Name of a proposition.
                  % Representation of the corresponding flat name of
* Name.
                  % this proposition.
DELAY
           ProgramPropositionName(x,y,z,u) UNTIL GROUND(x) &
                                ((GROUND(y) & GROUND(z)) \/ GROUND(u)).
PREDICATE ProgramPredicateName :
  Program
                  % Representation of a program.
* String
                  % Name of a module in this program.
* String
                  % Name of a predicate.
                  % Arity of this predicate.
* Integer
* Name.
                  \% Representation of the corresponding flat name of
                  % this predicate.
```

```
Program
                 % Representation of a program.
                 % Name of the main module in this program.
* String.
           MainModuleInProgram(x,_) UNTIL GROUND(x).
DELAY
PREDICATE ModuleInProgram:
                 % Representation of a program.
  Program
* String.
                 % Name of a module in this program.
DELAY
          ModuleInProgram(x,_) UNTIL GROUND(x).
PREDICATE
          OpenModule :
  Program
                 % Representation of a program.
* String.
                 % Name of an open module in this program.
DELAY
           OpenModule(x,_) UNTIL GROUND(x).
PREDICATE DeclaredInOpenModule:
  Program
                 % Representation of a program.
                 \% Name of an open module in this program.
* String
* Formula.
                 % of this program whose proposition or predicate is
                 % declared in this module.
DELAY
          DeclaredInOpenModule(x, \_, z) UNTIL GROUND(x) & GROUND(z).
PREDICATE DeclaredInClosedModule:
                 % Representation of a program.
  Program
* String
                 % Name of a closed module in this program.
* Formula.
                 % Representation of an atom in the flat language of
                 \% this program whose proposition or predicate is
                 % declared in the export part of this module.
          DeclaredInClosedModule(x, \_, z) UNTIL GROUND(x) & GROUND(z).
DELAY
```

ProgramPredicateName(x,y,z,u,v) UNTIL GROUND(x) &

 $((GROUND(y) \& GROUND(z) \& GROUND(u)) \setminus / GROUND(v)).$

DELAY

PREDICATE MainModuleInProgram:

PREDICATE StatementInModule :

```
Program % Representation of a program.
```

- * String % Name of an open module in this program.
- * Formula. % Representation of a statement in this module.

DELAY StatementInModule(x,_,_) UNTIL GROUND(x).

PREDICATE StatementMatchAtom:

```
Program % Representation of a program.
```

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

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

PREDICATE DefinitionInProgram:

```
Program % Representation of a program.
```

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

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

PREDICATE ControlInProgram :

Program % Representation of a program.

- * String % Name of an open module in this program.
- * Name % Representation of the flat name of a predicate
 - % declared in this module.
- - % the heads of all DELAY declarations for this
 - % predicate.
- * List(Condition).% List of representations of the DELAY conditions
 - % corresponding to each head in the fourth argument.

DELAY ControlInProgram(x,_,_,_) UNTIL GROUND(x).

PREDICATE AndCondition:

Condition % Representation of a DELAY declaration condition,

% Cond1.

% Cond2.

PREDICATE OrCondition:

Condition % Representation of a DELAY declaration condition,

% Cond1.

% Cond2.

PREDICATE TrueCondition:

Condition. % Representation of the DELAY condition TRUE.

PREDICATE NonVarCondition:

Term % Representation of a variable, var.

* Condition. % Representation of the DELAY condition NONVAR(var).

PREDICATE GroundCondition:

Term % Representation of a variable, var.

* Condition. % Representation of the DELAY condition GROUND(var).

PREDICATE StringToCondition:

String % The string representation of a DELAY condition.

DELAY StringToCondition(x,_) UNTIL GROUND(x).

PREDICATE ConditionToString:

Condition % Representation of a DELAY condition.

% a canonical form).

DELAY ConditionToString(x,_) UNTIL GROUND(x).

PREDICATE BaseInModule:

- Program % Representation of a program.

 * String % Name of a module in this program.

% which cannot be LOCAL if this module is closed.

* Name % Representation of the flat name of a base % accessible to this part of this module.

* String. % Name of module in which this base is declared.

DELAY BaseInModule($x, _{-}, _{-}, _{-}$) UNTIL GROUND(x).

PREDICATE ConstructorInModule :

Program % Representation of a program.

* String % Name of a module in this program.

 $\mbox{\ensuremath{\mbox{\sc W}}}$ which cannot be LOCAL if this module is closed.

* Name % Representation of the flat name of a constructor

% accessible to this part of this module.

* Integer % Arity of this constructor.

* String. % Name of module in which this constructor is

% declared.

DELAY ConstructorInModule(x,_,_,_,) UNTIL GROUND(x).

PREDICATE ConstantInModule :

Program % Representation of a program.

* String % Name of a module in this program.

 $\mbox{\ensuremath{\mbox{\tiny M}}}$ which cannot be LOCAL if this module is closed.

* Name % Representation of the flat name of a constant

% accessible to this part of this module.

* Type % Representation of the type of this constant.

* String. % Name of module in which this constant is declared.

DELAY ConstantInModule(x,_,_,_,_) UNTIL GROUND(x).

PREDICATE FunctionInModule:

```
Program
                  % Representation of a program.
* String
                  % Name of a module in this program.
* ModulePart
                  % Representation of a part keyword of this module
                  \mbox{\ensuremath{\mbox{\tiny M}}} which cannot be LOCAL if this module is closed.
                  \% Representation of the flat name of a function
* Name
                  % accessible to this part of this module.
* FunctionInd
                  % Representation of the indicator for this function.
* List(Type)
                  % List of the representations of the domain types of
                  % this function.
                  % Representation of the range type of this function.

    Type

* String.
                  % Name of module in which this function is declared.
           FunctionInModule(x,_,_,_,_,) UNTIL GROUND(x).
DELAY
PREDICATE PropositionInModule:
  Program
                  % Representation of a program.
* String
                  % Name of a module in this program.
* ModulePart
                  % Representation of a part keyword of this module
                  % which cannot be LOCAL if this module is closed.
* Name
                  % Representation of the flat name of a proposition
                  % accessible to this part of this module.
* String.
                  % Name of module in which this proposition is
                  % declared.
           PropositionInModule(x,_,_,_,) UNTIL GROUND(x).
DELAY
PREDICATE PredicateInModule :
  Program
                  % Representation of a program.
                  % Name of a module in this program.
* String
* ModulePart
                  % Representation of a part keyword of this module
                  % which cannot be LOCAL if this module is closed.
                  % Representation of the flat name of a predicate
* Name
                  \% accessible to this part of this module.
                  % Representation of the indicator for this predicate.
* PredicateInd
* List(Type)
                  % List of the representations of the types for this
                  % predicate.
* String.
                  % Name of module in which this predicate is declared.
           PredicateInModule(x,_,_,_,_,) UNTIL GROUND(x).
DELAY
PREDICATE DelayInModule:
  Program
                  % Representation of a program.
```

% Name of a module in this program.

* String

```
* ModulePart
                  % Representation of a part keyword of this module
```

% which cannot be LOCAL if this module is closed.

* Formula % Representation of the Atom part of a DELAY

% declaration appearing in this part of this module.

% Representation of the Cond part of this DELAY * Condition.

% declaration.

DelayInModule(x,_,_,_,) UNTIL GROUND(x). DELAY

PREDICATE ImportInModule:

Program % Representation of a program.

* String % Name of a module in this program.

% Representation of a part keyword of this module * ModulePart

 $\mbox{\ensuremath{\mbox{\tiny M}}}$ which cannot be LOCAL if this module is closed.

* String. % Name of a module appearing in an IMPORT declaration

% in this part of this module.

ImportInModule(x,_,_,_) UNTIL GROUND(x). DELAY

PREDICATE LiftInModule:

% Representation of a program. Program

* String % Name of an open module in this program.

% Name of a module appearing in a LIFT declaration * String.

% in the local part of this module.

DELAY LiftInModule(x,_,_) UNTIL GROUND(x).

PREDICATE NewProgram:

String % A string.

* Program. % Representation of a program with this string as

% the name of the main module. If the name of the

% module is that of one of the system modules, then

% the program consists of that module and all the

% ones upon which it depends. Otherwise, the program

% is an empty one and the main module has both a

% local and an export part.

NewProgram(x,_) UNTIL GROUND(x). DELAY

PREDICATE InsertProgramBase :

Program % Representation of a program.

