The Gödel Programming Language

P.M. Hill[1] and J.W. Lloyd

October 1992 CSTR-92-27
(Revised May 1993)

Department of Computer Science
University of Bristol
University Walk
Bristol BS8 1TR

[1]School of Computer Studies, University of Leeds, Leeds, LS2 9JT

Preface

Would that I had unknown utterances and strange phrases,
In a new language that does not pass away,
Free from repetition,
Without a phrase of familiar speech which the ancestors spoke.

From *The Words of Khakheperreseneb*, around 1890 B.C.

Gödel is a declarative, general-purpose programming language in the family of logic programming languages. It is a strongly typed language, the type system being based on many-sorted logic with parametric polymorphism. It has a module system. Gödel supports infinite precision integers, infinite precision rationals, and also floating-point numbers. It can solve constraints over finite domains of integers and also linear rational constraints. It supports processing of finite sets. It also has a flexible computation rule and a pruning operator which generalises the commit of the concurrent logic programming languages. Considerable emphasis is placed on Gödel's meta-logical facilities which provide significant support for meta-programs that do analysis, transformation, compilation, verification, debugging, and so on. The declarative nature of Gödel makes it particularly suitable for use as a teaching language; narrows the gap which currently exists between theory and practice in logic programming; makes possible advanced software engineering tools such as declarative debuggers and compiler generators; reduces the effort involved in providing a parallel implementation of the language; and offers substantial scope for parallelization in such implementations.

This book describes the Gödel language and is divided into two parts. The first part gives an informal overview of the language and includes example programs. The second part contains a definition of the syntax and semantics of the language. We assume readers have some programming experience, have some acquaintance with logic programming concepts, and are familiar with the most basic material on the syntax and semantics of first order logic. We provide an appendix on polymorphic many-sorted logic for readers not familiar with this generalisation of (unsorted) first order logic. At the end of the first chapter, we give, for those familiar with Prolog, a comparison between Gödel and Prolog. This provides much of the motivation for Gödel and explains why various design decisions were taken. We suggest readers follow the first nine chapters in order, looking ahead to the example programs in chapter 10 and referring to the formal definitions in the second part, as appropriate. Augmented by the reference manual of a particular implementation, the book contains everything a programmer needs to know to write large-scale Gödel programs.

Readers will notice that little is said about the procedural semantics of Gödel in this book. This is a decision which probably requires some explanation, since one might expect a book which gives a formal definition of a programming language to be rather precise on this point. First, for the syntax and declarative semantics, we have tried to be precise and comprehensive. The main criticism that could be made of our description of the declarative semantics is that the specification of the system predicates is given in English rather than some formal language. Our chosen method achieves conciseness, but at the expense of some lack of precision. In contrast, for the procedural semantics, we have adopted a different approach and merely specified that the implementation be sound and satisfy some other conditions. (For example, the conditions under which a `DELAY` declaration will delay a call are specified, as is the pruning effect of the commit operator.)

The main advantage of specifying the procedural semantics in great detail is portability, so that one could port a program from one implementation to another and be confident of identical behaviour of the program in the two systems. In spite of this, we have decided not to follow this approach. The reason is that logic programming language implementation is currently in a state of rapid development, so that whatever precise procedural semantics we might specify now is likely to look outdated and unnecessarily conservative in a couple of years. By leaving the details of the procedural semantics so open, we hope to encourage, or at least not impede, this development. This approach gives implementors of Gödel considerable freedom and, for example, admits theorem proving methods which provide a more flexible handling of negation than the usual SLDNF-resolution. However, the reference manual of an implementation will need to discuss whatever further details on the procedural semantics are required to write (reasonably) efficient programs.

The most likely difference between implementations of Gödel will be that programs terminating on one implementation may flounder on another. The two places where this is most likely to happen are with constraint solving and the handling of negation. This is because a considerable range of sophistication of constraint solvers for the integers and rationals is possible. Also, an implementation employing safe negation may flounder on calls which would run on an implementation employing a more flexible implementation of negation. However, only completeness is affected: all implementations must be sound.

It is hardly possible to design a new programming language without relying to a great extent on previous work. Gödel is no exception to this. First, we owe an indirect debt to Alain Colmerauer who designed the first Prolog language, many features of which are clearly evident in Gödel. We have also borrowed good ideas from more recent logic programming languages. In particular, we have adopted the if-then-else construct and the `when` declarations of NU-Prolog. Both of these facilities are due to Lee Naish. We drew inspiration from languages, such as ML and Modula-2, which demonstrate the clear necessity and value of type systems and module systems.

We owe a particular debt to Antony Bowers and Jiwei Wang, who have undertaken so successfully the substantial and demanding task of implementing the language. Corin Gurr helped with the implementation of the meta-programming modules. Also his work on self-applicable partial evaluators influenced the design of the language. Antony Bowers contributed to the design, especially of the meta-programming facilities, and introduced the concept of type lifting. Alistair Burt, who spent 12 months on the project, produced the first implementation of a subset of the language and also contributed to its design, especially the idea that the ground representation used by the meta-programming facilities should be handled like an abstract data type. Andrea Domenici contributed to an early implementation of the parser. Frank Defoort experimented at a formative stage with writing interpreters using the ground representation.

Various people commented on drafts of this book. These include Antony Bowers, Frank Defoort, Andrea Domenici, John Gallagher, Corin Gurr, Feliks Kluzniak, Micha Meier, Lee Naish, Vitor Santos Costa, Jeffrey Schultz, Zoltan Somogyi, Rodney Topor, and Jiwei Wang.

The first author is indebted to the Division of Artificial Intelligence (and in particular Tony Cohn) at the University of Leeds for encouraging her work on Gödel. The second author would like to thank many people, too numerous to mention, for many interesting discussions over the last decade, which have helped shape the form that Gödel has finally taken.

We are also indebted to Dr. Richard Parkinson of the Department of Egyptian Antiquities at the British Museum for the use of his translation of the above passage from the Words of Khakheperreseneb, which is preserved on a wooden writing tablet dating from the 18th Dynasty and is now in the British Museum (EA 5645). Khakheperreseneb is said to have been a priest of the city of Heliopolis. His name indicates that the text cannot predate the reign of Sesostris II (1895–1878 B.C.) in the Late Middle Kingdom.

The design and implementation of Gödel was partly supported by the ESPRIT Basic Research Action 3012 (Compulog) and Project 6810 (Compulog 2), SERC Grants GR/F/26256 and GR/H/79862, the University of Bristol, and the University of Leeds.

For details on obtaining an implementation of Gödel by ftp, send a message to `goedel@compsci.bristol.ac.uk`

May 1993

PMH
JWL

Contents

Part I

Overview of Gödel

Chapter 1

Introduction

In this chapter, we introduce briefly the main ideas of declarative programming, outline the main facilities of Gödel[1], and give a comparison of Gödel and Prolog.

1.1 Declarative Programming

Before getting down to the details of the Gödel language, we present a brief discussion of the general principles of declarative programming. The starting point for the programming process is the particular problem that the programmer is trying to solve. The problem is then formalised as an interpretation (called the *intended* interpretation) of a language which we assume here to be a (polymorphic many-sorted) first order language. (See appendix A.) The intended interpretation specifies the various domains and the meaning of the symbols of the language in these domains. In practice, the intended interpretation is rarely written down precisely, although in principle this should always be possible.

Now, of course, it is taken for granted here that it *is* possible to capture the intended application by a first order interpretation. Not all applications can be (directly) modelled this way and for such applications other languages and formalisms may have to be employed. However, a very large class of applications can be modelled naturally by means of a first order interpretation. In fact, this class is larger than is sometimes appreciated. For example, it might be thought that such an approach cannot (directly) model situations where a knowledge base is changing over time. Now it is true that the intended interpretation of the knowledge base is changing. However, the knowledge base should properly be regarded as data to various meta-programs, such as query processors or assimilators. Thus the knowledge base can be accessed and changed by these meta-programs. The meta-programs themselves have fixed intended interpretations which fits well with the setting for declarative programming given above.

Based on the intended interpretation, the *logic component* of a program is then written. This logic component is a particular kind of (polymorphic many-sorted) first order theory which is usually suitably restricted so as to admit an efficient theorem proving procedure. It is crucial that the intended interpretation be a model for the logic component of the program. This is because computed answers, which must be guaranteed to be correct by the implementation,

[1]The name Gödel is appropriate since the key concept of the language's approach to meta-programming is representation (i.e. naming), which was introduced by Gödel in his incompleteness result. Alternatively, Gödel can be regarded as an acronym formed from God's Own DEclarative Language.

therefore give instantiated goals which are true in the intended interpretation. Ultimately, the programmer is interested in *computing truth in the intended interpretation.*

Typically, in logic programming languages, the logic component of a program is the theory obtained by completing the collection of program statements, that is, by adding to the program the only-if halves of the statements in the program. (See appendix A.) In other approaches to declarative programming, different logics are used. For example, in functional programming, the logic component of a program can be understood to be a collection of formulas in the λ-calculus. However, while the logic used is different to that in logic programming, the underlying ideas of declarative programming from the functional programming perspective are the same.

Having written a correct logic component of a program, the programmer then turns to the *control component* of the program, which is concerned with the construction and pruning of search trees. Typically, the computation rule (which selects a literal in a goal for expansion) is partly specified by control declarations and the pruning of a search tree is specified by pruning operators. To some extent, it is possible to relieve the programmer of responsibility for the control aspects by the use of preprocessors which automatically generate appropriate control. Note that, in practice, control considerations may cause a programmer to go back and rewrite the logic component to improve the efficiency of a program. In any case, an important requirement of the control declarations and pruning operators of a program is that they can be stripped away and what remains is a correct logic component of the program.

Declarative programming can be understood in two main senses. In the weak sense, declarative programming means that programs are theories, but that a programmer may have to supply control information to produce an efficient program. Declarative programming, in the strong sense, means that programs are theories and all control information is supplied automatically by the system. In other words, for declarative programming in the strong sense, the programmer only has to provide the intended interpretation (or, perhaps, a theory which has the intended interpretation as a model). This is to some extent an ideal which is probably not attainable (nor, in some cases, totally desirable), but it does at least provide a challenging and appropriate target. Gödel itself is a contribution to declarative programming in the weak sense. Thus Gödel programs are (with some minor caveats) theories, but programmers are largely responsible for providing suitable control information.

In summary, declarative programming is much more concerned with writing down *what* should be computed and much less with *how* it should be computed. Ideally, in many circumstances, we would like to be able to dispense altogether with having to write down the "how" part (that is, the control part), but the current state of programming technology, including Gödel, does not permit this. Nevertheless, declarative programming in the strong sense would be a major contribution to programming if it could be substantially achieved and fully justifies the considerable effort which continues to be put into this challenge by research groups all over the world.

1.2 Gödel Facilities

Gödel is a declarative, general-purpose programming language in the family of logic programming languages. Its main facilities are as follows:

- types
- modules

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

We now introduce each of these facilities in turn.

First we consider types. The reasons for having types in logic programming languages are well known. The major reason is for knowledge representation. Intended interpretations in most logic programming applications are typed and hence using a typed language is the most direct way of capturing the relevant knowledge in the application. Also, the information given by the language declarations of a type system can be used by a compiler to produce more efficient code. Furthermore, type declarations can help to catch programming errors. For example, in an untyped language, simple typographical errors often lead to bizarre program behaviour which can usually only be identified by laborious tracing of the program. In contrast, in a typed language, such errors often lead to syntax errors which can be caught by the compiler. It is nearly always easier to correct an error caught by the compiler than it is to discover and correct an error that leads to wrong program behaviour. Our experience with the Gödel type system supports the contention that it greatly decreases the effort required for program development and greatly increases the likelihood of correctness of programs compared with untyped languages.

There are two kinds of type systems studied in logic. The first kind are type systems suitable for higher order logics (for example, the type system of typed λ-calculus). These provide the foundation for the type systems of functional languages and higher order logic programming languages. The other kind of type system studied in logic arises from many-sorted (first order) logic. It is many-sorted logic which provides the foundation for type facilities in first order logic programming languages and it is an extension of this logic that is used by Gödel. In the following, to conform to the usual terminology of programming languages, we often refer to a sort as a type.

However, a many-sorted logic alone is not sufficiently flexible for a type system in a logic programming language. The reason is that we want to write predicates which take a variety of types of arguments. For example, the usual `Append` predicate will normally be required to append lists each of whose elements has the same fixed but arbitrary type. For this reason, Gödel allows a form of polymorphism, called parametric polymorphism, familiar from functional programming languages. Parametric polymorphism necessitates the introduction of type variables, which range over all types. By this means, a polymorphic version of `Append`, for example, can be written in a similar way to the functional languages.

Gödel's type system is a strongly typed one, in the sense that each constant, function, proposition, and predicate in a program and its type must be specified by a language declaration. By contrast, variables have their types inferred from their context.

Next we briefly discuss Gödel's module system. The usual software engineering advantages of a module system are well known and they apply equally well to Gödel. In its most basic form, a module system simply provides a way of writing large programs so that various pieces of the program don't interfere with one another because of name clashes and also provides a way of hiding implementation details. The Gödel module system is based on these standard ideas. A program is a collection of modules. A module generally consists of two parts, the export part and the local part. The export part of a module consists of language declarations for the symbols available for use in either part of the module and in other modules which import them, control declarations, and module declarations giving the symbols from other modules which are available for use in either part of the module and in other modules which import them. The local part of a module consists of language declarations for the symbols available for use only in this part, module declarations giving the symbols from other modules available for use in this part, control declarations, and definitions of those propositions and predicates having language declarations in the module.

There is a rich collection of system modules – 20 in all – provided by the Gödel language. These include modules which process numbers, modules which process lists, strings, and sets, modules which provide input/output, and modules which provide meta-programming facilities. Gödel also makes significant use of abstract data types, which are implemented by means of the module and type systems. An abstract data type consists of a type and a collection of operations on the type. The details of the way the type is represented and the way the operations are implemented are hidden from programmers. Abstract data types have many advantages for software engineering, including the provision of a higher level of abstraction for programmers and the ability to change the implementation of a type without affecting existing code. The Gödel system modules provide a number of important abstract data types, including `String` (the type of strings), `Unit` (the type of term-like data structures), `Program` (the type of terms representing Gödel programs), and `Theory` (the type of terms representing many-sorted first order theories). In each case, the corresponding module provides a collection of operations on the type and the module system is used to hide the implementation details of these operations. Furthermore, a number of other Gödel system modules provide types which are at least partly abstract. For example, the implementation of the operations on the types `Integer` and `Rational` are hidden from programmers. However, for pragmatic reasons, it is preferable to make explicit the symbols used to represent these types. For example, the constants representing the integers are made available to programmers. It would be possible to avoid this by having a predicate which returned the integer `0` and a predicate which returned the successor of an integer, but this is much more cumbersome for programmers than simply making the constants available. In any case, a good way to think about the Gödel language is that essentially it provides a rich collection of (mostly abstract) data types, useful for a variety of applications, and provides facilities for programmers to define their own abstract data types.

Next we discuss Gödel's control facilities. Gödel has a flexible computation rule, which may select a literal other than the leftmost literal. The advantages of a flexible computation rule are that it can be used to ensure safeness (especially of negative calls), assist termination, assist efficiency, solve constraints, and control pruning. The computation rule is partly specified by means of `DELAY` control declarations, which cause certain calls to be delayed until they are sufficiently instantiated. Many Gödel system predicates have `DELAY` declarations. Furthermore, a programmer can encourage co-routining behaviour by means of these declarations.

Gödel has constraint-solving capabilities in the domains of integers and rationals. It can solve systems of (not necessarily linear) constraints which involve integers, variables which range over bounded intervals of integers, and the usual functions and predicates with integer arguments. It can also solve systems of linear rational constraints involving rationals, variables ranging over the rationals, and the usual functions and predicates with rational arguments.

We now discuss another control facility, which is the Gödel pruning operator called the commit. Consider the definition

```
P <- {Q}_1 & R & T.
P <- {S}_1 & T.
P <- {U}_2 & V.
```

in which `P`, `Q`, and so on, are propositions. The commit notation has the form `{...}_l`, where `l` is a label (which is a positive integer). The brackets `{...}` capture the scope of the commit inside a statement. The label captures the scope of the commit over the statements in the definition. When a conjunction `{L1 & ... & Ln}_l` succeeds, all other statements in the definition which contain a commit labelled `l` are pruned. The other pruning that takes place is that only one solution is found for `L1 & ... & Ln`. Thus, if `P` is called and then `Q` subsequently succeeds, the second statement (but not the third) will be pruned and only one solution for `Q` will be found. Two special cases of the commit are provided. These are the bar commit, which is similar to the commit of the concurrent languages, and the one solution operator. In fact, programmers generally have no need for the general commit and can do all the explicit pruning they want to do with the bar commit and one solution operator. The general commit is primarily meant for source-level tools, such as partial evaluators and program transformers. Furthermore, commits are rarely necessary in Gödel programs, since they can often be replaced by declarative constructs, such as negation and if-then-else.

The commit operator provides a powerful tool for pruning away unwanted branches of search trees, but it can affect the declarative semantics of programs in the sense that correct answers can be pruned. Therefore, it detracts from the major aim of Gödel, which is to have programs with the best possible declarative semantics. However, as was argued in detail in [12], (sound) pruning operators *do* have an important place in logic programming languages, even though they may adversely affect the semantics of programs in which they appear. Furthermore, in Gödel, it is always possible to strip away all commits from a program to reveal its underlying logic component. In addition, the Gödel commit is sound and has been shown to have excellent procedural semantics. For example, a theorem in [12] shows that commit is well-behaved under partial evaluation in the sense that, under reasonable conditions, the original program can be arranged to have the same set of computed answers as the partially evaluated program.