```
* String
                  % Name of an open user module in this program.
* ModulePart
                  % Representation of a part keyword of this module.
* Name
                  \% Representation of the flat name of a base not
                  \% declared in this part of this module.
* Program.
                  % Representation of a program which differs from
```

% the program in the first argument only in that it % also contains the declaration of this base in this

% part of this module.

InsertProgramBase(x,y,z,u,_) UNTIL GROUND(x) & GROUND(y) & DELAY GROUND(z) & GROUND(u).

PREDICATE DeleteProgramBase :

	Program	% Representation of a program.
*	String	% Name of an open user module in this program.
*	${ t ModulePart}$	% Representation of a part keyword of this module.
*	Name	% Representation of the flat name of a base declared
		% in this part of this module.
*	Program.	% Representation of a program which differs from the
		% program in the first argument only in that it does
		% not contain the declaration of this base in this
		% part of this module.

DeleteProgramBase(x,y,z,_,_) UNTIL GROUND(x) & GROUND(y) & DELAY GROUND(z).

PREDICATE InsertProgramConstructor:

	Program	%	Representation of a program.
*	String	%	Name of an open user module in this program.
*	${ t ModulePart}$	%	Representation of a part keyword of this module.
*	Name	%	Representation of the flat name of a constructor
		%	not declared in this part of this module.
*	Integer	%	Arity of this constructor.
*	Program.	%	Representation of a program which differs from the $$
		%	program in the first argument only in that it also
		%	contains the declaration of this constructor in
		%	this part of this module.

DELAY InsertProgramConstructor(x,y,z,u,v,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z) & GROUND(u) & GROUND(v).

PREDICATE DeleteProgramConstructor :

Program % Representation of a program.

% this part of this module.

PREDICATE InsertProgramConstant :

	Program	%	Representation of a program.
*	String	%	Name of an open user module in this program.
*	${ t ModulePart}$	%	Representation of a part keyword of this module.
*	Name	%	Representation of the flat name of a constant not
		%	declared in this part of this module.
*	Туре	%	Representation of the type of this constant.
*	Program.	%	Representation of a program which differs from the
		%	program in the first argument only in that it also
		%	contains the declaration of this constant in this
		%	part of this module.

PREDICATE DeleteProgramConstant :

```
Program
                   % Representation of a program.
* String
                   % Name of an open user module in this program.
                   % Representation of a part keyword of this module.
* ModulePart
                   % Representation of the flat name of a constant
* Name
                   % declared in this part of this module.
                   \mbox{\ensuremath{\mbox{\%}}} Representation of the type of this constant.
* Type
                   % Representation of a program which differs from the
* Program.
                   % = 1000 program in the first argument only in that it does
                   % not contain the declaration of this constant in this
                   % part of this module.
```

PREDICATE InsertProgramFunction:

```
Program
                  % Representation of a program.
* String
                  % Name of an open user module in this program.
* ModulePart
                  % Representation of a part keyword of this module.
* Name
                  \% Representation of the flat name of a function not
                  % declared in this part of this module.
* FunctionInd
                  \mbox{\ensuremath{\mbox{\%}}} List of the representations of the domain types of
* List(Type)
                  % this function.
                  % Representation of the range type of this function.
* Type
                  \mbox{\ensuremath{\mbox{\%}}} Representation of a program which differs from the
* Program.
                  % program in the first argument only in that it also
                  \% contains the declaration of this function in this
                  % part of this module.
           InsertProgramFunction(x,y,z,u,v,w,r,_) UNTIL GROUND(x) &
DELAY
                                  GROUND(y) & GROUND(z) & GROUND(u) &
```

GROUND(v) & GROUND(w) & GROUND(r).

PREDICATE DeleteProgramFunction :

e.
ion.
s of
ion.
the
does
n

PREDICATE InsertProgramProposition :

	Program	%	Representation of a program.
*	String	%	Name of an open user module in this program.
*	${ t ModulePart}$	%	Representation of a part keyword of this module.
*	Name	%	Representation of the flat name of a proposition
		%	not declared in this part of this module.
*	Program.	%	Representation of a program which differs from the $$
		%	program in the first argument only in that it also

```
\% contains the declaration of this proposition in \% this part of this module.
```

PREDICATE DeleteProgramProposition :

	Program	%	Representation of a program.
*	String	%	Name of an open user module in this program.
*	${ t ModulePart}$	%	Representation of a part keyword of this module.
*	Name	%	Representation of the flat name of a proposition
		%	declared in this part of this module.
*	Program.	%	Representation of a program which differs from the
		%	program in the first argument only in that it does
		%	not contain the declaration of this proposition in
		%	this part of this module.

PREDICATE InsertProgramPredicate :

	Program	%	Representation of a program.
*	String	%	Name of an open user module in this program.
*	${ t ModulePart}$	%	Representation of a part keyword of this module.
*	Name	%	Representation of the flat name of a predicate
		%	not declared in this part of this module.
*	${\tt PredicateInd}$	%	Representation of the indicator for this predicate.
*	${\tt List}({\tt Type})$	%	List of the representations of the types of this
		%	predicate.
*	Program.	%	Representation of a program which differs from the
		%	program in the first argument only in that it also
		%	contains the declaration of this predicate in this
		%	part of this module.

PREDICATE DeleteProgramPredicate :

Program	% Representation of a program.
* String	% Name of an open user module in this program.
* ModulePart	% Representation of a part keyword of this module.
* Name	% Representation of the flat name of a predicate
	% declared in this part of this module.

% predicate.

% program in the first argument only in that it does % not contain the declaration of this predicate in

% this part of this module.

DELAY DeleteProgramPredicate(x,y,z,_,_,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE InsertDelay:

Program % Representation of a program.

* String % Name of an open user module in this program.

* ModulePart % Representation of a part keyword of this module.

 $\mbox{\ensuremath{\mbox{\%}}}$ program in the first argument only in that it also

 $\mbox{\ensuremath{\%}}$ contains in this part of this module the DELAY

% declaration consisting of the Atom part in the

 $\mbox{\ensuremath{\mbox{\%}}}$ fourth argument and the Cond part in the fifth

% argument.

PREDICATE DeleteDelay:

Program % Representation of a program.

* String % Name of an open user module in this program.

* ModulePart % Representation of a part keyword of this module.

* Formula % Representation of the Atom part of DELAY

% declaration in this part of this module.

* Condition % Representation of the Cond part of this DELAY

% declaration.

 $\mbox{\ensuremath{\mbox{\%}}}$ program in the first argument only in that it does

% not contain this DELAY declaration in this part of

% this module.

DELAY DeleteDelay(x,y,z,_,_,) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE InsertStatement:

```
Program
                  % Representation of a program.
* String
                  % Name of an open user module in this program.
* Formula
                  % Representation of a statement in the flat language
                  % of this module wrt this program.
                  % Representation of a program which differs from the
* Program.
                  % program in the first argument only in that it also
                  % contains this statement in this module.
           InsertStatement(x,y,z,_) UNTIL GROUND(x) & GROUND(y) &
DELAY
                                                             GROUND(z).
PREDICATE DeleteStatement :
  Program
                  % Representation of a program.
* String
                  % Name of an open user module in this program.
* Formula
                  % Representation of a statement in the flat language
                  % of this module wrt this program appearing in this
                  % module.
* Program.
                  % Representation of a program which differs from the
                  % program in the first argument only in that it does
                  % not contain this statement in this module.
           DeleteStatement(x,y,\_,\_) UNTIL GROUND(x) & GROUND(y).
DELAY
PREDICATE InsertProgramImport :
  Program
                  % Representation of a program.
* String
                  % Name of an open user module in this program.
                  % Representation of a part keyword of this module.
* ModulePart
                  % Name of a module.
* String
* Program.
                  % Representation of a program which differs from the
                  % program in the first argument in that it also
                  \% contains in this part of this module the IMPORT
                  % declaration importing the module in the fourth
                  % argument.
                  % If the module named in the IMPORT declaration is not
                  % already in the program and it is not a system module,
                  % then an empty module with that name is added to the
                  % program; if the module named in the IMPORT
                  % declaration is a system module not already in the
                  \% program, then the system module is added to the
                  % program.
```

DELAY InsertProgramImport(x,y,z,u,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z) & GROUND(u).

PREDICATE DeleteProgramImport :