For an arbitrary computation rule, the pruning done by the commit, while correct according to its definition, may be very difficult for a programmer to understand and control. For example, if an insufficiently instantiated call is allowed to proceed, the system may commit to a statement which is not intended and, as a consequence, the computation may later fail. Thus an important use of `DELAY` declarations is to control pruning. Appropriate `DELAY` declarations ensure calls only proceed if they are sufficiently instantiated, thus avoiding unexpected failure and, more generally, giving the programmer sufficient control over what is pruned.

The other control facility provided by Gödel is the `IF-THEN-ELSE` construct, which has several variations. This construct has a declarative meaning, but also provides important control information in that it ensures the condition in the `IF` part is only computed once. This can result in an important saving if this condition is computationally expensive. One variation omits the `ELSE` part and another allows existentially quantified variables to appear in the `IF` and `THEN` parts. As the reader will see from the programs in this book, the `IF-THEN-ELSE` construct figures prominently in the Gödel programming style.

Next we turn to Gödel's meta-programming facilities. The essential characteristic of meta-programming is that a meta-program is a program which uses another program (the object program) as data. Meta-programming techniques underlie many of the applications of logic programming. For example, knowledge base systems consist of a number of knowledge bases (the object programs), which are manipulated by interpreters and assimilators (the meta-programs). Other important kinds of software, such as debuggers, compilers, and program transformers, are meta-programs.

The Gödel approach to meta-programming is to exploit abstract data types. One of the main such types is `Program` which is the type of a term representing an object Gödel program. Terms of type `Program` are complicated terms which contain a representation of all the components of a Gödel program. So, for example, the module structure must be represented, as must all the language declarations and statements appearing in the program. The system module `Programs` provides a large number of operations on this abstract data type, including adding and deleting statements in a program, accessing language declarations, running goals to a program, and many others. There are two other major abstract data types for meta-programming. One is `Theory` which is the type of a term representing an object many-sorted theory. The system module `Theories` provides operations on this abstract data type which can be used to implement theorem provers, for example. The other type is `Script` which is the type of a term representing an object Gödel script. A script is similar to a program except there is no module structure. Scripts are particularly useful as the target for partial evaluators. The system module `Scripts` provides operations on the type `Script`. This approach to meta-programming is declarative – all the predicates providing the operations on the various abstract data types can be understood declaratively. This, combined with the fact that a large number of useful operations are provided by these modules, means that Gödel provides a congenial environment for meta-programming.

Finally, we briefly discuss input/output. Unfortunately, at the interface between a program and the external world, the declarative semantics of a program can be severely compromised. To try to ameliorate such difficulties, we recommend that programmers exploit Gödel's module system. The idea is to have all the modules which use input/output predicates as high as possible in the module hierarchy. This means that modules not above or equal to these input/output modules will not make any calls to input/output predicates. Thus they will contain pure code together possibly with commits. In other words, it will actually be modules not above or equal to the input/output modules of a Gödel program which will have the significantly improved declarative semantics.

In summary, there are precisely two facilities, input/output and pruning, which adversely affect the declarative semantics of Gödel programs. Input/output introduces side-effects via read predicates. However, as was pointed out above, this problem can be ameliorated by appropriate use of the module system. Commit affects the declarative semantics because it can cause correct answers to be pruned. Fortunately, Gödel programs generally need few uses of pruning. Except where use is made of either of these two facilities, Gödel programs are declarative.

The declarative nature of Gödel programs provides a number of important advantages. First, Gödel is eminently suited as a teaching language partly because students can concentrate much more on writing down what it is they want to compute without being so concerned about how to compute it. Second, Gödel narrows the gap between theory and practice in logic programming.

At the moment, most practical logic programming languages rely on a significant number of non-logical features for which there is no useful semantics. By greatly reducing its reliance on non-logical features, Gödel thus provides a solid bridge between theory and practice. Third, the fact that Gödel meta-programs are declarative makes some desirable applications possible. One of these is to build a compiler-generator by partial evaluating a (self-applicable) partial evaluator with respect to itself as input. Another such application is the use of a declarative debugger to debug Gödel programs. Declarative debugging is an attractive debugging technique which only requires that the programmer know the intended interpretation of a program to locate certain bugs. In particular, knowledge of the procedural behaviour of the Gödel system is not needed. Fourth, Gödel more easily allows a parallel implementation since there are only a couple of non-logical features to complicate matters. Finally, because the language has so few features (namely, input/output and pruning) which have side-effects or have global effects on a search tree, there is very little to unnecessarily impede the exploitation of parallelism in programs.

1.3 Comparison with Prolog

In this section, we make a detailed comparison between Prolog and Gödel. This comparison provides much of the motivation for why various design decisions for Gödel were made. In particular, we examine those Prolog features which are non-logical, since these are the ones that need most attention. We begin by discussing some general requirements for any programming language.

Five desirable properties of any programming language are that it be

- high level,
- expressive,
- efficient,
- practical, and have a
- simple (mathematical) semantics.

A high level language is one which provides concepts as close as possible to those which people like to use to express their thoughts and ideas. An expressive language is one which provides concepts that can be used to model real-world situations easily and concisely. An efficient language is one for which typical programs run at speeds and memory costs similar to competing languages. A practical language is one that can be used for large-scale, real-world applications. A language with a simple semantics is one for which programmers can easily verify and debug their programs and be assured of the correctness of program transformations, optimisations, and so on.

Prolog has proved to be a great success in a wide variety of application areas. This success is undoubtedly due to the fact that Prolog is high level, expressive, efficient, and practical. Prolog's importance and widespread use is well justified by these properties. However, Prolog's semantics (and by Prolog, we mean the practical programming language as it is embodied in currently available Prolog systems, not the idealised pure subsets studied in [15], for example) is much less satisfactory. The problems with the semantics are numerous and well known: the lack of occur check, the use of unsafe negation, the use of non-logical predicates, such as `var`, `nonvar`, `assert`, and `retract`, the undisciplined use of cut, and so on. These problematical aspects of Prolog cause many practical Prolog programs to have no declarative semantics at all and to have unnecessarily complicated procedural semantics. This means that the analysis, transformation, optimisation, verification, and debugging of many Prolog programs is extremely difficult.

The solution to these problems is to take more seriously the central thesis of logic programming, which is that

- a program is a (first order) theory, and
- computation is deduction from the theory.

It is crucially important that programs can be understood directly as theories. When this is the case, they have simple declarative semantics and can be much more easily verified, transformed, debugged, and so on. Each of the problematical aspects mentioned above causes difficulties precisely because it creates an impediment to the understanding of a program as a theory. A Prolog program which cannot be understood in some simple way as a theory has only a procedural semantics. This leaves us in no better position to understand the program than if it was written in a conventional procedural language.

Gödel directly addresses these semantic problems of Prolog. The main design aim of Gödel is to have functionality and expressiveness similar to Prolog, but to have greatly improved declarative semantics compared with Prolog. Gödel is intended to have the same relation to Prolog as Pascal does to Fortran. Fortran was one of the earliest high-level languages and suffered from a lack of understanding at that time of what were good programming language features. In particular, it had a meagre set of data types and relied on the `goto` for control. Pascal, which was introduced 10 years later, benefitted greatly from a considerable amount of research into programming language design. In particular, Pascal had a rich set of data types and relied on structured control facilities instead of the `goto`. Similarly, Prolog was designed at the birth of logic programming before researchers had a clear understanding of how best to handle many facilities. Consequently, these facilities compromised its declarative semantics. In the period since Prolog first appeared, various research projects have shown how to design logic programming languages with better software engineering facilities and greatly improved declarative semantics with all the well-known advantages that these bring. Our aim has been to exploit this research in the design of Gödel.

Now we turn to some problematical features of Prolog. To begin with, we discuss Prolog's approach to meta-programming, which is one of its most unsatisfactory aspects. Unfortunately, Prolog does not make a clear distinction between the object level and meta-level, and does not provide explicit language facilities for representation of object level expressions at the meta-level. Thus important representation (that is, naming) and semantic issues are glossed over. As we show below, Prolog's meta-programming problems can be traced to the fact that it doesn't handle the representation requirements properly. A consequence of this is that it is not possible to (directly) understand most Prolog meta-programs as theories and hence they do not have a declarative semantics. The most obvious symptom of this is with `var`, which has no declarative semantics at all in Prolog. However, within the framework of the appropriate representation, a meta-program is a first order theory and the meta-logical predicates of Prolog, such as `var`, `nonvar`, `assert`, and `retract`, have declarative counterparts.

We now point out precisely where Prolog's meta-programming problems lie. The first problem is illustrated by one of the best known meta-programs, the standard `solve` interpreter. This interpreter consists of the following definition for `solve`.

```
solve(empty).
solve(x and y) <- solve(x) & solve(y).
solve(x) <- clause(x,y) & solve(y).
```

together with a definition for `clause`, which is used to represent the object program. For example, if the object program contains the clause

```
p(x,y) <- q(x,z) & r(z,y).
```

then there is a corresponding clause of the form

```
clause(p(x,y), q(x,z) and r(z,y)).
```

appearing in the definition of `clause`.

However, the declarative meaning of the `solve` interpreter is by no means clear. The problem is that the variables in the definition of `clause` and the variables in the definition of `solve` intuitively range over different domains [10]. Thus the intended interpretation is simply not a model of the program. What is at stake here is whether it is possible to give a simple and precise semantics to the `solve` interpreter and other meta-programs. Without such a semantics, it is impossible, for example, to verify them or prove the correctness of transformations performed on them.

If the different kinds of variables are intended to range over different domains, then there is a clear solution. We should introduce types into the language underlying the meta-program. The key aspect of the appropriate representation is that an object level variable is represented by a meta-level variable. This representation is called the *non-ground* representation. Then, for example, using an appropriately typed version of the `solve` interpreter, it is possible to prove its soundness and completeness for both the declarative and procedural semantics [10].

However, Prolog has another problem related to the fact that an object level variable is (implicitly) represented by a meta-level *variable.* This leads to severe semantic problems with `var`, for example. With this representation, there seems to be no way of giving a declarative semantics to `var`. To see the difficulty, consider the goals

```
<- var(x) & solve(p(x)).
```

and

```
<- solve(p(x)) & var(x).
```

If the object program consists solely of the clause `p(a)`, then (using the "leftmost literal" computation rule) the first goal succeeds, while the second goal fails.

These considerations lead to another representation, called the *ground* representation, in which object level expressions are represented by ground terms at the meta-level. In particular, in the ground representation, an object level variable is represented by a ground meta-level term. The ground representation is a standard tool in mathematical logic. Using this representation, it is possible to give appropriate definitions for declarative counterparts of the static meta-logical predicates of Prolog, such as `var` and `nonvar`.

We remark that it is possible to identify two different uses of `var` in Prolog. The first is the "meta-logical" use which can be understood as above. This use of `var` is exemplified by the unification algorithm given on page 152 of [19]. The other use is the "control" use, which is exemplified by the program for `grandparent` given on page 148 of [19]. In Gödel, the meta-logical use of `var` is covered by a declarative replacement for it. Most of the control uses of `var` are covered by a flexible computation rule.

Meta-programs are often required to manipulate the representations of object programs, creating new object program representations dynamically. For example, a partial evaluator and a program transformer both do this. To do this declaratively, we need one further idea, which is that object programs should be represented not as meta-programs, but instead as meta-level terms [3]. Using this idea together with the ground representation, it is straightforward to set up the abstract data types used in the Gödel approach to meta-programming and to give appropriate definitions for declarative counterparts of the dynamic meta-logical predicates of Prolog, such as `assert` and `retract` [11].

We remark that the ground representation is easily the more important of the two representations. In fact, the non-ground representation seems useful mostly for the `solve` interpreter and various extensions of it. As soon as, for example, we want to do any significant manipulation of meta-level terms representing object level expressions, we must use the ground representation. This means that most of the important uses of meta-programming, such as compiling, transforming, partially evaluating, and debugging, will require the ground representation. Gödel does not provide any special support (other than the type system) for the non-ground representation. This means that, while it is straightforward to write in Gödel a `solve` interpreter and extensions of it in a similar way to Prolog, the non-ground representation of the object program must be given explicitly by the programmer. On the other hand, Gödel provides considerable support for the ground representation for the reasons given above.

Representing object level variables by ground meta-level terms means that we can no longer make direct use of the underlying unification of the system. In principle, much low-level computation, which is normally done directly and efficiently by the system, must now be done less efficiently by explicit Gödel code. However, there is a major implementation advantage in making programs more declarative, which is that more declarative programs are more easily parallelised. In fact, we conjecture that the cost of implementing the ground representation will be more than repaid by the greater degree of parallelism available on the coming generation of parallel machines. In other words, we conjecture that ultimately Gödel meta-programs will run faster than corresponding Prolog meta-programs!

Next we turn to pruning. Prolog uses the cut as its pruning operator. However, cut has a number of semantic problems, which were discussed in detail in [12]. The first of these problems is that cut, at least as it is employed in existing Prolog systems, allows considerable uncertainty about what the underlying logic component of the program is. This is because programmers can exploit the sequential nature of cut to leave "tests" out and when this is done the logic component of the program cannot be obtained by simply removing all the cuts from the program. Furthermore, there is no convention for systematically putting back the omitted tests so as to define the logic component precisely. The second problem with cut is that its use, in the presence of negation, can be unsound, in the sense that a computed answer may not be correct with respect to the completion of the logic component of the program. The third problem is that the class of programs containing cut is not closed under program transformations.

For these reasons and because the commit of the concurrent logic programming languages [18] has better semantics, Gödel's pruning operator, also called commit, is based on the commit of the concurrent languages. In fact, the simplest form of the commit (denoted `|` and called bar commit) is similar to the commit of the concurrent languages. Consider the following definition of the proposition `P`, in which we write `|` as a connective with the declarative meaning of conjunction.

```
P <- Q | R.
P <- S | T.
P <- U | V.
```

The order in which the statements are tried is not specified, so that bar commit does not have the sequentiality property of cut. The informal procedural meaning of bar commit is that it finds only one solution for the successful guard (that is, the conjunction of literals in a body to the left of a commit) and prunes all other statements in the definition which contain a commit.

We now investigate the extent to which `|` supports program transformations. Consider the following program.

```
P <- M & L.
P <- N.
M <- Q | R.
M <- S | T.
N <- U | V.
```

How can we unfold on `M` and `N` in the definition for `P`? A first attempt might be the following definition for `P`.

```
P <- Q | R & L.
P <- S | T & L.
P <- U | V.
```

However, this is not correct as the `|` notation is unable to distinguish the scopes of the commits in the first two statements from the scope of the commit in the third statement. This difficulty leads us to introduce a more flexible notation than `|`. This notation has the form `{...}_l`, where `l` is a label (which is a positive integer). In this notation, the previous program would be written as follows.

```
P <- M & L.
P <- N.
M <- {Q}_1 & R.
M <- {S}_1 & T.
N <- {U}_1 & V.
```

The brackets `{...}` capture the scope of the commit inside a statement. By convention, for `|` this scope is the conjunction of literals to its left. The label captures the scope of the commit over the statements in the definition. When a conjunction `{L1 & ... & Ln}_l` succeeds, all other statements in the definition which contain a commit labelled `l` are pruned. The other pruning that takes place is that only one solution is found for `L1 & ... & Ln`. Now when we unfold on `M` and `N` in the definition for `P`, we obtain the following definition

```
P <- {Q}_1 & R & L.
P <- {S}_1 & T & L.
P <- {U}_2 & V.
```

in which the label in the commit in the definition for `N` has been standardised apart to avoid an unwanted coincidence with the label of the commits in the definition for `M`. Note that we have now correctly distinguished the scopes of the commits.

The notation for the commit is very flexible. For example, we could have the following definition for `P`.

```
P <- {Q & {R}_2}_1.
P <- {S & {W}_2}_1 & T.
P <- {U & V}.
P <- W & V.
```

This definition illustrates that a single statement can contain several commits and that commits can be nested. Note also that the commit `{...}` in the third statement has no label. This is syntactic sugar for a commit with a label different from all other commit labels in the definition. This commit is a one-solution operator. Thus the commit operator contains a one-solution operator as a special case. We envisage programmers writing almost all their programs with just the two important special cases of the commit – the `|` and the one-solution operator `{...}`. However, source-level tools, such as partial evaluators and program transformers, will require the flexibility of the commit, since definitions as complex as the previous one can easily be obtained from definitions using the `|` notation after just a few unfolding steps.

Other problematical aspects of Prolog can be repaired comparatively easily. For example, Gödel requires a sound implementation of negation instead of Prolog's unsound version. A sound implementation of negation is not expensive or difficult to implement. Related to this, Gödel adopts the declarative and flexible if-then-else construct of NU-Prolog [20]. Also Gödel (at least in principle) does the occur check during unification. However, the occur check is generally very expensive and so it is important for an implementation to avoid occur checks whenever possible. Fortunately, in practice the occur check very rarely causes the unification algorithm to fail, so there is considerable scope for reducing the occur check overhead by determining through program analysis the few places where the occur check may be needed. In this way, an implementation can be both sound and efficient.

Many extensions and variations of Prolog have been introduced and studied by the logic programming community over the last 15 years. These include concurrent languages, constraint languages, and higher order languages. Unfortunately, these languages have been essentially built on top of Prolog and therefore inherit many of Prolog's semantic problems. For example, most of these languages use Prolog's approach to meta-programming. We hope that the designers of future logic programming languages will see from Gödel that it really *is* possible to design and implement a logic programming language which is both practical and declarative.

Chapter 2

Types

In this chapter, we discuss the Gödel type system, giving details of the various kinds of language declarations which are used to define the polymorphic many-sorted languages in which programs are written.

2.1 Many-Sortedness

The main purpose of this chapter is to explain the Gödel type system. However, since a program consists of a collection of modules, to do that it is convenient to introduce here the simplest form of module. Modules are explained in detail in chapter 5.

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

Now let us turn to the Gödel type system, which is based on many-sorted logic with parametric polymorphism. (See appendix A.) We first discuss the many-sorted aspect of the type system. Consider module `M1` below which defines the predicates `Append` and `Append3` for appending lists of days of the week. Note that variables are denoted by identifiers beginning with a lower case letter and constants by identifiers beginning with an upper case letter.

In general, language declarations begin with the keywords `BASE`, `CONSTRUCTOR`, `CONSTANT`, `FUNCTION`, `PROPOSITION`, or `PREDICATE`. These declarations declare the *symbols* of the language, which belong in one of the *categories* : base, constructor, constant, function, proposition, or predicate. In module `M1`, the language declaration beginning with the keyword `BASE` gives the types of the many-sorted language of the module. It declares `Day` and `ListOfDay` to be *bases*, which are the only types of the language. (More complicated types will be introduced shortly.) The next three declarations declare the constants, functions, and predicates of the language. The first part of the `CONSTANT` declaration declares `Nil` to be a constant of type `ListOfDay`. The second part declares `Monday`, `Tuesday`, etc., to be constants of type `Day`. The `FUNCTION` declaration declares `Cons` to be a binary function which maps a tuple of arguments, where the first argument is of type `Day` and the second argument is of type `ListOfDay`, to an element of

```
MODULE        M1.

BASE          Day, ListOfDay.

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

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

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

type `ListOfDay`. The `PREDICATE` declaration declares `Append` to be a ternary predicate each of whose arguments has type `ListOfDay`. It also declares `Append3` to be a quaternary predicate each of whose arguments has type `ListOfDay`. Statements and goals are written in the language defined by the language declarations. A proposition is declared with a `PROPOSITION` declaration. So, for example, a module may contain the declaration

```
PROPOSITION   P,Q.
```

which declares `P` and `Q` to be propositions.

In general, the identifier immediately after the keyword of a language declaration is called the *name* of the symbol being declared. Until the last section of this chapter, distinct symbols declared in the same module have distinct names. Thus, until then, in the context of a single module, a symbol can be uniquely identified by its name and we are free, as we have already done in the previous paragraph, to refer to a symbol via its name. Later we will introduce a more flexible mechanism which allows overloading, that is, the use of the same name for distinct symbols.

Module `M1` forms a complete program on its own. Typical goals for this program could be as follows.

```
<- Append3(Cons(Monday,Nil), Cons(Tuesday,Nil), Cons(Wednesday,Nil), x).

<- Append3(x, y, z, Cons(Monday,Cons(Tuesday,Cons(Wednesday,Nil)))).
```

Note that every type of the language must be declared. Also every constant, function, proposition, and predicate of the language must be declared. The types of variables are not declared, but are assigned using their position in statements and the language declarations given. Needless to say, a variable must be used consistently within a single statement. That is, each occurrence of the variable must have the same (assigned) type. However, there is no requirement that a variable have a consistent type when used in different statements. That is, a variable `x`, say, could have one type in one statement and another type in another statement. This notational flexibility is very convenient in a programming language. By a suitable renaming of variables, if necessary, it is straightforward to map a program employing this licence with variables to a many-sorted theory with the stricter variable restrictions of many-sorted logic.

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

```
MODULE        M2.

BASE          Day.
CONSTRUCTOR   List/1.

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

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

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

All bases and constructors of the language must be declared. If there is no constructor, then the set of all types is just the set of all bases. If at least one constructor is declared, then the set of all types is obtained by applying the above construction using the bases as "constants" and the constructors as "functions" of the appropriate arity. For example, if a base `Person` were declared in addition to those bases declared in module `M2`, then the set of all types would be `Day`, `Person`, `List(Day)`, `List(Person)`, `List(List(Day))`, `List(List(Person))`, ...

In mathematical treatments of many-sorted logic, there is no need for the types (i.e. sorts) to have any structure. They are usually considered to be simply (unstructured) elements of the set of types. In contrast, for programming languages, for notational reasons it is very convenient for types to have structured names as Gödel does.

Note also that once there is at least one constructor and one base declared, the set of types is (countably) infinite. In some cases, this may mean that the language contains more types than the programmer really wants. For example, in module `M2`, the only types of interest are `Day` and `List(Day)`. It would be possible to have a more flexible way of specifying types so that, in the case that a programmer wanted only some subset of the types given by the above construction, this could be achieved. For example, a more flexible notation for language declarations could allow a programmer to have only the types `Day` and `List(Day)` in the language. In most cases, it is not worth the extra complication in the language declarations needed to achieve this since no great harm is done in assuming that the intended interpretation is augmented with the extra types.

Type assignment for variables is handled as follows. For each statement in a module, there must be an assignment of a type to each variable and its corresponding (possibly implicit) quantifier in the statement so that the statement is a many-sorted formula. If there is such an assignment of types, then it is unique. For example, for module `M2`, the variable `x` in the first statement is assigned the type `List(Day)`. In the second statement, the variable `u` is assigned the type `Day`, and the variables `x`, `y`, and `z` are assigned the type `List(Day)`. So the collection of declarations and statements of module `M2` together do constitute a module.

However, the declarations

```
BASE          Day, Person.
PREDICATE     P : Day * Person.
```

and the statement

```
P(x,x).
```

(together with a `MODULE` declaration) do not constitute a module, since there is no type which can be assigned to `x` to satisfy the declaration for `P`.

2.2 Polymorphism

Next we introduce the second aspect of the type system, which is parametric polymorphism. (See appendix A.) It is common for a programmer to want to write a definition of a predicate for which the arguments of the predicate can have a variety of types. For example, the `Append` predicate is normally written so that it can append lists of any type. For this purpose, we add parametric polymorphism to the type system, as illustrated by module `M3`.

The logic on which module `M3` is based is called *polymorphic many-sorted logic*. In module `M3`, `a` is a *parameter*, which is a type variable. A parameter can be instantiated to any type of the language of the logic. Like variables, parameters are not declared. For a polymorphic many-sorted language, we need to extend the concept of a type. For such a language, a type is a "term" constructed using the bases as "constants", the parameters as "variables", and the constructors as "functions". Thus, for module `M3`, the following are types: `a`, `List(a)`, and `List(List(Day))`. A *ground* type (also called a *monotype*) is a type not containing parameters. Thus the ground types for module `M3` are `Day`, `Person`, `List(Day)`, `List(Person)`, `List(List(Day))`, ...

```
MODULE        M3.

BASE          Day, Person.
CONSTRUCTOR   List/1.

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

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

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

A symbol is *polymorphic* if its declaration contains a parameter; otherwise, it is *monomorphic*. A polymorphic symbol can be understood as representing a collection of (monomorphic) symbols. For example, the `CONSTANT` declaration for the polymorphic constant `Nil` in module `M3` declares a countably infinite set of (monomorphic) constants of type `List(`τ`)`, where τ ranges over all ground types. It is helpful to imagine the distinct constants in this collection as being subscripted by the corresponding value for τ. Thus the (monomorphic) constants declared by the declaration for `Nil` are $\texttt{Nil}_{\texttt{Day}}$ of type `List(Day)`, $\texttt{Nil}_{\texttt{Person}}$ of type `List(Person)`, $\texttt{Nil}_{\texttt{List(Day)}}$ of type `List(List(Day))`, and so on. There is a similar understanding of polymorphic functions and predicates.

The definition of a formula in a polymorphic many-sorted language is given in appendix A. It is decidable whether an expression is a formula or not. A polymorphic formula can be understood as representing a collection of (monomorphic) formulas. For example, consider module `M3`. During checks that the statements in `M3` are actually formulas, the variable `u` in the second statement for `Append` has the type `a` assigned, and the variables `x`, `y`, and `z` have the type `List(a)` assigned. Thus the second statement for `Append` represents the following collection of formulas:
$\texttt{Append}_{\texttt{Day}}\texttt{(Cons}_{\texttt{Day}}\texttt{(u,x),y,Cons}_{\texttt{Day}}\texttt{(u,z)) <- Append}_{\texttt{Day}}\texttt{(x,y,z)}$,
$\texttt{Append}_{\texttt{Person}}\texttt{(Cons}_{\texttt{Person}}\texttt{(u,x),y,Cons}_{\texttt{Person}}\texttt{(u,z)) <- Append}_{\texttt{Person}}\texttt{(x,y,z)}$,
$\texttt{Append}_{\texttt{List(Day)}}\texttt{(Cons}_{\texttt{List(Day)}}\texttt{(u,x),y,Cons}_{\texttt{List(Day)}}\texttt{(u,z)) <- Append}_{\texttt{List(Day)}}\texttt{(x,y,z)}$,
and so on.

As another example, consider the module `ParametricTypes` below. The variable `x` has type `a` in the atom `Q(x)` and has type `M` in the atom `P(F(x))`. Thus the variable `x` has type `M` in `P(F(x))`

`<- Q(x)` because the types of the two occurrences of `x` in the statement can be unified to give the type `M`. Thus the argument `F(x)` of `P` has type `L(M)` in this statement.

```
MODULE        ParametricTypes.

BASE          M.

CONSTRUCTOR   L/1.

FUNCTION      F : a -> L(a).

PREDICATE     P : L(M);
              Q : a.

P(F(x)) <- Q(x).
```

In addition, we need to extend the usual definitions of interpretation, model, logical consequence, and so on, to polymorphic many-sorted logic. For a polymorphic many-sorted language, an interpretation has a domain corresponding to every ground type and these domains are pairwise disjoint. Then a constant such as `Nil` in module `M3` with polymorphic type `List(a)` is assigned a (countably infinite) set of domain elements, one in the domain of each ground type which can be obtained by instantiating `a` in `List(a)` with all possible ground types. This corresponds to the assignment as for a (monomorphic) many-sorted language of a single domain element to each of the (monomorphic) constants which `Nil` represents. In a similar way, one can define the assignments to functions and predicates. The definitions of model, logical consequence, and so on, also generalise from a (monomorphic) many-sorted language to a polymorphic many-sorted language in a similar way. The details can be found in appendix A.

For a module such as `M3` which does not depend upon other modules, the *language of the module* is the polymorphic many-sorted language defined by the language declarations in the module. In chapter 5, we will extend this concept to the case where a module is one of a number of modules constituting a program.

A *statement*[1] in a module is a formula in the language of the module having the form either `A` or `A <- W`, where `A` is an atom, called the *head* of the statement, and `W` is a formula, called 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. In addition, each statement must satisfy a condition on the types of the arguments in the head. This condition will be given shortly.

A *goal*[1] for a program consisting of a single module[2] has the form `<- W`, where `W` is a formula in the language of the module. `W` is called the *body* of the goal. Any free variables in `W` are assumed to be universally quantified at the front of the goal.

Module `M3` forms a complete program on its own. Typical goals for this program could be as follows:

[1]In chapter 7 we shall extend this definition to include commits.

[2]The module must also be open. See chapter 5 for the definition of an open module.

```
<- Append3(Cons(Monday,Nil), Cons(Tuesday,Nil), Cons(Wednesday,Nil), x).

<- Append3(x, y, z, Cons(Fred,Cons(Bill,Cons(Mary,Nil)))).
```

When a goal is given for a program consisting of a single module[2], the goal is first checked to make sure it is a formula in the language of the module. A desirable requirement of a type system is that goals can then be run with no run-time type checking. However, we need to impose two conditions to ensure this property holds.

The first of these conditions is the head condition.

- 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.

Note that the head condition is satisfied by the statement in module `ParametricTypes` and also each statement in module `M3`.

The second condition is transparency. To state this, some new terminology is convenient. The type appearing on the right of the `->` in a function declaration is called the *range type*.

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

Then, under the assumptions that each statement satisfies the head condition and that each function declaration is transparent, it can be shown that no run-time type checking is needed with standard proof procedures. For example, no type error will occur if a program and goal are run under the usual (untyped) SLDNF-resolution. (See [13, theorem 5.7].) The head condition and transparency are general and natural conditions, so their imposition rarely causes any inconvenience.

Lists are a common data structure. Thus Gödel provides, via the module system, a built-in constructor for list processing. The constructor is called `List` and has arity 1. There is also a constant `Nil` and a function `Cons`, whose declarations are as follows.

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

Gödel also provides a special list syntax which is syntactic sugar on top of `Nil` and `Cons`. This is the usual `[...]` and `|` syntax. Thus the list

```
Cons(Fred,Cons(Bill,Cons(Mary,Nil)))
```

can be written more conveniently as

```
[Fred,Bill,Mary].
```

Similarly, the list

```
Cons(Fred,Cons(Bill,x))
```

can be written as

```
[Fred,Bill|x].
```

The way the constructor `List/1`, the list notation, and the various list processing predicates are made available for use is by means of the module `Lists`, which is provided by the system. In the module `Lists`, the constructor `List/1` is declared, as are the constant `Nil` and the function `Cons`. Also a collection of useful list processing predicates, including `Append`, is defined there. Another module can make these predicates available for use by means of an `IMPORT` module declaration. For example, in module `M4`, the `IMPORT` declaration imports into `M4` the constructor `List`, the constant `Nil`, the function `Cons`, and various predicates including `Append`. Thus `List`, `Nil`, `Cons`, and `Append` are available for use in `M4`. We shall see the way `Lists` exports these in chapters 5 and 6. In general, if a module imports from another module, it imports *all* the symbols exported by the other module.

With this use of modules and the special list notation, we can now give the final version of module `M1`, which is module `M4`. Modules `M4`, `Lists`, and `Integers` (see chapter 4), which is imported by `Lists`, together form a program. Typical goals for this program could then be as follows.

```
<- Append3([Monday], [Tuesday], [Wednesday], x).

<- Append3(x, y, z, [Fred,Bill,Mary]).
```

```
MODULE      M4.

IMPORT      Lists.

BASE        Day, Person.

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

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

Chapter 3

Formulas

As we indicated in chapter 2, arbitrary formulas can be used in the bodies of statements and goals. In this chapter, we examine the use of this facility in more detail.

3.1 Quantifiers and Connectives

In Gödel, conjunction is denoted by `&`, disjunction by `\/`, negation by `~`, (left) implication by `<-`, (right) implication by `->`, and equivalence by `<->`. The universal quantifier is denoted by `ALL` and the existential quantifier is denoted by `SOME`. Each quantifier has two arguments, the first being a list of the quantified variables and the second the scope of the quantifier. Note that the types of the quantified variables in the body of a statement or goal are not declared. The types of these variables are assigned by the type checker along with the types of the other variables in the statement or goal.

As an example, module `Inclusion` below contains the definition of the predicate `IncludedIn`, which is true when its first argument is a list of elements of some type and its second argument is a list of elements of the same type which includes all the elements appearing in the first argument. Thus, for example, the goal[1]

```
<- IncludedIn([1,3,2],[4,3,2,1]).
```

succeeds and the goal

```
<- IncludedIn([1,5,2],[4,3,2,1]).
```

fails.

Allowing an arbitrary formula to appear in the body of a goal is very useful for querying databases, such as module `DB` below. A typical goal to the program formed by this module could be the following:
Does every person with a mother also have a father?

```
<- ALL [x] (SOME [z] Father(z,x) <- SOME [y] Mother(y,x)).
```

[1]Note that the integers are available in the module `Inclusion` since they are imported via `Lists`. See chapter 6.

```
MODULE         Inclusion.

IMPORT         Lists.

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

IncludedIn(x,y) <-
               ALL [z] (Member(z,y) <- Member(z,x)).
```

Taking advantage of the notational convention that a variable name beginning with an '_' stands for a unique variable existentially quantified at the front of the atom in which it appears, the previous goal can be written more compactly as follows.

```
<- ALL [x] (Father(_,x) <- Mother(_,x)).
```

Another goal could be the following:
Find a mother who has no father.

```
<- SOME [z] Mother(x,z) & ~ SOME [y] Father(y,x).
```

Using the underscore notation, this can be written as follows.

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

Sometimes a programmer may only be interested in the values of a subset of the free variables in the body of a goal. In this case, the unwanted variables can be masked out using the following colon notation. For example, suppose the goal was as follows:
Find all grandparents of Jane.

```
<- x : Parent(x,y) & Parent(y,Jane).
```

The "`x :`" reads "find the `x` such that". In general, there can be zero or more variables before the colon, separated by commas. Actually, the "`x,y,... :`" can be replaced by an equivalent "`SOME [u,v,...]`", as in

```
<- SOME [y] (Parent(x,y) & Parent(y,Jane)).
```

However, the : notation is usually more convenient, since it expresses directly the variables of interest and is usually shorter. It is an error if a variable appears before the colon but is not a free variable in the body of the goal. Note that the colon notation is not available for use in statements.

```
MODULE        DB.

BASE          Person.

CONSTANT      Fred, Mary, George, James, Jane, Sue : Person.

PREDICATE     Ancestor, Parent, Mother, Father : Person * Person.

Ancestor(x,y) <-
              Parent(x,y).
Ancestor(x,y) <-
              Parent(x,z) &
              Ancestor(z,y).

Parent(x,y) <-
              Mother(x,y).
Parent(x,y) <-
              Father(x,y).

Father(Fred, Mary).
Father(George, James).

Mother(Sue, Mary).
Mother(Jane, Sue).
```

3.2 Conditionals

The connectives and quantifiers introduced so far provide an expressive language in which to write goals and statements. However, there are situations for which it is advantageous to provide constructs with specialised procedural semantics. The most common such situation is when a programmer wants to use a formula of the form

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

in a body. Although this formula is written with the connectives introduced so far, this approach has the disadvantage that *Condition* may be computed twice. Obviously, if *Condition* is computationally expensive, this is undesirable. For this reason, Gödel also has `IF-THEN` and `IF-THEN-ELSE` constructs, called *conditionals*, with specialised procedural semantics to avoid this inefficiency.