```
Program
                  % Representation of a program.
                  % Name of an open user module in this program.
* String
* ModulePart
                  % Representation of a part keyword of this module.
* String
                  % Name of a module.
* Program.
                  % Representation of a program which differs from the
                  % program in the first argument in that it contains
                  % one less IMPORT declaration importing the module in
                  % the fourth argument in this part of this module.
                  % If the IMPORT declaration deleted is the last import
                  \% declaration in the program containing the name of a
                  % particular module, then this module is deleted from
                  % the program.
DELAY
           DeleteProgramImport(x,y,z,_,_) UNTIL GROUND(x) & GROUND(y) &
                                                               GROUND(z).
PREDICATE InsertProgramLift :
  Program
                  % Representation of a program.
* String
                  % Name of an open user module in this program.
* String
                  % Name of a module.
                  % Representation of a program which differs from the
* Program.
                  % program in the first argument in that it also
                  % contains in the local part of this module the LIFT
                  % declaration importing the module in the third
                  % argument.
                  %
                  \mbox{\ensuremath{\mbox{\%}}} If the module named in the LIFT declaration is not
                  % already in the program and it is not a system module,
                  \% then an empty module with that name is added to the
                  % program; if the module named in the LIFT declaration
                  % is a system module not already in the program, then
                  % the system module is added to the program.
           InsertProgramLift(x,y,z,_) UNTIL GROUND(x) & GROUND(y) &
DELAY
                                                                GROUND(z).
PREDICATE DeleteProgramLift:
  Program
                  % Representation of a program.
* String
                  % Name of an open user module in this program.
* String
                  % Name of a module.
```

```
* Program.

* Representation of a program which differs from the

* program in the first argument in that it contains

* one less LIFT declaration importing the module in

* the third argument in the local part of this module.

*

* If the LIFT declaration deleted is the last import

* declaration in the program containing the name of a

* particular module, then this module is deleted from

* the program.
```

DELAY DeleteProgramLift(x,y,_,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE RunnableAtom:

Program % Representation of a program.

* Formula. % Representation of an atom in the flat language of % this program which has a user-defined predicate and % is not delayed (according to any DELAY declarations % for the predicate).

DELAY RunnableAtom(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE Succeed:

Program % Representation of a program.

* Formula % Representation of the body of a goal in the flat

% language of this program.

* TermSubst. % Representation of a computed answer for this goal

% and the flat form of this program.

DELAY Succeed(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE Compute:

% Representation of a program. Program % Representation of the body of a goal in the flat * Formula % language of this program. * Integer % A non-negative integer. * Integer % A non-negative integer. * TermSubst % Representation of a term substitution. * TermSubst % Representation of the term substitution obtained by % composing the term substitution in the fifth % argument with a computed answer for the goal whose % body is obtained by applying the term substitution % in the fifth argument to the body in the second % = 1000 argument and the flat form of this program or, if

```
% the computation flounders, the representation of the
                  % = 1000 answer computed up to the step at which the
                  % computation floundered. Indexes of any new variables
                  \% in the terms to which variables in the goal are bound
                  \% in the computed answer are greater than or equal to
                  % the third argument and strictly less than the fourth
                  % argument.
                  % Representation of the body of the last goal in the
* Formula.
                  \% derivation (which is empty if the derivation
                  % succeeded and is non-empty if the derivation
                  % floundered).
DELAY
           Compute(x,y,z,_,w,_,_) UNTIL GROUND(x) & GROUND(y) &
                                                  GROUND(z) & GROUND(w).
PREDICATE SucceedAll:
                  % Representation of a program.
  Program
* Formula
                  % Representation of the body of a goal in the flat
                  % language of this program.
* List(TermSubst). % List of representations of computed answers for
                  % this goal and the flat form of this program.
           SucceedAll(x,y,_) UNTIL GROUND(x) & GROUND(y).
DELAY
PREDICATE ComputeAll:
                  % Representation of a program.
  Program
                  % Representation of the body of a goal in the flat
* Formula
                  % language of this program.
* Integer
                  % A non-negative integer.
* Integer
                  % A non-negative integer.
* TermSubst
                  % Representation of a term substitution.
* List(TermSubst) % List of representations of all term substitutions
                  % obtained by composing the term substitution in the
                  % fifth argument with a computed answer for the goal
                  % whose body is obtained by applying the term
                  % substitution in the fifth argument to the body in the
                  % second argument and the flat form of this program or,
                  % for computations which end in a flounder, the
                  % representation of the answer computed up to the step
                  % at which the computation floundered. Indexes of any
                  \% new variables in the terms to which variables in the
                  % goal are bound in the computed answers are greater
                  % than or equal to the third argument and strictly less
```

% than the fourth argument.

```
\mbox{\ensuremath{\%}} goals in each of the corresponding derivations (which
```

% are empty if the derivation succeeded and are

 $\mbox{\ensuremath{\mbox{\%}}}$ non-empty if the derivation floundered).

DELAY ComputeAll(x,y,z,_,w,_,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z) & GROUND(w).

PREDICATE Fail:

Program % Representation of a program.

 $\mbox{\ensuremath{\mbox{\%}}}$ language of this program such that this goal and the

 $\mbox{\ensuremath{\mbox{\%}}}$ flat form of this program have a finitely failed

% search tree.

DELAY Fail(x,y) UNTIL GROUND(x) & GROUND(y).

Bibliography

- [1] K.R. Apt. Arrays, bounded quantification and iteration in logic and constraint programming. In *Proc. of GULP-PRODE'95* Salerno, Italy, 1995.
- [2] K.R. Apt and H.C. Doets. A new definition of SLDNF-resolution. *Journal of Logic Programming*, 18(2):177–190, 1992.
- [3] H. Arro, J. Barklund, and J. Bevemyr. Parallel bounded quantification—preliminary results. *ACM SIGPLAN Notices*, 28(5):117–124, August 1993.
- [4] H Barklund and A Hamfelt. Hierarchical representation of legal knowledge with meta-programming in logic. *Journal of Logic Programming*, 18(1):55–80, 1994.
- [5] J. Barklund. Bounded quantifications for iterations and concurrency in logic programming. New Generation Computing, 12(2), 1994.
- [6] J. Barklund. Metaprogramming in logic. In A. Kent and J.G. Williams, editors, Encyclopedia of Computer Science and Technology, volume 33, pages 205–227. Marcel Dekker, 1995. Also available as UPMAIL Technical Report No. 80.
- [7] J. Barklund and J. Bevemyr. Executing bounded quantifications on shared memory multiprocessors. In Jaan Penjam, editor, *Proc. Intl. Conf. on Programming Language Implementation and Logic Programming 1993*, LNCS 714, pages 302–317, Berlin, 1993. Springer-Verlag.
- [8] J. Barklund and J. Bevemyr. Prolog with arrays and bounded quantification. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, pages 28 39. Springer-Verlag, 1993. Proceedings of the 4th International Conference, LPAR'93.
- [9] J. Barklund, K. Boberg, P. Dell'Aqua, and M. Veanes. Meta-programming with theory systems. In K.R. Apt and F. Turini, editors, Meta-Logics and Logic Programming, chapter 8, pages 195–224. MIT Press, 1995.

- [10] J. Barklund and H. Millroth. Integrating complex data structures in Prolog. In 1987 Symposium on Logic Programming, pages 415–425, 1987.
- [11] K.A. Bowen and R.A. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.
- [12] K.A. Bowen and T. Weinberg. A meta-level extension of Prolog. In *Proceedings of 2nd IEEE Symposium on Logic Programming*, Boston, pages 669–675. Computer Society Pres, 1985.
- [13] A. Bowers and P. M. Hill. An introduction to Gödel. In K. Broda, editor, Proceedings of the ALPUK92 Conference. Springer-Verlag, 1992. Technical Report 92.06, School of Computer Studies, University of Leeds.
- [14] A.F. Bowers. Representing Gödel object programs in Gödel. Technical Report CSTR-92-31, Department of Computer Science, University of Bristol, 1992.
- [15] A.F. Bowers and C.A. Gurr. Towards fast and declarative meta-programming. In K.R. Apt and F. Turini, editors, *Meta-Logics and Logic Programming*, chapter 6, pages 137–166. MIT Press, 1995.
- [16] A.F. Bowers, P.M. Hill, and F. Ibañez. Resolution for logic programming with universal quantifiers. In *PLILP 97*. Springer-Verlag, 1997.
- [17] I. Bratko. Prolog Programming for Artificial Intelligence. Addison-Wesley, 1986.
- [18] A. Brogi and S. Contiero. Composing logic programs by meta-programming in Gödel. In K.R. Apt and F. Turini, editors, *Meta-Logics and Logic Programming*, chapter 7, pages 167–193. MIT Press, 1995.
- [19] I. Cervesato and G. F. Rossi. Logic meta-programming facilities in 'LOG. In A. Petterossi, editor, Proceedings of the Third Workshop on Meta-programming in Logic, Uppsala, Sweden, pages 148–161. Springer-Verlag, 1992.
- [20] H. Christiansen. Efficient and complete demo predicates for definite clause languages. Technical Report 51, Computer Science Department, Roskilde University, 1994.
- [21] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

- [22] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un systemes de communication homme-machine en Français. Technical report, Groupe d'Intelligence Artificialle, University d'Aix Marseille II, Luminy, France, 1973.