Each of these constructs has two forms. The first form of the `IF-THEN` construct is

`IF` *Condition* `THEN` *Formula*

where *Condition* and *Formula* are formulas. (If *Condition* has an existential quantifier at the top level, then *Condition* must be bracketed for this form of the `IF-THEN`. The reason for this will become clear shortly.) This formula is defined to mean

(*Condition* & *Formula*)   \/   ~*Condition*

or, equivalently,

Condition -> *Formula*.

So this construct is redundant. It is included mainly because it completes the collection of conditionals.

Sometimes this first form of the IF-THEN construct is not suitable because *Condition* and *Formula* share local variables. For this reason, a second form of the IF-THEN construct due to Naish [17] is available. This is

IF SOME [*x1,...,xn*] *Condition* THEN *Formula*

where *Condition* and *Formula* are formulas. This formula is defined to mean

(SOME [*x1,...,xn*] (*Condition* & *Formula*))   \/   ~ SOME [*x1,...,xn*] *Condition*.

Note carefully that this is a non-standard use of the connective IF-THEN in that the quantification extends over both *Condition* and *Formula*. However, from a programming viewpoint, this non-standard use is very convenient. An implementation of this form of IF-THEN must avoid the inefficiency of computing *Condition* twice.

Now we turn to the IF-THEN-ELSE construct. The first form of this is

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

where *Condition*, *Formula1* and *Formula2* are formulas. (If *Condition* has an existential quantifier at the top level, then *Condition* must be bracketed for this form of the IF-THEN-ELSE.) This formula is defined to mean

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

As an example of the use of this form of IF-THEN-ELSE, here is a definition of the predicate Max, which computes the maximum of its first two arguments.

```
Max(x,y,z) <- IF x =< y THEN z = y ELSE z = x.
```

Gödel has a second, more flexible, form of the IF-THEN-ELSE construct due to Naish [17], which is

IF SOME [*x1,...,xn*] *Condition* THEN *Formula1* ELSE *Formula2*

where *Condition*, *Formula1* and *Formula2* are formulas. This formula is defined to mean

(SOME [*x1,...,xn*] (*Condition* & *Formula1*))   \/   (~ SOME [*x1,...,xn*] *Condition* & *Formula2*).

Implementations of both forms of IF-THEN-ELSE must avoid the inefficiency of computing *Condition* twice.

If a programmer wants to make use of the first form of IF-THEN, but the condition has an existential quantifier at the top level, then explicit scoping must be indicated by using brackets. Thus

IF (SOME [*x1,...,xn*] *Condition*) THEN *Formula*

means

`((SOME [`*x1,…,xn*`]` *Condition*`) &` *Formula*`)   \/   ~ SOME [`*x1,…,xn*`]` *Condition*

and not

`(SOME [`*x1,…,xn*`] (`*Condition* `&` *Formula*`))   \/   ~ SOME [`*x1,…,xn*`]` *Condition.*

Similar remarks apply to the first form of `IF-THEN-ELSE`.

Nesting of these constructs is allowed, so that a conditional can appear inside another conditional. For example, a case statement can be simulated by a formula of the form

```
IF C1 THEN S1
      ELSE IF C2 THEN S2
                 ELSE ...
                            ELSE IF Cn THEN Sn.
```

Note that the ambiguity in a formula such as

```
IF C1 THEN IF C2 THEN S1 ELSE S2
```

is resolved as

```
IF C1 THEN (IF C2 THEN S1) ELSE S2
```

that is, the `ELSE` is associated with the outermost `IF`. A construct such as

```
IF SOME [v1,...,vn] (P1 & ... & Pk) THEN S1 ELSE S2
```

can be written without the parentheses as follows:

```
IF SOME [v1,...,vn] P1 & ... & Pk THEN S1 ELSE S2.
```

The precise grammar for conditionals is given in chapter 11.

As an example of the use of the `IF-THEN-ELSE` construct, consider the module `AssocList` below, which defines the predicate `Lookup` for looking up an association list. The list consists of pairs having an integer key in the first argument and a string in the second. (See the module `Strings` in chapter 6.) The definition of `Lookup` is due to Naish [17]. If `Lookup` is called with only its first and third arguments instantiated, and there is a pair in the association list having key value equal to the first argument, it returns all such pairs. If the second argument is also instantiated and there is a pair in the association list with arguments equal to the first and second arguments, resp., of the call to `Lookup`, then it succeeds. If it is called with the first and third argument, and possibly the second argument, instantiated and there is no pair in the association list having key value equal to the first argument, it inserts the pair containing the first and second arguments into the association list. Note the quantification of the variable `v` over the condition and the `THEN` part, and how this gives the required behaviour of `Lookup`. Thus, for example, the goal

```
<- Lookup(5, value, [Pair(5,"abc"), Pair(5,"xyz"), Pair(1,"lmn")], list).
```

has the answers

```
list = [Pair(5,"abc"),Pair(5,"xyz"),Pair(1,"lmn")]
value = "abc"
```

and

```
list = [Pair(5,"abc"),Pair(5,"xyz"),Pair(1,"lmn")]
value = "xyz"
```

while the goal

```
<- Lookup(4, "rst", [Pair(5,"abc"), Pair(2,"xyz"), Pair(1,"lmn")], list).
```

has the answer

```
list = [Pair(4,"rst"),Pair(5,"abc"),Pair(2,"xyz"),Pair(1,"lmn")].
```

```
MODULE      AssocList.

IMPORT      Strings.

BASE        PairType.

FUNCTION    Pair : Integer * String -> PairType.

PREDICATE   Lookup : Integer * String * List(PairType) * List(PairType).

Lookup(key, value, assoc_list, new_assoc_list) <-
            IF SOME [v] Member(Pair(key,v), assoc_list)
            THEN
                value = v &
                new_assoc_list = assoc_list
            ELSE
                new_assoc_list = [Pair(key,value) | assoc_list].
```

The Gödel programming style makes significant use of conditionals. This is illustrated in chapter 10 where several programs use statements involving complicated nested conditionals.

3.3 Operators

It is often convenient to employ prefix, infix, or postfix notation instead of the standard logical notation for terms and atoms. For example, we usually prefer to write `1 + 2` instead of `+(1,2)`, `x < y` instead of `<(x,y)`, and `x * y + z` instead of `+(*(x,y),z)`. For this reason, Gödel provides a mechanism for declaring operators, which are functions or predicates employing such notation.

`FUNCTION` declarations allow the optional declaration of fixity, precedence, and associativity of unary or binary functions via *function indicators*. The general form of an indicator for an infix binary function is either `xFx(`N`)`, `xFy(`N`)`, or `yFx(`N`)`. Each of `xFx`, `xFy`, and `yFx` is a *specifier*.

The N, which is a positive integer, is the *precedence*. The position of the `F` indicates that the fixity of the function is infix. We shall see shortly how the `x` and `y` indicate associativity. Prefix and postfix binary functions are not allowed. For example, the module `Integers` provided by the system contains the following function declarations.

```
FUNCTION      ^ : yFx(540) : Integer * Integer -> Integer;
              * : yFx(520) : Integer * Integer -> Integer;
              + : yFx(510) : Integer * Integer -> Integer;
              - : yFx(510) : Integer * Integer -> Integer.
```

These declarations allow the use of infix notation. For example, the specifier `yFx` in the declaration for `+` in `Integers` indicates that `+` is infix so that `x + y` can be written instead of `+(x,y)`, and `x + (y + z)` instead of `+(x,+(y,z))`.

However, with the fixity information alone, although we can use the infix notation, we must always bracket subterms with structure. For a term containing functions with distinct precedences, the precedence information is used to disambiguate the meaning of the term in the absence of brackets. The precedence rule is that *the higher its precedence, the more strongly a function binds its arguments*. For example, the precedence `510` for `+` and the precedence `520` for `*` in `Integers` indicate that the term `x * y + z` means `(x * y) + z` and not `x * (y + z)`. If the latter meaning is intended, then the term must be written with explicit bracketing to indicate this. In general, precedence (and associativity) information in indicators can always be over-ridden by explicit bracketing.

The associativity information is needed when a term contains functions with the same precedence. An `x` indicates that the argument in that position must have precedence strictly higher than the precedence of the function having the indicator, while a `y` indicates that the argument in that position must have precedence higher than or equal to the precedence of the function having the indicator. For this purpose, the precedence of a term is the precedence of its top-level function, where the precedence of a function without an indicator is taken to be (plus) infinity. The precedence of variables, constants, and terms enclosed in brackets is also assumed to be (plus) infinity. Thus the specifier `xFy` means the function is right associative, `yFx` means the function is left associative, and `xFx` means the function is non-associative. For example, the indicator `yFx(510)` in the declaration for `+` in `Integers` declares it to be infix, left associative, and to have precedence `510`. So the term `x + y + z` means `(x + y) + z` and not `x + (y + z)`. Also the indicator `yFx(520)` in the declaration for `*` in `Integers` declares it to be infix, left associative, and to have precedence `520`. So the term `x * (y + z) * w` means `(x * (y + z)) * w`. Since `^` (exponentiation) has indicator `yFx(540)`, the term

```
x + 45 * z ^ w ^ u * 2
```

means

```
x + ((45 * ((z ^ w) ^ u)) * 2).
```

Indicators do provide notational convenience. However, this last example shows that it is possible to overdo their use. Often a set of well-placed brackets, even if they are strictly redundant, greatly reduces the difficulty of understanding the meaning of a complicated term.

The general form of a unary function indicator is either `xF(`N`)`, `Fx(`N`)`, `yF(`N`)`, or `Fy(`N`)`, where the specifier is either `xF`, `Fx`, `yF`, or `Fy`, and N, which is a positive integer, is the precedence.

`Fx` and `Fy` indicate prefix, and `xF` and `yF` indicate postfix. The `x` and `y` have the same meanings as for binary functions. For example, the module `Integers` also contains the following function declaration.

```
FUNCTION     - : Fy(530) : Integer -> Integer.
```

The indicator `Fy(530)` declares (unary) `-` to be prefix, right associative, and to have precedence `530`. Thus the term `-x + y` means `(-x) + y` and the term `- -x` means `-(-x)`.

`PREDICATE` declarations have an optional fixity declaration using a *predicate indicator*. Predicate indicators consist solely of a *specifier* which indicates the fixity of the predicate. The specifier for a binary infix predicate is `zPz`. Prefix and postfix binary predicates are not allowed. A specifier for a unary predicate is either `Pz`, indicating that the predicate is prefix, or `zP`, indicating that the predicate is postfix.

For example, `Integers` contains the following predicate declarations.

```
PREDICATE    > : zPz : Integer * Integer;
             < : zPz : Integer * Integer;
             >= : zPz : Integer * Integer;
             =< : zPz : Integer * Integer.
```

The indicator `zPz` in the declaration for `=<` declares it to be infix. Thus one can write the atom `=<(t1,t2)` in the more convenient form `t1 =< t2`.

More formally, an *operator* is a function or predicate whose language declaration contains an indicator. The precise syntax of operators can be found in chapter 11. The use of operators can lead to ambiguities in the parsing of expressions, as is illustrated in chapter 5.

Chapter 4

Equality and Numbers

In this chapter, we discuss the equality predicate and also the modules which provide the integers, rationals, and floating-point numbers.

4.1 Equality

Equality is a distinguished predicate in logic because it is so fundamental. The same is true of the equality predicate = in Gödel. Equality is built into the system and is available in every module without being imported. It has the following declaration.

```
PREDICATE  = : zPz : a * a.
```

A companion to equality is the disequality predicate ~=, whose declaration and definition are as follows.

```
PREDICATE  ~= : zPz : a * a.

x ~= y <-> ~(x = y).
```

Disequality is also built into the system and is available in every module without being imported.

The equality theory (for user-defined types) is given in section A.3 for the general case. We illustrate this here by giving the equality theory for the program which consists just of the module M3. Note that, in each of the following axioms, any variables in an axiom are assumed to be universally quantified at the front of the axiom. First, we have the axioms corresponding to the axiom (schema) 1 of the equality theory.

```
Monday ~= Tuesday.

Monday ~= Wednesday.
         ⋮
Sunday ~= Saturday.

Fred ~= Bill.

Fred ~= Mary.

Bill ~= Mary.
```

There is nothing corresponding to axiom 2 since there is only one function in the language of module `M3`. Corresponding to axiom 3, we have the following axiom.

```
Cons(x,y) ~= Nil.
```

The following axiom (schema) corresponds to axiom 4.

`Cons(`s`,`t`) ~= x`, where `Cons(`s`,`t`)` contains `x`.

Then comes the axiom corresonding to axiom 5.

```
(x1 ~= y1 \/ x2 ~= y2) -> Cons(x1,x2) ~= Cons(y1,y2).
```

Finally, there are the reflexivity and substitutivity axioms corresponding to axioms 6, 7 and 8.

```
x = x.

(x1 = y1 & x2 = y2) -> Cons(x1,x2) = Cons(y1,y2).

(x1 = y1 & x2 = y2) -> (x1 = x2 -> y1 = y2).

(x1 = y1 & x2 = y2 & x3 = y3) -> (Append(x1,x2,x3) -> Append(y1,y2,y3)).

(x1 = y1 & x2 = y2 & x3 = y3 & x4 = y4) ->
                                   (Append3(x1,x2,x3,x4) -> Append3(y1,y2,y3,y4)).
```

The equality theory consisting of all the above axioms corresponds to "syntactic identity", that is, two (ground) terms of the same type are equal if and only if they are syntactically identical. It is often the case that an intended interpretation of such a theory is an Herbrand interpretation [15] with equality interpreted as identity. However, for some system-defined base types, such as `Integer`, `Rational`, and `Float` corresponding to the integers, rationals, and floating-point numbers, respectively, equality is handled differently. For these types, we do not want to treat equality as syntactic identity. To see this, consider the terms (of type `Integer`)

```
23 + (3 * 5)
```

and

```
40 - 2.
```

With the axioms of the equality theory of section A.3, these two terms would not be equal. However, the domain of the intended interpretation is the integers and the intended interpretation of the functions in these terms are the usual arithmetic operations on the integers. For this (non-Herbrand) interpretation, the terms are equal because they both evaluate to `38`. Consequently, for the type `Integer`, the corresponding `=` is given the standard meaning of equality on the domain of integers. The types `Rational` and `Float` are treated similarly. More complicated types that involve `Integers`, `Rationals`, and `Floats`, as well as other bases and constructors, are handled in the obvious way. Consider, as an example, the type `List(Integer)`. For this, the components of a term corresponding to the constructor `List` of the type (that is, `Cons` and `Nil`) are handled using syntactic identity and the components corresponding to the base `Integer` are handled using the intended meaning of equality on the integers. Thus, for example, the goal

```
<- [2+3,6] = [5,7-1].
```

succeeds because `[2+3,6] = [5,7-1]` is true in the intended interpretation.

One subtle point when using underscores with `~=` is worth noting. Recall that a variable beginning with an underscore in an atom in a body means a unique variable existentially quantified at the front of the atom. If an underscore appears in S `~=` T, where S and T are terms, we treat this as `~(`S `=` T`)`, in which case the underscore becomes a unique variable *universally* quantified at the front of S `~=` T. So, for example, `F(x) ~= F(_)` means `ALL [y] F(x) ~= F(y)`, where `y` is a unique variable, and the call `F(x) ~= F(_)` will fail. Similarly, `x ~= [_|_]` means `ALL [u,v] x ~= [u|v]`, where `u` and `v` are unique variables. Thus the call S `~= [_|_]` succeeds if S is the empty list and fails if it is a non-empty list. Note that `x ~= [_|_]` should be distinguished from `x ~= [y|z]`, for which the `y` and `z` are treated as ordinary variables. Thus, if, for example, `x ~= [y|z]` occurs as a top-level conjunct in the body of a statement, and `y` and `z` do not occur elsewhere in the statement, then it is equivalent to `SOME [y,z] x ~= [y|z]`.

In addition to the built-in predicates `=` and `~=`, the system also has two built-in propositions, `True` and `False`. The definition of `True` is

```
True.
```

The proposition `False` has the empty definition. `True` and `False` are built into the system and are available in every module without being imported. So the complete list of built-in propositions and predicates is `True`, `False`, `=`, and `~=`.

4.2 Integers

Gödel provides a comprehensive set of functions and predicates with numerical arguments via the modules `Integers`, `Rationals`, and `Floats`. We discuss the module `Integers` in this section and the modules `Rationals` and `Floats` in subsequent sections.

The base type `Integer` is provided by the module `Integers`. The export part (except for the control declarations) of `Integers` is given below. The complete export part of `Integers` is given in chapter 13. The export part of a module contains language declarations for symbols that it makes available to other modules which import from it. The keyword `EXPORT` indicates the export part of a module.

The export part of the `Integers` "declares" the constants `0`, `1`, `2`, and so on. Since there are infinitely many non-negative integers and since the syntax does not support the declaration[1] of infinitely many symbols of any kind, this "declaration" is simply indicated by a comment in the export part of `Integers`. The system knows that `Integers` is to be treated specially in this regard. The export part of `Integers` also contains declarations of the prefix unary function `-` (minus), and the infix binary functions `^` (exponentiation), `*` (multiplication), `Div` (integer quotient), `Mod` (modulus), `+` (addition), and `-` (subtraction), each of whose arguments has type `Integer` and which map to an element of type `Integer`. `Integers` declares the unary function `Abs` (absolute value) and the binary functions `Max` (maximum), and `Min` (minimum). It also declares the binary predicates `>`, `<`, `>=`, `=<`, and the ternary predicate `Interval` all having arguments of type `Integer`. The intended meaning of `Interval` is that its second argument is greater than or equal to its first argument and less than or equal to its third argument.