- [23] S. Costantini and G. Lanzarone. A metalogic programming language. In G. Levi and M. Martelli, editors, 6th International Conference on Logic Programming, Lisbon, pages 218–233. MIT Press, 1989.
- [24] D.A. de Waal. Analysis and Transformation of Proof Procedures. PhD thesis, University of Bristol, Dept. of Computer Science, October 1994.
- [25] S.K. Debray. Compile-time syntax checking of Janus is undecidable. Unpublished Note, 1994.
- [26] H. B. Enderton. A Mathematical Introduction to Logic. Academic Press, 1972.
- [27] S. Feferman. Transfinite recursive progressions of axiomatic theories. The Journal of Symbolic Logic, 27(3):259–316, September 1962.
- [28] J. Gallagher. Transforming logic programs by specializing interpreters. In *ECAI-86*, Brighton, pages 109–122, 1986.
- [29] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatsh. Math. Phys., 38:173–98, 1931.
- [30] A. Guessoum and J.W. Lloyd. Updating knowledge bases. New Generation Computing, 8(1):71–89, 1990.
- [31] A. Guessoum and J.W. Lloyd. Updating knowledge bases ii. Technical Report TR-90-13, Department of Computer Science, University of Bristol, May 1990.
- [32] C.A. Gurr. A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel. PhD thesis, Dept. of Computer Science, University of Bristol, 1993.
- [33] C.A. Gurr. Specialising the ground representation in the logic programming language Gödel. In Y. Deville, editor, *Proceedings of LOPSTR 93*. Springer-Verlag, 1993.
- [34] M. Hanus (ed.). Curry: An integrated functional logic language. Available at http://www-i2.informatik.rwth-aachen.de/~hanus/curry.
- [35] J van Heijenoort. From Frege to Gödel: a source book in Mathematical Logic, 1879-1931. Harvard University Press, Cambridge, Mass, 1967.

- [36] P. M. Hill and J. Gallagher. Meta-programming in logic programming. In D Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 5. Oxford University Press, 1996.
- [37] 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.
- [38] 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*, pages 23–52. MIT Press, 1989. Proceedings of the Meta88 Workshop, June 1988.
- [39] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
- [40] 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.
- [41] 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.
- [42] N. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Software Generation*. Prentice Hall, 1993.
- [43] M. Kalsbeek. Correctness of the vanilla interpreter and ambivalent syntax. In K.R. Apt and F. Turini, editors, *Meta-Logics and Logic Programming*, chapter 1, pages 3–26. MIT Press, 1995.
- [44] M. Kalsbeek and Y. Jiang. A vademecum of Ambivalent Logic. In K.R. Apt and F. Turini, editors, *Meta-Logics and Logic Programming*, chapter 2, pages 27–81. MIT Press, 1995.
- [45] R. A. Kowalski. Predicate logic as a programming language. In *Information Processing* 74, pages 569–574, Stockholm, 1974. North Holland.
- [46] R. A. Kowalski. Problems and promises of computational logic. In J. W. Lloyd, editor, *Computational Logic*, pages 1–36. Springer-Verlag, 1990.
- [47] M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. Report CW210, Katholieke Universiteit Leuven, April 1995.

- [48] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, January 1995.
- [49] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.
- [50] J.W. Lloyd. Foundations of Logic Programming. Springer-Verlag, 2nd edition, 1987.
- [51] J.W. Lloyd. Practical advantages of declarative programming. In Proceedings of the Joint Conference on Declarative Programming, GULP-PRODE'94, 1994.
- [52] J.W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1997. To appear.
- [53] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3&4):217–242, 1991.
- [54] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
- [55] B. Martens and D. De Schreye. A perfect Herbrand semantics for untyped vanilla meta-programming. In K. Apt, editor, *Proceedings of the Joint International Conference on Logic Programming*, Washington, USA, pages 511–525, 1992.
- [56] B. Martens and D. De Schreye. Why untyped non-ground metaprogramming is not (much of) a problem. Technical Report CW 159, Department of Computer Science, Katholieke Universiteit Leuven, 1992. An abridged version will published in the Journal of Logic Programming.