[1]Such a syntax could be provided, but we do not consider it important enough just for this one case.

```
EXPORT        Integers.

BASE          Integer.

% CONSTANT      0, 1, 2, ... : Integer.

FUNCTION      ^ : yFx(540) : Integer * Integer -> Integer;
              - : Fy(530) : Integer -> Integer;
              * : yFx(520) : Integer * Integer -> Integer;
              Div : yFx(520) : Integer * Integer -> Integer;
              Mod : yFx(520) : Integer * Integer -> Integer;
              + : yFx(510) : Integer * Integer -> Integer;
              - : yFx(510) : Integer * Integer -> Integer;
              Abs : Integer -> Integer;
              Max : Integer * Integer -> Integer;
              Min : Integer * Integer -> Integer.

PREDICATE     > : zPz : Integer * Integer;
              < : zPz : Integer * Integer;
              >= : zPz : Integer * Integer;
              =< : zPz : Integer * Integer;
              Interval : Integer * Integer * Integer.
```

The module `Integers` conforms to the standard for the data type Integer in the Language Independent Arithmetic Standard (LIAS) [1]. This is a draft ISO standard which specifies the essential properties of integer and floating-point numbers that can be relied on in writing portable software. An implementation of Gödel must therefore conform to this standard. Furthermore, an implementation of the module `Integers` must be sound, that is, the body of a goal concerning the integers with a computed answer applied should be true in the intended interpretation.

To illustrate how equality for the integers is used, suppose a module which imports `Integers` defines a predicate `P` as follows.

```
PREDICATE     P : Integer * Integer.

P(x,x).
```

Then a goal such as

```
<- P(3 + 4, (17 Div 6) + 5).
```

succeeds. The reason it succeeds is, of course, because `3 + 4 = (17 Div 6) + 5` is true in the intended interpretation. Similarly, the goal

```
<- P(2*x-3, 5)
```

gives the answer `x = 4`.

As an example of the use of `Integers` consider the module `GCD` below. The predicate `Gcd` is intended to be true when its first and second arguments are integers and its third argument is the greatest common divisor of the first two arguments. Note that `GCD` contains essentially a *specification* of `Gcd` rather than an efficient algorithm, such as the Euclidean algorithm. Note also that the system provides the usual mathematical syntax for expressing ranges of integers. A range expression of the form r `=<` s `=<` t, where r, s and t are terms of type `Integer`, is compiled into the call `Interval(`r`,`s`,`t`)`. Range expressions involving `<` are also be compiled into calls to `Interval` by adjusting appropriate arguments of the expression by `1` so that `=<` can be used instead of `<`.

```
MODULE      GCD.

IMPORT      Integers.

PREDICATE   Gcd : Integer * Integer * Integer.

Gcd(i,j,d) <-
            CommonDivisor(i,j,d) &
            ~ SOME [e] (CommonDivisor(i,j,e) & e > d).

PREDICATE   CommonDivisor : Integer * Integer * Integer.

CommonDivisor(i,j,d) <-
            IF (i = 0 \/ j = 0)
            THEN
              d = Max(Abs(i),Abs(j))
            ELSE
              1 =< d =< Min(Abs(i),Abs(j)) &
              i Mod d = 0 &
              j Mod d = 0.
```

Gödel can solve systems of (not necessarily linear) constraints which involve integers, variables which range over bounded intervals of integers, and the functions and predicates exported by `Integers` (including, of course, `=` and `~=`). The precise constraint solving capabilities available in Gödel are not specified but are implementation dependent. However, we give some examples of what could be considered as "minimal" capabilities that should be provided. For example, for the program consisting of the module `Integers`, the goals

```
<- 32*4 >= (130 Mod 4).

<- x + 43 = 73 + (34 Mod 4).
<- 0 =< x =< 10  &  0 < y =< 10  &  35*x + 33*y =< 34.

<- 0 < x =< 10  &  x ~= 2  &  x^2 < 12.
```

should succeed with the answer `x = 32` for the second goal, answer `x = 0` and `y = 1` for the third goal, and answers `x = 1` and `x = 3` for the fourth goal.

One can also find Pythagorean numbers using the goal

```
<- x^2 + y^2 = z^2  &  0 < x < 50  &  0 < y < 50  &  0 < z.
```

This goal should give the answers

```
x = 3
y = 4
z = 5

x = 4
y = 3
z = 5

x = 5
y = 12
z = 13
```

and so on.

Note also that infinite precision integer arithmetic is provided. The only limit to the complexity of an arithmetic computation is the limit on the memory available to the Gödel system. A computation involving very large integers may eventually exhaust the stack or heap space available, in which case the computation will halt with an error message. Also, an attempt to divide by zero will cause the computation to halt with an error message.

4.3 Rationals

Next we discuss the module `Rationals` which provides a similar collection of functions and predicates with rational arguments, as does `Integers` for integer arguments. The intended domain is the rationals Q. The various functions, such as `+`, `-`, etc., have their usual interpretation as mappings from $Q \times Q$ (or Q, as appropriate) into Q. Similarly, the various predicates, such as `>`, `<`, etc., have their usual interpretation on $Q \times Q$.

The function `//` of type `Integer * Integer -> Rational` is the function which is used in the usual mathematical construction of the rationals from the integers. The notation `N//M` means the rational with numerator `N` and denominator `M`. As a useful notational convention, a rational of the form `N//1` can be written more simply as `N`. The system will assume by default that a number `N`, which could be an integer or a rational, is a rational if no other information can resolve the ambiguity. Since rational division in Gödel is denoted by `/`, this convention means that, in many situations, the usual mathematical notation for rationals can be used instead of `//`. For example, it is possible to write `2/3` with the usual mathematical meaning. Here the numerator

```
EXPORT       Rationals.

IMPORT       Integers.

BASE         Rational.

FUNCTION     // : yFx(520) : Integer * Integer -> Rational;
             ^ : yFx(540) : Rational * Integer -> Rational;
             - : Fy(530) : Rational -> Rational;
             * : yFx(520) : Rational * Rational -> Rational;
             / : yFx(520) : Rational * Rational -> Rational;
             + : yFx(510) : Rational * Rational -> Rational;
             - : yFx(510) : Rational * Rational -> Rational;
             Abs : Rational -> Rational;
             Max : Rational * Rational -> Rational;
             Min : Rational * Rational -> Rational.

PREDICATE    > : zPz : Rational * Rational;
             < : zPz : Rational * Rational;
             >= : zPz : Rational * Rational;
             =< : zPz : Rational * Rational;
             StandardRational : Rational * Integer * Integer.
```

2 is understood as the rational `2//1` and the denominator `3` as the rational `3//1`. Thus `2/3` is indeed the rational `2//3`. Using these conventions, for the program consisting of the modules `Rationals` and `Integers`, the goal

```
<- x = 5 + 9/7.
```

succeeds with answer `x = 44/7` and the goal

```
<- x = 4/3 + 2/3.
```

succeeds with answer `x = 2`.

When giving answers, the system returns (non-zero) rationals in the form `N/M`, where `M>0` and `Gcd(N,M) = 1`. For example, the goal

```
<- x = 120/25.
```

succeeds with answer `x = 24/5` and the goal

```
<- x = -4/(-5).
```

succeeds with answer `x = 4/5`.

Notice that a goal such as

```
<- x//y = 4/5.
```

has infinitely many answers and hence may flounder. (This depends on the implementation.) However, Gödel does provide a way to get the numerator and denominator of a rational in standard form. Let us say a rational is in *standard form* if the greatest common divisor of the numerator and the denominator is `1`, and the denominator is positive. Then the predicate `StandardRational` is intended to be true when its second and third arguments are the numerator and denominator of the (unique) rational in standard form equal to the rational in the first argument. Thus the goal

```
<- StandardRational(34/(-8),x,y).
```

gives the answer `x = -17` and `y = 4`.

Gödel solves systems of *linear* rational constraints involving rationals, variables of type `Rational`, and the functions and predicates exported by `Rationals`. For this class of constraints, there are polynomial time algorithms for deciding solvability (see, for example, [14]). However, the precise constraint solving capabilities provided are implementation dependent. The following examples give an indication of what may be provided. For example, the goals

```
<- 3*x + 2*y = 3  &  (7/2)*x + y = 0.

<- 4*x + 5 >= 16  &  -x =< 2.

<- 3*x + 4*y = 1  &  x - 2*y = 2  &  x >= 0.
```

should succeed with answer `x = -3/4` and `y = 21/8` for the first goal, answer `x >= 11/4` for the second goal, and answer `x = 1` and `y = -1/2` for the third goal. Note that infinite precision rational arithmetic is provided.

4.4 Floats

The module `Floats` provides floating point numbers and a large set of functions and predicates with floating-point arguments. Part of the export part of `Floats` is given below. The complete export part of `Floats` is given in chapter 13. The module `Floats` conforms to the standard for the data type Floating-Point in LIAS. It also conforms to the ANSI/IEEE Standard for Binary Floating-Point Arithmetic 754–1985 [2]. An implementation of this module must therefore conform to both these standards.

The intended interpretation of the symbols in this module is as follows. The intended domain of the interpretation is the finite set F of floating-point numbers characterised by a fixed radix, a fixed precision, and fixed smallest and largest exponent. Thus F is the finite set of numbers of the form either 0 or $\pm 0.f_1...f_p * r^e$, where r is the radix, p is the precision, e is the exponent, and each f_i satisfies $0 \leq f_i < r$. Note that the LIAS boolean *denorm* is true. Thus denormalised floating-point numbers are provided.

The language of `Floats` contains finitely many constants, exactly one corresponding to each floating-point number in F. However, for the convenience of the user, there is some syntactic sugar used instead of the names of these constants. This is the usual decimal number notation, with or without an exponent. Typical decimal numbers without exponent are `3.1416` and `0`, and typical

```
EXPORT      Floats.
% Part of the export part of Floats.

IMPORT      Integers.

BASE        Float.

% CONSTANT   Finitely many constants, one for each number in the finite set F
% of floating-point numbers determined by the radix, precision, and smallest and
% largest exponent.

FUNCTION    ^ : yFx(540) : Float * Float -> Float;
            - : Fy(530) : Float -> Float;
            * : yFx(520) : Float * Float -> Float;
            / : yFx(520) : Float * Float -> Float;
            + : yFx(510) : Float * Float -> Float;
            - : yFx(510) : Float * Float -> Float;
            Abs : Float -> Float;
            Max : Float * Float -> Float;
            Sqrt : Float -> Float;
            Sign : Float -> Float;
            Fraction : Float -> Float;
            Scale : Float * Integer -> Float;
            Successor : Float -> Float;
            Truncate : Float * Integer -> Float;
            Round : Float * Integer -> Float;
            IntegerPart : Float -> Float;
            Sin : Float -> Float;
            ArcSin : Float -> Float;
            Exp : Float -> Float.

PREDICATE   IntegerToFloat : Integer * Float;
            TruncateToInteger : Float * Integer;
            Exponent : Float * Integer;
            Radix : Integer;
            MaxExponent : Integer;
            MaxFloat : Float;
            Epsilon : Float;
            > : zPz : Float * Float.
```

decimal numbers with exponent are `-2.345619E-12` and `674328.89E+2`. Such decimal numbers are converted (according to the ANSI/IEEE standard 754–1985) by the system to floating-point numbers in the form above. Then the convention is that a decimal number is syntactic sugar for the constant whose interpretation is the floating-point number obtained from the decimal number. This means that there is more than one way of denoting each of these constants. For example, both `3.1416` and `314.16E-2` denote the same constant. Similarly, when answers are displayed by the system, floating-point numbers are converted back to the more convenient decimal form.

This module is intended for straightforward numerical computation and no (non-trivial) constraint solving capabilities are provided. For example, for the program consisting of the modules `Floats` and `Integers`, the goals

```
<- x = Sin(Sqrt(23.45812E+23) + 4.56E-13/0.432167E-2).

<- Exp(-2.0389E+1) < ArcSin(0.7349001).
```

both succeed, with answer `x = -0.99172602406708` for the first goal. Similarly, suppose a module which imports `Floats` defines a predicate `P` as follows.

```
PREDICATE    P : Float * Float.

P(x,x).
```

Then a goal such as

```
<- P(3.345 + 4.89E+2, x + 23.8889).
```

succeeds with answer `x = 468.4561`.

4.5 Numbers

The module `Numbers` is provided as a convenient way of loading all the number modules together. The export part of `Numbers` is given below. It also contains two predicates for converting between rationals and floating-point numbers. The conversion functions for integers/rationals are in `Rationals` and the conversion predicates for integers/floats are in `Floats`.

```
EXPORT       Numbers.

IMPORT       Rationals, Floats.

PREDICATE    RationalToFloat : Rational * Float;
             FloatToRational : Float * Rational.
```

Chapter 5

Modules

Gödel is intended to be used for the development of large and complex programs. For this, there needs to be a means of decomposing a program into smaller components so these components can be developed independently, as far as possible. Also there needs to be a way of avoiding interference between the components because of name clashes and a way of hiding the implementation details of the components. In Gödel, these components are called modules. We now turn to an explanation of the Gödel module system, which satisfies these requirements.

5.1 Importing and Exporting

In general, modules consist of two parts, an export part and a local part. The *export part* of a module is indicated by an export declaration, which is either an `EXPORT` or `CLOSED` declaration. The *local part* of a module is indicated by a local declaration, which is either a `LOCAL` or `MODULE` declaration. In these declarations, the keywords `EXPORT`, `CLOSED`, `LOCAL` and `MODULE` 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. The other kind of module declaration is the import declaration, which is either an `IMPORT` or `LIFT` declaration. In these declarations, the keywords `IMPORT` and `LIFT` are followed by the name of a module.

The export part of a module begins with an export declaration, and contains zero or more import declarations, language declarations, and control declarations. The local part of a module begins with a local declaration, and contains zero or more import declarations, language declarations, control declarations, and statements. If a module consists *only* of a local part, then this is indicated by using a `MODULE` declaration instead of a `LOCAL` declaration. The use of a `CLOSED` declaration instead of an `EXPORT` declaration will be explained shortly.

We now introduce the concepts of importation, accessibility, and exportation in an informal way, ignoring the concept of type lifting which is discussed later. The precise definition of these concepts is given in section 5.2.

We say a part of a module *declares* a symbol if it contains a language declaration for that symbol. We say a module *declares* a symbol if either the local or export part of the module declares the symbol. We say a part of a module *imports* a symbol if the part contains an `IMPORT` declaration with the module name `N`, say, and either the module `N` declares this symbol in its export part or, inductively, `N` imports the symbol into its export part. We say a symbol is *accessible to* the local (resp., export) part of a module if it is either declared in, or imported into, a part of the module (resp., the export part of the module). We say a module *exports* a symbol if the symbol is accessible to the export part of the module.

Subject to the module conditions given in section 5.2, a symbol accessible to a part of a module is available for use in that part of the module. More precisely, a base or constructor accessible to a part of a module can appear in a `CONSTANT`, `FUNCTION`, or `PREDICATE` declaration in that part of the module. Similarly, a constant, function, proposition, or predicate accessible to a part of a module can appear in a statement in the module. Of course, a symbol can only be used according to its language declaration.

We illustrate these concepts with the modules `M5` and `M6` below. Module `M5` has an export part and a local part. The export part of `M5` makes all the symbols it declares or imports available for use by other modules, such as `M6`. In particular, the declarations for the bases `Day` and `Person`, the declarations for the constants `Monday`, `Fred`, and so on, and the declaration for the predicate `Append3` make these symbols available for use by other modules which import `M5`. The `IMPORT` declaration in the export part of `M5` makes the symbols exported by `Lists` available for use in `M5`. It also makes the symbols exported by `Integers` available for use in `M5`, since the export part of `Lists` imports `Integers`. Any module which imports `M5` automatically imports all the symbols exported by `Lists` and `Integers` and hence does not need to explicitly import `Lists` or `Integers` to make these available for use. The local part of `M5` contains the definition of the predicate `Append3`, which uses the definition of `Append` from `Lists`.

```
EXPORT       M5.

IMPORT       Lists.

BASE         Day, Person.

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

```
LOCAL        M5.

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

By contrast with module `M5`, module `M6` has only a local part. Hence its first module declaration uses the keyword `MODULE` instead of `LOCAL`. It imports all the symbols exported by `M5`, which include `Append3` together with all the symbols exported by `Lists` and `Integers`. Module

```
MODULE      M6.

IMPORT      M5.

PREDICATE   Member2 : a * a * List(a).

Member2(x,y,z) <-
            Append3(_,[x|_],[y|_],z).
```

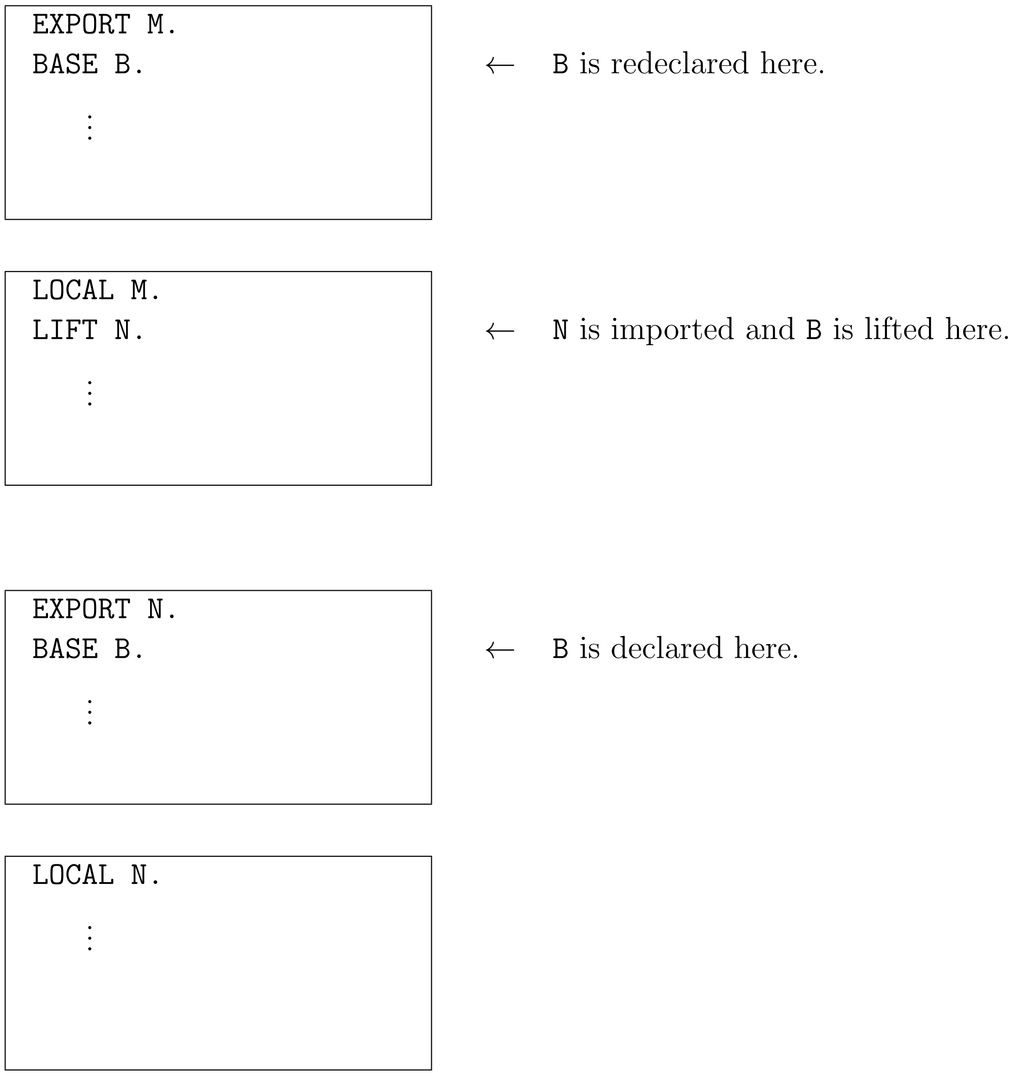
M6 contains the definition of the predicate Member2. Member2(x,y,z) is intended to be true if and only if z is a list which contains x and y as members so that x precedes y in the list.

Informally, a *program* consists of a set of modules $\{\texttt{Mi}\}_{\texttt{i=0}}^{\texttt{n}}$ ($\texttt{n} \geq 0$), where M0 is a distinguished module called the *main* module and $\{\texttt{Mi}\}_{\texttt{i=1}}^{\texttt{n}}$ is the set of modules which are either imported by M0 or are imported by modules imported by M0, and so on. (A formal definition is given in section 5.4 and chapter 11.) For the above example, {M6, M5, Lists, Integers} is a program with main module M6.

Certain modules are provided by the system and hence are called *system modules*. For example, Integers and Lists are system modules. The complete set of system modules is given in chapter 13. Other modules are called *user-defined modules*. Note that the export declaration in the export part of a system module may contain the keyword CLOSED instead of EXPORT. A module is *closed* if its export part contains a CLOSED declaration and *open* if it is not closed. In closed modules the visibility of symbols accessible to the local part of the module is restricted. (The precise restrictions are given later.) This effectively increases the range of implementations available. For example, the local part of a closed module does not have to be implemented in Gödel. An implementation of Gödel may make some or all of the system modules closed. However, user-defined modules must be open.

In general, symbols declared in, or imported into, the local part of a module are not available for use in the export part of the module. However, there are occasions when it is desirable that selected bases and constructors imported into the local part of a module should also be available for use in its export part (and hence other modules that import it). For this reason, Gödel has a facility called *type lifting*. We illustrate how type lifting works in figure 5.1 below. Suppose B is a base declared in the export part of N and we want to lift B from the local to the export part of M. Then N would be imported into the local part of M, but via a LIFT declaration instead of an IMPORT declaration. The LIFT declaration imports N, as an IMPORT declaration does, but also provides the opportunity to lift bases and constructors exported by N so as to make them accessible to the export part of M. This is achieved by redeclaring in the export part of M the bases and constructors that are to be lifted. For the example, the base B is redeclared in the export part of M. A base or constructor is *redeclared* by repeating its declaration in the export part of the module in which it is being lifted. The symbol is thereby made accessible to the export part of the module. Only bases and constructors can be lifted.

Type lifting has several desirable properties which we illustrate with the example. First, the fact that the base B is exported from M is made explicit by repeating its declaration in the export



```
EXPORT M.
BASE B.              ←  B is redeclared here.
    ⋮
```

```
LOCAL M.
LIFT N.              ←  N is imported and B is lifted here.
    ⋮
```

```
EXPORT N.
BASE B.              ←  B is declared here.
    ⋮
```

```
LOCAL N.
    ⋮
```

Figure 5.1: Type Lifting

part of M. In fact, since a user who only has access to the export part of M cannot tell whether the declaration declares or redeclares B and since it makes no difference anyway, the user might just as well assume B is actually declared there. Second, we can change the details of N without affecting in any way the users of M. This includes changing the name of N or even removing N altogether, if this became possible. Third, other symbols exported by N which are needed just for the implementation of M are hidden from the users of M.

5.2 Module Declarations and Conditions

We now give a precise meaning to the various module declarations and also give the module conditions which every module must satisfy.

A *symbol* is a base, constructor, constant, function, proposition, or predicate.

A *type symbol* is a base or constructor. Other symbols are *non-type symbols*.

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

```
IMPORT    N.
```

or

```
LIFT      N.
```

A module `M` *refers to* a module `N` if either the local or export part of `M` refers to `N`.

The local part of a module `M` *1-imports* a symbol `S` *via* a module `N` if `S` has a declaration in the export part of `N` and the local part of `M` refers to `N`.

The export part of a module `M` *1-imports* a symbol `S` *via* a module `N` if `S` has a declaration in the export part of `N` and either (i) the export part of `M` refers to `N`, or (ii) the local part of `M` refers to `N` via a `LIFT` declaration, `S` is a type symbol, and there is a declaration for `S` in the export part of `M`. In the latter case, the export part of `M` also *1-redeclares* `S`.

The local part of a module `M` *n-imports*, $n > 1$, a symbol `S` *via* a module `N` if there is a module `L` such that the export part of `L` $(n-1)$-imports `S` via `N` and the local part of `M` refers to `L`.

The export part of a module `M` *n-imports*, $n > 1$, a symbol `S` *via* a module `N` if there is a module `L` such that the export part of `L` $(n-1)$-imports `S` via `N` and either (i) the export part of `M` refers to `L`, or (ii) the local part of `M` refers to `L` via a `LIFT` declaration, `S` is a type symbol, and there is a declaration for `S` in the export part of `M`. In the latter case, the export part of `M` also *n-redeclares* `S`.

A part of a module `M` *imports* a symbol `S` *via* a module `N` if the part of `M` n-imports `S` via `N`, for some $n \geq 1$.

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

The export part of a module *redeclares* a type symbol `S` if it n-redeclares `S`, for some $n \geq 1$.

The export part of a module *declares* a type symbol if it contains a declaration for, but does not redeclare, the symbol. It *declares* a non-type symbol if it contains a declaration for the symbol.

The local 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 part of `M` imports `S` via `N` and the export part of `N` declares `S`.

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 is *accessible to* the local (resp., export) part of a module if it is either declared in, or imported into, the module (resp., export part of the module).

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

Now let us turn to the *module conditions*, M1 to M5. The first of these ensures that the module structure of a program is non-circular and hence greatly simplifies compilation. To state this condition, we introduce a new concept. We define the relation *depends upon* between modules to be the transitive closure of the relation refers to. So, for example, module `M6` depends upon the modules `M5`, `Lists`, and `Integers`.

M1: No module may depend upon itself.

The next module condition ensures that, in any given module part, the only names that can appear are names of symbols that are accessible.

M2: For every name appearing in a part of a module, there must be a symbol having that name accessible to that part.

Overloading is a useful programming language facility. For example, it is convenient to use the name `-` for both unary and binary minus, and the name `+` for addition of integer, rational, and floating point numbers. For this reason, Gödel has a flexible scheme for overloading. The only condition on naming symbols enforced by the system is the following module condition (in which the arity of a base, constant, or proposition is 0).

M3: Distinct symbols cannot be declared in the same module with the same category, name, and arity.

To ensure that, for example, every constant of type a base, `B` say, and every function of range type `B` is declared in the module where `B` is declared, the following module condition is enforced.

M4: In a module, a 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.

For example, module condition M4 means that a user-defined module, which imports `Lists`, is not allowed to declare a constant of type `List(`τ`)`, for any type τ.

To ensure that no definition of a proposition or predicate can be split across several modules, the following final condition is enforced.

M5: A module must declare every proposition or predicate defined in that module.

5.3 Ambiguity

The enforcement of module condition M3 avoids gratuitous ambiguity, but at the same time allows considerable flexibility in overloading. The system uses all type and indicator information available to try to disambiguate the use of a name which could refer to different symbols and, in most cases, will be able to resolve the ambiguity. In those circumstances where it cannot resolve the ambiguity, an overloading error message is issued. Note carefully that a precondition for an overloading error is the *use* of a name for which there is more than one accessible symbol having this name. By the *use* of a name in a module we mean any appearance of the name except as the identifier after the keyword of a language declaration. So, for example, it is always legal for a module to import two distinct symbols with the same name or to import a symbol with the same name as a symbol declared in the module if the name of these symbols is not used anywhere in the module. This requirement reduces the reporting of overloading errors to those that cause genuine difficulty.

Another cause of ambiguity is the use of operators, which can lead to ambiguities in the parsing of expressions. Suppose a local part of a module has two symbols accessible to it with the following declarations.

```
PREDICATE    P : zPz : Integer * Integer.
FUNCTION     P : xFy(500) : Integer * Integer -> Integer.
```

Then the expression `u P v P w`, used for example in a place where an atom should appear, is ambiguous.

Also, if the function with name `F1` and indicator `yFx(500)`, and the function with name `F2` and indicator `xFy(500)` are accessible to the same module, then the expression `u F2 v F1 w` is ambiguous. Similarly, if the function with name `F3` and indicator `yF(400)`, and the function with name `F4` and indicator `Fy(400)` are accessible to the same module, then the expression `F4 u F3` is ambiguous.

As another example of a situation where module condition M3 is satisfied but where an ambiguity cannot be resolved, consider the modules `M7`, `M8`, and `M9` below. The system has no way of determining which symbol with name `Q` is the one intended in the body of the statement for `P`. The only way to fix the problem in modules `M7`, `M8`, and `M9` is to change the name of one of the predicates with name `Q` in modules `M8` and `M9`.

```
MODULE       M7.

IMPORT       M8, M9.

PROPOSITION  P.

P <-
             Q(x).
```

```
EXPORT       M8.

BASE         B1.

PREDICATE    Q : B1.
```

```
EXPORT       M9.

BASE         B2.

PREDICATE    Q : B2.
```

5.4 Programs and Goals

A *program* consists of a set of modules $\{\texttt{Mi}\}_{\texttt{i=0}}^{\texttt{n}}$ ($\texttt{n} \geq \texttt{0}$), where $\{\texttt{Mi}\}_{\texttt{i=1}}^{\texttt{n}}$ is the set of modules upon which `M0` depends and where each module satisfies the module conditions given in section 5.3. The module `M0` is called the *main* module of the program.

To define a statement, we introduce the language of a module in 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`.[1]

We will also need the export language of a module.

- 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`.

To define a goal, we introduce the goal language of a program.

- The *goal language of* a program `P` is the language of the main module `M0` of `P`, if `M0` is open, or the export language of `M0`, if `M0` is closed.

A *statement*[2] in a module `M` in a program is a formula in the language of `M` having the form either `A` or `A <- W`, where `A` is an atom, called the *head* of the statement, and where `W` is a formula, called 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. Each statement in a module must satisfy the head condition given in chapter 2.

A *goal*[2] for a program has the form `<- W`, where `W` is a formula in the goal language of the program. `W` is called the *body* of the goal. Any free variables in `W` are assumed to be universally quantified at the front of the goal.

As an example, typical goals for the program {`M6`, `M5`, `Lists`, `Integers`} could be

```
<- Member2(x,y,[Fred,Bill,Mary]).

<- Member2(1,4,[56,7,3,4,5]).

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

If a variable in a goal becomes bound to a term which contains a subterm not in the goal language of the program, then only the type of the subterm is given in an answer. (There is one exception to this rule. If the term is of type `String`, then the string is printed in its external syntax. See chapter 6.) For example, consider the program consisting of the modules `Hidden1` and `Hidden2` below. The goal

[1]Some care must be taken with this definition and the two that follow. The reason is that, because of ambiguity, a symbol may not be uniquely identified by its name. For example, there is a predicate with name `Q` declared in the module `M8` and another predicate with the same name declared in module `M9` and both these symbols appear in the language of `M7`. It is understood that a formula is only admitted by the language of a module if the use of each name in the formula can be disambiguated. Later, for the ground representation (see chapter 9), we will introduce flat names to avoid such problems. The flat name of a symbol is a quadruple consisting of the name of the module in which the symbol is declared, the name of the symbol, its category, and its arity. Module condition M3 ensures that the flat name of a symbol in a program uniquely identifies it.

[2]This definition will be generalized in chapter 7 to the case when commits are present in the body.

```
<- Father(x).
```

produces the answers

```
x = Individual(Bill)
x = Individual(<Name>)
```

where the phrase `<Name>` indicates that the argument of `Individual` is a term with a top-level constant or function which is not accessible and that this term has type `Name`.

```
MODULE       Hidden1.

IMPORT       Hidden2.
```

```
EXPORT       Hidden2.

BASE         Name, Person.

CONSTANT     Bill : Name.

FUNCTION     Individual : Name -> Person.

PREDICATE    Father : Person.
```

```
LOCAL        Hidden2.

CONSTANT     Fred : Name.

Father(Individual(Bill)).
Father(Individual(Fred)).
```

Chapter 6

Various Data Types

In this chapter, we discuss six fundamental data types: lists, strings, sets, tables, units, and flocks; and also the system modules which provide them.

6.1 `Lists`

The symbols exported by the system module `Lists` are shown below in its export part (which is complete except for the control declarations). The constructor `List` is used to give a type to domains whose elements are lists. Thus, for example, the domains of the intended interpretation for the program {`Lists`, `Integers`} are the integers (having type `Integer`), all lists of integers (having type `List(Integer)`), all lists of lists of integers (having type `List(List(Integer))`), and so on. Note that the elements of a list must all be of the same type, so, for example, it is not possible for a list to have some of its elements integers and some of its elements lists of integers. If a programmer would like to form a list having some elements of type `A` and some of type `B`, say, then this can in essence be achieved, as follows. Declare a new type, say,

```
BASE          Element.
```

and two new functions, say,

```
FUNCTION      F : A -> Element;
              G : B -> Element.
```

Then a list of elements of type `Element` can be formed, each member of which "packages up" an element of type `A` or `B`. In fact, the existence of `F` and `G` facilitates the processing of the elements of such lists and is good programming practice.

The module `Lists` provides a collection of useful list processing predicates, the details of which can be found in chapter 13. As an illustration of the use of lists, consider the module `Qsort` below. The predicate `Quicksort` in `Qsort` is intended to be true when its first argument is a list of integers and its second argument is the list of integers occurring in the first argument sorted into non-decreasing order. The predicate `Partition` is intended to be true when its first argument is a list L of integers, its second argument is an integer n, its third argument is the list of integers in L which are less than or equal to n, and its fourth argument is the list of integers in L which are greater than n.

```
EXPORT       Lists.

IMPORT       Integers.

CONSTRUCTOR  List/1.

CONSTANT     Nil : List(a).

FUNCTION     Cons : a * List(a) -> List(a).

PREDICATE    Member : a * List(a);
             MemberCheck : a * List(a);
             Append : List(a) * List(a) * List(a);
             Permutation : List(a) * List(a);
             Delete : a * List(a) * List(a);
             DeleteFirst : a * List(a) * List(a);
             Reverse : List(a) * List(a);
             Prefix : List(a) * Integer * List(a);
             Suffix : List(a) * Integer * List(a);
             Length : List(a) * Integer;
             Sorted : List(Integer);
             Sort : List(Integer) * List(Integer);
             Merge : List(Integer) * List(Integer) * List(Integer).
```

6.2 Strings

Next we turn to strings. A base type `String` is defined in the system module `Strings` which provides a collection of useful predicates for processing strings. The export part of `Strings` (except for the control declarations) is given below. Note that there is no type for characters.

Strings are treated as an abstract data type. Thus it is not possible to directly access or display the constants and functions used in the (internal) representation of strings. However, as a convenience, Gödel provides the usual double quotes notation for strings as a kind of external syntax for them. Thus `"ABC"` is the external syntax for the string whose first character is `A`, second character is `B`, and third character is `C`. Similarly, `""` is the external syntax for the empty string. A double quote can appear in a string by escaping it with a \, as in `"This is a string with a \" in it."`. To include a \ in a string, use \\. A single \ (not escaping a \ or a ") in a string is ignored. The predicate `StringInts` allows conversion between a string and the list of ASCII codes of characters in the string. Thus the call `StringInts(x, [77,111,110,100,97,121])` would bind `x` to the string whose external syntax is `"Monday"`.

This external syntax can be used to input string arguments in the following way. Suppose a predicate `P` having declaration

```
PREDICATE  P : String * ... .
```

```
MODULE      Qsort.

IMPORT      Lists.

PREDICATE   Quicksort : List(Integer) * List(Integer);
            Partition : List(Integer) * Integer * List(Integer) * List(Integer).