- [57] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3:184–195, 1960.
- [58] E. Mendelson. Introduction to Mathematical Logic. 3rd Edition, Van Nostrand, 1987.
- [59] E Moggi. Computational lambda-calculus and monads. In *Proceedings* of the Logic in Computer Science Conference, 89.
- [60] A. Mulkers. Live Data Structures in Logic Programs. Lecture Notes in Computer Science 675. Springer-Verlag, 1993.

- [61] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. Artificial Intelligence, 23:295–307, 1984.
- [62] L. Naish. Negation and quantifiers in NU-Prolog. In E. Shapiro, editor, Proceedings of the Third International Conference on Logic Programming, London, pages 624-634. Lecture Notes in Computer Science 225, Springer-Verlag, 1986.
- [63] R.A. O'Keefe. The Craft of Prolog. MIT Press, 1990.
- [64] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.
- [65] L.M. Pereira, F. Pereira, and D.H.D. Warren. User's guide to DECsystem-10 Prolog. Technical report, Department of A.I., University of Edinburgh, September 1978.
- [66] J. Peterson and K. Hammond (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.4). Available at http://haskell.org/.
- [67] S.L. Peyton Jones and J. Launchbury. State in haskell. Lisp and Symbolic Computation, 8(4):293–341, December 1995.
- [68] S.L. Peyton Jones and D. Lester. A modular fully-lazy lambda lifter in Haskell. *Software Practice and Experience*, 21(5):479–506, May 1991.
- [69] S.L. Peyton Jones and E. Meijer. Henk, a typed intermediate language. In R. Harper and R. Muller, editors, ACM Workshop on Types in Compilation, June 1997.
- [70] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
- [71] V.A. Saraswat, K. Kahn, and J. Levy. Janus: A step towards distributed constraint programming. In S. Debray and M. Hermenegildo, editors, Logic Programming: Proceedings of the 1990 North American Conference, pages 421–446, 1990.
- [72] T. Sato and H. Tamaki. First order compiler: A deterministic logic program synthesis algorithm. *Journal of Symbolic Computation*, 8:605–627, 1988.
- [73] L. Sterling and E. Shapiro. The Art of Prolog. MIT Press, 1986.
- [74] V.S. Subrahmanian. A simple formulation of the theory of metalogic programming. In H.D. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 65–102. MIT Press, 1989. Proceedings of the Meta88 Workshop, June 1988.

- [75] Swedish Institute of Computer Science. SICStus Prolog User's Manual, release 2.1 # 6 edition, 1992.
- [76] Swedish Institute of Computer Science. SICStus Prolog User's Manual, release 3 # 0 edition, 1995.
- [77] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta programming. In H.-J. Kugler, editor, *Information Processing 86*, Dublin, pages 415–420. North Holland, 1986.
- [78] R. D. Tennent. Semantics of Programming Languages. Prentice-Hall International, 1991.
- [79] J.A. Thom and J. Zobel. NU-Prolog reference manual, version 1.3. Technical Report TR 86/10, Machine Intelligence Project, Department of Computer Science, University of Melbourne, 1988.
- [80] A. M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc.*, 42:230–265, 1936.
- [81] M. van Emden. AVL tree insertion: a benchmark program biased towards Prolog. *Logic Programming Newsletter*, 2, 1981.
- [82] F. van Harmelen. Definable naming relations in metalevel systems. In A. Pettorossi, editor, *Meta-Programming in Logic*, *Proceedings of the 3rd International Workshop*, *META-92*. Springer-Verlag, 1992.
- [83] A. Voronkov. Logic programming with Bounded Quantifiers. Journal of Logic Programming, to appear.
- [84] A. Voronkov. Logic programming with bounded quantifiers. In A. Voronkov, editor, Logic Programming 2nd Russian Conf. on Logic Programming, pages 486–514. Springer-Verlag, 1992.
- [85] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*, Munich, Germany, January 1987.
- [86] D.H.D. Warren. An abstract PROLOG instruction set. Technical Note 309, SRI International, 1983.
- [87] R.W. Weyhrauch. An example of FOL using Metatheory. Formalizing reasoning systems and introducing derived inference rules. In *Proceedings of the 6th Conference on Automatic Deduction*, 1982.
- [88] N. Wirth. Algorithms + Data Structures = Programs. Prentice-Hall, 1976.

[89] F. Henderson Z. Somogyi and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, October-December 1996.