Quicksort([],[]).
Quicksort([x|xs],ys) <-
            Partition(xs,x,l,b) &
            Quicksort(l,ls) &
            Quicksort(b,bs) &
            Append(ls,[x|bs],ys).

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

appears in a program. Then an atom of the form `P("abc",...)`, for example, may appear in a statement or goal. Such an atom is expanded (in principle) by the system into

`StringInts(x,[97,98,99]) & P(x,...).`

This convention gives users all the convenience of the usual syntax for inputting strings and, at the same time, respects the Gödel module conditions.

For output at the top level of the system, if the binding for a variable of type string in a goal has to be written, the system intervenes by calling the predicate `WriteString` from the `IO` module (see chapter 8) to write it. Thus what is being written is not the term the variable is bound to, but the external syntax for the term. For stand-alone programs, `WriteString` would be called directly by the program to write out any strings.

The function `++` is used to concatenate strings, so that the call `x = "MOD" ++ "ULE"` would bind `x` to the string `"MODULE"`. The predicate `Width` gives the number of characters in a string, so that the call `Width("Fred" ++ ".prm", x)` would bind x to `8`. The predicate `<` is lexical less than for strings, `>` is lexical greater than, and so on.

6.3 Sets

By a "set", we understand the usual mathematical concept. However, for reasons of simplicity, Gödel only handles *finite* sets. Thus, "set" means "finite set" throughout. The export part

```
EXPORT     Strings.

IMPORT     Lists.

BASE       String.

FUNCTION   ++ : String * String -> String.

PREDICATE  StringInts : String * List(Integer);
           FirstSubstring : String * Integer * String;
           LastSubstring : String * Integer * String;
           Width : String * Integer;
           > : zPz : String * String;
           < : zPz : String * String;
           >= : zPz : String * String;
           =< : zPz : String * String.
```

(excluding the control declarations) of the system module `Sets` is given below. `Sets` exports the constructor `Set`, which plays a role for sets that is analogous to the role played by the constructor `List` for lists. Thus, for example, the domains of the intended interpretation for the program {`Sets`, `Integers`} are the integers (having type `Integer`), all sets of integers (having type `Set(Integer)`), all sets of sets of integers (having type `Set(Set(Integer))`), and so on.

Two kinds of set terms are allowed: extensional set terms and intensional set terms. We first consider extensional set terms. The module `Sets` exports the constant `Null` and the function `Inc`, which are used to form extensional set terms. (This is similar to the treatment of finite sets in [6].) The intended meaning of `Null` is the empty set (of the appropriate type). The intended meaning of `Inc` (which is short for `Include`) for some type τ is the mapping Inc such that $Inc(d, S) = \{d\} \cup S$, where d is an element from the domain of type τ and S is a set of elements of type τ. Thus, the intended meaning of the term

```
Inc(6,Inc(5,Inc(6,Null)))
```

is the set $\{5, 6\}$.

In a similar way to lists, some notational sugar for sets is provided. An expression of the form

```
{}
```

stands for the constant

```
Null.
```

An expression of the form

```
{t1,...,tn}
```

stands for the extensional set term

```
EXPORT       Sets.

IMPORT       Integers.

CONSTRUCTOR  Set/1.

CONSTANT     Null : Set(a).

FUNCTION     Inc : a * Set(a) -> Set(a);
             * : yFx(120) : Set(a) * Set(a) -> Set(a);
             + : yFx(110) : Set(a) * Set(a) -> Set(a);
             \ : yFx(100) : Set(a) * Set(a) -> Set(a).

PREDICATE    In : zPz : a * Set(a);
             Subset : zPz : Set(a) * Set(a);
             StrictSubset : zPz : Set(a) * Set(a);
             Size : Set(a) * Integer.
```

`Inc(t1,Inc(t2,...,Inc(tn,Null)...)).`

An expression of the form

`{t1,...,tn|s}`

stands for the extensional set term

`Inc(t1,Inc(t2,...,Inc(tn,s)...)).`

It is important to appreciate that an extensional set term is *not* a set.[1] However, *its intended meaning is a set* and this is the reason for the use of the brackets `{` and `}` in the above notation.

A special equality is required for sets because the order of elements in a set is irrelevant and because sets do not have duplicate elements. Thus, for example, the following equalities are true in the intended interpretation:

`{5,6} = {6,5}`

and

`{5,6} = {5,6,6}.`

[1]Actually, we could make a similar point about lists. An expression such as `[1,2]` is not a list, but notational sugar for the term `Cons(1,Cons(2,Nil))`. However, the intended meaning of this term is a list. It is pedantic to insist on this distinction for lists, but the more complicated equality for sets makes the distinction important in the latter case. It explains why it makes sense to write such things as `{1,2,1}`, where elements are repeated.

The module `Sets` also provides some standard set processing functions and predicates. Set-theoretic intersection is given by the function `*`, union by the function `+`, and difference by the function `\`. Set membership is given by the predicate `In`, inclusion by the predicate `Subset`, and strict inclusion by the predicate `StrictSubset`. The cardinality of a set is given by the predicate `Size`.

For the program {`Sets, Integers`}, the goal

```
<- x = Inc(1,Inc(2,Inc(1+1,Null))).
```

gives the answer `x = {1,2}` and the goal

```
<- x = {1,2,3,4,5} + {4,5,6,7,8} \ {3,4,5,6}.
```

gives the answer `x = {1,2,7,8}`.

As a further example of set processing, consider the module `SetProcessing` below, which defines two predicates `Sum` and `Max`. The intended meaning of `Sum` is that the second argument is the sum of the integers in the set in the first argument. Thus the goal

```
<- Sum({1,2,3,3},s).
```

gives the answer `s = 6`. The intended meaning of `Max` is that the second argument is the maximum of the integers in the set in the first argument. Thus the goal

```
<- Max({1,2,3},m).
```

gives the answer `m = 3`. A further example of this kind of set processing appears in chapter 10, where a solution of the wolf-goat-cabbage problem is given.

Next we discuss intensional set terms. These have the form

```
{T : W}
```

where `T` is a term with free variables `y1`,...,`yn`, say, and `W` is a formula (not involving commits) which has `y1`,...,`yn` amongst its free variables. The variables `y1`,...,`yn` must be local to `{T : W}`. The free variables of `{T : W}` are the free variables of `W` other than `y1`,...,`yn`. (Note that `T` may itself be an intensional set term and that it is possible for `n` to be `0`.) Intuitively, `{T : W}` means "the set of all instances of `T` corresponding to the instances of `W` which are true".

The meaning of a term `{T : W}` appearing in a statement or goal is given by the following transformation: `{T : W}` is replaced by a new variable, `s` say, and the formula

```
ALL [x] (x In s  <->  SOME [y1,...,yn] ((x = T) & W))
```

is added as an additional conjunct to the body of the statement or goal. For a term of the form `{{T : W1} : W2}`, this transformation is first applied to `{{T : W1} : W2}` and then to `{T : W1}`, that is, the transformation is applied first to outermost intensional set terms. This semantics for intensional set terms was first given in [5]. An implementation of intensional set terms must guarantee that it is sound, although it does not have to explicitly use the above transformation, of course.

Now we give some examples of processing with intensional set terms. Consider the program {`Sets, Integers`}. Then the goal

```
MODULE     SetProcessing.

IMPORT     Sets.


PREDICATE  Sum : Set(Integer) * Integer.

Sum(s,y) <-
        x In s &
        Sum1(s\{x},x,y).

PREDICATE  Sum1 : Set(Integer) * Integer * Integer.

Sum1({},x,x).
Sum1(s,x,x+w) <-
        z In s &
        Sum1(s\{z},z,w).

PREDICATE  Max : Set(Integer) * Integer.

Max(s,y) <-
        x In s &
        Max1(s\{x},x,y).

PREDICATE  Max1 : Set(Integer) * Integer * Integer.

Max1({},x,x).
Max1(s,x,y) <-
        z In s &
        IF z>x THEN Max1(s\{z},z,y) ELSE Max1(s\{z},x,y).
```

```
<- x = {s : s Subset {z : n Mod z = 0 & 1 =< z < 10}} & n = 10.
```

gives the answer `x = {{},{1},{2},{5},{1,2},...,{1,2,5}}` and the goal

```
<- 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}}`.

As another example of the use of intensional sets, consider the module `SetProcessing` again. The goal

```
<- Sum({x : 100 Mod x = 0 & 1 < x < 100}, s).
```

gives the answer `s = 116` and the goal

```
<- Max({x^2 + y^2 : -100 =< x =< 20  &  -10 =< y =< 30}, m).
```

gives the answer `m = 10900`.

Consider finally the module `Sports` below. Then the goal

```
<- x = {p : Likes(p,Tennis)}.
```

gives the answer `x = {Mary, Bill, Joe}`, the goal

```
<- x = {s : Likes(Fred,s)}.
```

gives the answer `x = {}`, the goal

```
<- x = {p : Likes(p,s)} & s In {Cricket,Football}.
```

gives the answers

```
s = Cricket, x = {Mary,Bill}
s = Football, x = {Joe}
```

and the goal

```
<- x = {Pair(p,y) : y = {s : Likes(p,s)} & p In {Mary,Bill,Joe,Fred}}.
```

gives the answer

```
x = {Pair(Mary,{Cricket,Tennis}), Pair(Bill,{Cricket,Tennis}),
                                    Pair(Joe,{Tennis,Football}), Pair(Fred,{})}.
```

6.4 Tables

A table is a data structure containing an ordered collection of nodes, each of which has two components, a key and a value. The key must be a string, but the value can have any type. The key in a node uniquely identifies the node in the table. Furthermore, the keys in a table provide a natural ordering of the nodes via the lexical ordering.

A table is treated as an abstract data type and the system module `Tables` (see below) provides various operations on this type. `EmptyTable` is true when its argument is an empty table. `NodeInTable` is used to search for a node a table, `InsertNode` to insert a node into a table, and `DeleteNode` to delete a node from a table. `UpdateTable` is true when its first argument is a table, its second is the key of a node in the table, its third is a new value to be associated with this key, its fourth is the table with the node updated, and the fifth is the old value that was associated with the key. `AmendTable` is a variation of `UpdateTable` which can be used when the key in the second argument is not necessarily present in the table. If it is, `AmendTable` behaves similarly to `UpdateTable`. If it is not, a new node with this key is inserted into the table. (The precise definition of `AmendTable` is given in chapter 13.) To display a table as a pair of a list of keys (lexically ordered by the keys) and a list of corresponding values, the predicate `ListTable` is used. `FirstNode` gives the first node in a table and `LastNode` the last node (according to the

```
MODULE      Sports.

IMPORT      Sets.

BASE        Person, Sport, PersonSports.

CONSTANT    Mary, Bill, Joe, Fred : Person;
            Cricket, Football, Tennis : Sport.

FUNCTION    Pair : Person * Set(Sport) -> PersonSports.

PREDICATE   Likes : Person * Sport.

Likes(Mary, Cricket).
Likes(Mary, Tennis).
Likes(Bill, Cricket).
Likes(Bill, Tennis).
Likes(Joe, Tennis).
Likes(Joe, Football).
```

```
EXPORT       Tables.

IMPORT       Strings.

CONSTRUCTOR  Table/1.

PREDICATE    EmptyTable : Table(a);
             NodeInTable : Table(a) * String * a;
             InsertNode : Table(a) * String * a * Table(a);
             DeleteNode : Table(a) * String * a * Table(a);
             UpdateTable : Table(a) * String * a * Table(a) * a;
             AmendTable : Table(a) * String * a * a * Table(a) * a;
             JoinTables : Table(a) * Table(a) * Table(a);
             ListTable : Table(a) * List(String) * List(a);
             FirstNode : Table(a) * String * a;
             LastNode : Table(a) * String * a;
             NextNode : Table(a) * String * String * a;
             PreviousNode : Table(a) * String * String * a.
```

lexical ordering of keys). `NextNode` finds the node next in this ordering after a specified node and `PreviousNode` finds the previous one.

As a simple example of the use of the module `Tables`, the module `DoubleTable` below contains the definition of a predicate `Double`, which is true when its first argument is a table of nodes with integer values and its second argument is a table which differs from the first only in that all nodes whose key has width > 5 have a value double that of the value in the corresponding node in the table in the first argument. Since it is necessary to check the width of every key in the table, one way to proceed is to use `ListTable` to produce a list of the keys and a list of the corresponding values, make a pass through the lists to find the keys which have width > 5 (and their corresponding values), and use `UpdateTable` to update those nodes which have a key satisfying this condition.

```
EXPORT     DoubleTable.

IMPORT     Tables.

PREDICATE  Double : Table(Integer) * Table(Integer).
```

6.5 Units

Units are term-like data structures having the following grammar:

Unit	$\longrightarrow$	Identifier
		| Identifier <‘(’> <‘)’>
		| Identifier <‘(’> UnitSeq <‘)’>
UnitSeq	$\longrightarrow$	Unit {<Comma> Unit}

where the notation {...} indicates 0 or more occurrences of the included item and | indicates alternative forms. An identifier can be quite complicated and, for example, includes quoted and double quoted identifiers, such as `'ABC'` and `"string_identifier"`. The grammar for identifier is given in chapter 11. For example, each of the following are units:

```
F(A,x)
ABC
f(g(a,b),1,h(x,y)).
```

Gödel provides the type `Unit` as an abstract data type. Thus there is a representation of units, which is hidden from the programmer, and some operations on this type provided by the system module `Units`. (See below.) `StringToUnit` is true when its first argument is the string representation of a unit and its second argument is this unit. `UnitToString` is true when its first argument is a unit and its second argument is the string representation of this unit. `UnitParts` is true when its first argument is a unit, the second argument is the top-level identifier of this

```
LOCAL      DoubleTable.

% PREDICATE  Double : Table(Integer) * Table(Integer).

Double(table, new_table) <-
           ListTable(table, keys, values) &
           Double1(table, keys, values, new_table).

PREDICATE  Double1 : Table(Integer) * List(String) * List(Integer) *
                     Table(Integer).

Double1(table, [], [], table).
Double1(table, [k|ks], [v|vs], new_table) <-
           DoubleNode(table, k, v, next_table) &
           Double1(next_table, ks, vs, new_table).

PREDICATE  DoubleNode : Table(Integer) * String * Integer * Table(Integer).

DoubleNode(table, key, value, next_table) <-
           IF (SOME [w] (Width(key,w) & w > 5))
           THEN
             UpdateTable(table, key, 2*value, next_table, value)
           ELSE
             next_table = table.
```

unit, and the third argument is the list of arguments of a unit. For example, for the program with main module `Units`, the goal

```
<- new_string : StringToUnit("F(A,x)", unit) &
   UnitParts(unit, identifier, arguments) &
   UnitParts(new_unit, "G", arguments) &
   UnitToString(new_unit, new_string).
```

gives the answer

```
new_string = "G(A,x)".
```

If the first argument of `UnitParts` is just an identifier, then the third is the empty list. Thus the goal

```
<- arguments : StringToUnit("ABC", unit) & UnitParts(unit, identifier, arguments).
```

gives the answer

```
arguments = [].
```

```
EXPORT     Units.

IMPORT     Strings.

BASE       Unit.

PREDICATE  StringToUnit : String * Unit;
           UnitToString : Unit * String;
           UnitParts : Unit * String * List(Unit);
           UnitArgument : Unit * Integer * Unit.
```

Units are a general data structure which can be used for a variety of applications. For example, it is easy to write a propositional or first order formula as a unit. Similarly, clauses in a Prolog program can be regarded as units. In fact, the reason for the introduction of quoted and double quoted identifiers is to have identifiers general enough for the latter application. Once a formula or Prolog clause or whatever is written as a unit, it can be conveniently manipulated by the predicates in `Units`.

6.6 Flocks

Units often congregate in flocks, which are ordered collections of units. There is a system module `Flocks` (see below) which provides some operations on the abstract data type `Flock`. `EmptyFlock` is true when its argument is an empty flock and `Extent` gives the number of units in a flock. `UnitInFlock` and `UnitWithIdentifier` allow a program to access units in a flock. `InsertUnit` is used to insert a unit into a flock and `DeleteUnit` is used to delete a unit from a flock. Furthermore, there is a utility `flock-compile` which takes a file containing a flock and produces a file containing the representation of this flock employed by the abstract data type. Another utility, `flock-decompile` reverses this process. The system module `FlocksIO` provides predicates to access this representation. (This is discussed in more detail in chapter 8.)

The module `Flocks` is useful for many applications. For example, suppose that a programmer wanted to use Gödel as a meta-programming language for doing transformation of Prolog programs. First, a programmer would use a utility (which may be supplied by the system) to convert a Prolog program into a flock. (The main thing that has to be done is to write all operators in standard form.) The module `Flocks` then provides much low-level support for doing whatever transformation was required on the Prolog program. As a final step, another utility could be applied to take a flock representation of the transformed Prolog program and rewite it to the usual form. A similar procedure could be applied to other programming languages, which would also require utilities for converting between programs in the language and flocks. Theories in most logics can also be mapped rather easily into flocks. Thus the module `Flocks` provides a convenient tool for implementing theorem provers and proof editors of such logics. As an application of this kind, we give in chapter 10 a tableau propositional theorem prover which exploits the utility `flock-compile` and the predicates in `Units`, `Flocks`, and `FlocksIO`.

```
EXPORT     Flocks.

IMPORT     Units.

BASE       Flock.

PREDICATE  EmptyFlock : Flock;
           Extent : Flock * Integer;
           UnitInFlock : Flock * Unit * Integer;
           UnitWithIdentifier : Flock * String * Unit * Integer;
           InsertUnit : Flock * Unit * Integer * Flock;
           DeleteUnit : Flock * Unit * Integer * Flock.
```

Chapter 7

Control

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

7.1 Computation Rule

The computation rule determines which literal or conditional in the current goal will be selected. There are five reasons for employing a flexible computation rule:

- To ensure soundness.
- To assist efficiency.
- To assist termination.
- To solve constraints.
- To control pruning.

We discuss the first four of these in this section. (The use of a flexible computation rule to control pruning will be discussed in the next section.)

We begin by discussing the role of a flexible computation rule in ensuring soundness. The major requirement of any implementation of Gödel is that it be sound. This means that considerable care must be taken when implementing certain facilities, such as negation, conditionals, and intensional set terms, which are well-known to have difficulties in this regard. For example, a standard way of implementing negation (called *safe* negation) is to delay a negative literal until it is ground. Under this restriction, the soundness of the implementation can be proved. In any case, whatever sound implementation of negation is employed, programmers may need to know some details of the implementation since normally there are restrictions on what negative literals can be run. Similar remarks apply to the implementation of conditionals and intensional set terms.

Next we turn to a detailed discussion of the `DELAY` control facility of Gödel. The computation rule is partly under the control of the programmer through explicit `DELAY` declarations. We first motivate the use of DELAY declarations to assist efficiency and termination by various examples. After this motivation, the precise syntax and semantics of DELAY declarations will be presented.

Computation rules which allow coroutining between subgoals can be much more efficient than, for example, the "leftmost literal" computation rule. As an example of this, consider the module Coroutine below.

```
MODULE       Coroutine.

IMPORT       Lists.

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

Slowsort(x,y) <-
             Sorted(y) &
             Permutation(x,y).
```

Coroutine imports, in particular, the predicate Permutation from Lists. Permutation has the following declarations and could be implemented by the following definition.

```
PREDICATE    Permutation : List(a) * List(a).
DELAY        Permutation(x,y) UNTIL NONVAR(x) \/ NONVAR(y).

Permutation([],[]).
Permutation([x|y],[u|v]) <-
             Delete(u,[x|y],z) &
             Permutation(z,v).
```

The DELAY declaration is a *control* declaration. In fact, DELAY declarations are syntactic variants of when declarations, which are due to Naish [20]. The meaning of the DELAY declaration for Permutation is that calls to Permutation will delay until either the first argument is not a variable or the second argument is not a variable.

Coroutine also imports the predicate Sorted from Lists. The declarations and possible definition for Sorted are as follows.

```
PREDICATE    Sorted : List(Integer).
DELAY        Sorted([]) UNTIL TRUE;
             Sorted([_]) UNTIL TRUE;
             Sorted([x,y|_]) UNTIL NONVAR(x) & NONVAR(y).

Sorted([]).
Sorted([_]).
Sorted([x,y|z]) <-
             x =< y &
             Sorted([y|z]).
```

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,y|_]`, then the call delays. Thus the call `Sorted(x)` delays. If the argument of a call to `Sorted` has the form `[s,t|r]`, where either `s` or `t` is a variable, then the call delays. Thus the call `Sorted([1,x|y])` delays. If the argument of a call to `Sorted` is either `[]` or has the form `[r]` or `[s,t|r]`, where `s` and `t` are not variables, then the call can proceed. Thus the calls `Sorted([])` and `Sorted([3,2|x])` can proceed.

Now, assuming the "leftmost literal" computation rule was employed, the statement for `Slowsort` would have to be written as follows

```
Slowsort(x,y) <-
             Permutation(x,y) &
             Sorted(y).
```

so that the call to `Permutation` would be run before the call to `Sorted`. Thus a goal such as

```
<- Slowsort([6,1,5,2,4,3],x).
```

would take time having order $n!$, where n is the length of the input list. However, if the above `DELAY` declaration on `Sorted` is employed, then the time taken for such goals for the module `Coroutine` can be considerably reduced. Note that, using the `DELAY` declaration for `Sorted`, the call to `Sorted` may have to be put before the call to `Permutation` in the statement for `Slowsort` to ensure that the desired coroutining behaviour is achieved. (This depends on the implementation.) First, `Sorted` delays, then `Permutation` computes the first couple of elements of the permuted list (assuming it has length greater than one), then `Sorted` checks if this much of the permuted list is sorted, and so on.

Next we discuss the use of `DELAY` declarations to assist termination. Consider the module `M5` again. `M5` imports the predicate `Append` from `Lists`, where `Append` has the following `DELAY` declaration

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

Without this `DELAY` declaration for `Append`, the computation (using the "leftmost literal" computation rule) of a goal such as

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

gives all possible ways of splitting `[1,2,3]`, but then goes into an infinite loop. However, with the above `DELAY` declaration on `Append` instead, the computation of the goal gives all possible ways of splitting the list and then terminates.

As another example, consider the predicate `Delete`, which the definition of `Permutation` uses and which has the following declarations and possible definition.

```
PREDICATE    Delete : a * List(a) * List(a).
DELAY        Delete(_,y,z) UNTIL NONVAR(y) \/ NONVAR(z).

Delete(x,[x|y],y).
Delete(x,[y|z],[y|w]) <-
              Delete(x,z,w).
```

Now, using the "leftmost literal" computation rule, the computation of a goal such as

```
<- Permutation(x,[1,2,3]).
```

will first return the answer `x = [1,2,3]` and then, upon backtracking, will go into an infinite loop. However, using the above `DELAY` declarations for `Permutation` and `Delete` instead, the system will return all 6 permutations of `[1,2,3]` and then fail.

As well as the condition `NONVAR`, a `DELAY` declaration can use the condition `GROUND`. For example. the predicate `<` in the module `Floats` has the following `DELAY` declaration.

```
DELAY        x < y UNTIL GROUND(x) & GROUND(y).
```

This declaration delays calls to `<` until both its arguments are ground.

The fourth reason for employing a flexible computation rule is to solve constraints. In general, a goal consists of a constraint and a non-constraint part, and coroutining between the two parts is usually necessary for efficiency. Furthermore, just the solving of a system of constraints alone may require coroutining. The control necessary for the efficient solving of constraints in the modules `Integers` and `Rationals` is exercised by the system itself and cannot be changed by the programmer.

We now turn to the definition of 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 given by the following grammar.

Cond ⟶ *Cond1*
| *Cond1* '`&`' *AndSeq*
| *Cond1* '`\/`' *OrSeq*

Cond1 ⟶ '`NONVAR(`' *var* '`)`'
| '`GROUND(`' *var* '`)`'
| '`TRUE`'
| '`(`' *Cond* '`)`'

AndSeq ⟶ *Cond1*
| *Cond1* '`&`' *AndSeq*

OrSeq ⟶ *Cond1*
| *Cond1* '`\/`' *OrSeq*

where *var* is a variable in *Atom.*

A `DELAY` declaration for a predicate can only appear in the module where the predicate is declared. Also, as is illustrated by the `DELAY` declaration for `Sorted`, a set of `DELAY` declarations can be compacted into a single one of the form

`DELAY` *Atom_1* `UNTIL` *Cond_1*;
⋮
Atom_n `UNTIL` *Cond_n*.

The following condition is placed on *Atom.*

- No pair of *Atom*s in the set of DELAY declarations for a predicate can have a common instance.

This condition simplifies the semantics of DELAY declarations. Without it, a situation could arise where one declaration allowed a call to proceed, while another delayed it.

In the grammar for *Cond*, the reserved word & stands for conjunction, \/ 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. Now suppose an atom A is an instance by a substitution θ of an *Atom* (that is, $A = Atom\theta$) in a DELAY declaration. Then we say 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, we say A does *not satisfy* the corresponding condition. For example, the atom Permutation(x,[1|y]) is an instance of the atom in the DELAY declaration for Permutation and satisfies the corresponding condition, Sorted([1,y|z]) and Sorted([1,3|y]) are both instances of the third DELAY declaration for Sorted but only Sorted([1,3|z]) satisfies the corresponding condition.

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

- 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[1] 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.

As another example to illustrate the control facilities, we consider the predicate Merge in the module Lists. Merge has the following declarations and possible definition.

```
PREDICATE  Merge : List(Integer) * List(Integer) * List(Integer).
DELAY      Merge(x,y,z) UNTIL (NONVAR(x) & NONVAR(y)) \/ NONVAR(z).

Merge([],x,x).
Merge(x,[],x).
Merge([u|x],[v|y],[u|z]) <-
           u < v &
           Merge(x,[v|y],z).
Merge([u|x],[v|y],[v|z]) <-
           u >= v &
           Merge([u|x],y,z).
```

The DELAY declaration for Merge shows that it can be used to either merge two given lists or else split a given list. Thus the goal

```
<- Merge([1,4,8],[3,6,9],x)
```

has the answer x = [1,3,4,6,8,9], the goal

[1] There can be at most one such declaration.

```
<- Merge(x,y,[1,4])
```

has the answers

```
x = [], y = [1,4]
x = [1,4], y = []
x = [1], y = [4]
x = [4], y = [1]
```

and the goal

```
<- Merge(x,[1,4],z)
```

flounders.

Now let us return to the `DELAY` declaration for `Sorted` again. It is clear from the above definitions that the first and second parts of this declaration can be omitted and the effect will be exactly the same. Thus the original `DELAY` declaration for `Sorted` could be replaced by

```
DELAY         Sorted([x,y|_]) UNTIL NONVAR(x) & NONVAR(y).
```

The reason is that all calls to `Sorted` have the form either `Sorted(x)`, or `Sorted([])`, or `Sorted([s])`, or `Sorted([s,t|r])`. Thus there is no call to `Sorted` which would be delayed by either of the first two parts of the declaration and which would not be delayed by the third part. However, we prefer to keep the original form of the declaration for clarity and documentation purposes.

Note that it is not always true that `DELAY` declarations with the condition `TRUE` have no effect and hence can be omitted. For example, if a predicate has a single `DELAY` declaration with the condition `TRUE`, then all calls which have a common instance with the atom of the declaration, but are not an instance of the atom, are delayed.

It is also worth remarking that other `DELAY` declarations for `Sorted` are possible. The weakest declaration (that is, the one that delays the least) is

```
DELAY         Sorted(x) UNTIL NONVAR(x).
```

This declaration will allow some calls to `Sorted` to proceed and then delay at the subsequent call to `=<`, whereas the original declaration will guarantee that once `Sorted` can proceed, the call to `=<` will also be able to proceed immediately. `DELAY` declarations intermediate between these two extremes are also possible and, in fact, there is little to choose between them. In general, when there is a choice, we have opted for the strongest declaration applicable.

Finally, in the module `EightQueens` below, we give the Gödel version of an 8-queens program due to Naish [20].

7.2 Pruning

The most general form of the Gödel pruning operator, called *commit*, has the form `{...}_`n, of which two special cases have the form `|` and `{...}`. We begin by explaining with examples the more familiar `|` commit, then we discuss the `{...}` commit, and finally give the most general form of the commit.

```
MODULE      EightQueens.

IMPORT      Lists.


PREDICATE   Queen : List(Integer).

Queen(x) <-
            Safe(x) &
            Permutation([1,2,3,4,5,6,7,8], x).

PREDICATE   Safe : List(Integer).
DELAY       Safe(x) UNTIL NONVAR(x).

Safe([]).
Safe([x|y]) <-
            NoDiagonal(x,1,y) &
            Safe(y).

PREDICATE   NoDiagonal : Integer * Integer * List(Integer).
DELAY       NoDiagonal(_,_,z) UNTIL NONVAR(z).

NoDiagonal(_,_,[]).
NoDiagonal(x,y,[z|w]) <-
            y ~= Abs(z - x) &
            NoDiagonal(x,y+1,w).
```

However, before going on to discuss the details of the pruning operator, we make the following important remark: *well-written Gödel programs generally have very little need for pruning.* For example, apart from the programs in this chapter which are deliberately designed to illustrate various aspects of pruning, there are remarkably few occurrences of the commit operator in programs in this book. Programmers are strongly encouraged to avoid the use of (explicit) pruning wherever possible, as such avoidance generally results in programs which have clearer semantics and are more amenable to analysis and transformation. In particular, the "non-failing" programming style of the propositional theorem prover in section 10.3 is worth studying as, when applicable, it often results in a program with clearer declarative semantics and no need for (explicit) pruning at all.

Now consider the module `P1`, which is a variation of the module `Qsort` in chapter 6 and for which the predicate `Quicksort` is intended to be called with its first argument instantiated. Module `P1` uses the `|` version of the commit, which we call the *bar* commit. Declaratively, `|` is just conjunction. However, for convenience, if either argument of `|` is `True` it can be omitted, as in the first statement for `Quicksort3`. Each statement can contain at most one `|`. The *scope* of `|` is the formula to its left in the body of the statement. The order in which the statements are tried is not specified, so that commit does not have the sequentiality property of cut. The procedural meaning of `|` is that only one solution is found for the formula in its scope and all other branches arising from the other statements in the definition which contain a `|` are pruned. Thus the meaning of `|` is close to the commit of the concurrent logic programming languages. Note that, while a `|` commit would normally appear in *every* statement of a definition for which at least one `|` appears, this is not obligatory.

The `DELAY` declarations in module `P1` enforce the intended use of `Quicksort`. Thus there are `DELAY` declarations for `Quicksort` and `Quicksort3` which delay calls to these predicates until their first argument is not a variable. The `DELAY` declaration for `Partition` ensures that either the first argument of a call to `Partition` will have its first argument the empty list, or the first element of the list in the first argument and the second argument will be known so that an arithmetic comparison between these can be carried out. In fact, the following weaker `DELAY` declaration for `Partition` is also suitable.

```
DELAY        Partition(x,_,_,_) UNTIL NONVAR(x).
```

If the first argument of a call to `Partition` is not a variable, the system can correctly commit to either the first statement in case this argument is the empty list or otherwise to one of the second and third statements. The commitment to either the second or third statement will be delayed until enough information is known to run the arithmetic tests.

For the intended use of `Quicksort`, the statements in the definition for `Quicksort3` are mutually exclusive, in the sense that the system can commit to at most one of these. The same is true for `Partition`. Thus no answer will be pruned – only useless computation will be pruned. So, in this case, the commits enhance efficiency without affecting completeness. More generally, the commits can prune answers.

Next we illustrate the Gödel one-solution commit, written `{...}`. Consider the module `P2`. The intended meaning of `Perm` in `P2` is that it should be true when its first argument is a list of integers and its second argument is a permutation of the list in the first argument.

A goal such as

```
<- Perm([1,2,3],x).
```

will produce (in some order) the sequence of computed answers

```
x = [1,2,3]
x = [1,3,2]
...
x = [3,2,1]
```

If instead the goal was

```
<- {Perm([1,2,3],x)}.
```

then only one answer, `x = [1,3,2]`, say, would be produced. 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. Of course, the one-solution commit can also be used in the bodies of statements in a program.

```
MODULE      P1.

IMPORT      Lists.


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

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


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

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


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

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

```
MODULE      P2.

IMPORT      Lists.


PREDICATE   Perm : List(Integer) * List(Integer).
DELAY       Perm(x,y) UNTIL NONVAR(x) \/ NONVAR(y).

Perm([],[]).
Perm([x|y],[u|v]) <-
            Del(u,[x|y],z) &
            Perm(z,v).


PREDICATE   Del : Integer * List(Integer) * List(Integer).
DELAY       Del(_,y,z) UNTIL NONVAR(y) \/ NONVAR(z).

Del(x,[x|y],y).
Del(x,[y|z],[y|w]) <-
            Del(x,z,w).
```

The bar commit and the one-solution commit provide programmers with powerful pruning facilities, which should suffice for the vast majority of programming tasks. However, source-level tools, such as program transformers and partial evaluators, have need of a more powerful pruning operator. The reason is that the class of programs containing `|` is not closed under even simple program transformations, such as unfolding. For this reason, Gödel has a more powerful pruning operator, `{...}_`n, which includes the `|` and `{...}` as special cases, and which gives a class of programs closed under the usual program transformations. This pruning operator was introduced in [12].

To motivate the pruning operator, consider module `P3`, which is a variant of module `P2`. Module `P3` can be used to *check* that one list is a permutation of another list. For example, the goal

```
<- Perm([1,2,3,4],[4,1,2,3]).
```

will produce the answer `Yes`.

The `DELAY` declaration for `Perm` will delay a call to it until its first argument is not a variable. This is sufficient to ensure commitment to the desired statement. In fact, the following weaker declaration also ensures the desired commitment.

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

Similarly, the `DELAY` declaration for `Del` delays calls to it until the desired commitment is ensured.

```
MODULE      P3.

IMPORT      Lists.


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

Perm([],[]) <-
            |.
Perm([x|y],[u|v]) <-
            |
            Del(u,[x|y],z) &
            Perm(z,v).


PREDICATE   Del : Integer * List(Integer) * List(Integer).
DELAY       Del(x,[u|_],_) UNTIL NONVAR(x) & NONVAR(u).

Del(x,[x|y],y) <-
            |.
Del(x,[y|z],[y|w]) <-
            x ~= y |
            Del(x,z,w).
```

An important heuristic for finding good `DELAY` declarations to control pruning is that they should be strong enough to avoid unexpected failure because of premature commitment to an inappropriate statement. For example, suppose the `DELAY` declaration for `Del` was omitted. Now consider the goal

```
<- Del(x,[1,2,3],y) & x = 2.
```

The first atom in this goal unifies with the head of the first statement and so the system could commit to this statement. However, subsequently the call `1 = 2` will cause the goal to fail unexpectedly. If instead the call to `Del` had been delayed according to the `DELAY` declaration for it in `P3`, then the goal would have succeeded, as expected.

Now consider the program transformation task of unfolding (once) the call to `Del` in the second statement for `Perm` in module `P3`. For this purpose, we first rewrite the module using the more general commit. This is module `P4`. The scope of `|` is the formula in the body to its left. So each `|` is rewritten using `{` and `}` to indicate its scope inside a statement explicitly and a label, which is a positive integer, to indicate the scope across the statements in each definition. The actual values of the labels in the two definitions are unimportant. It is only important that the two commits in the definition of `Perm` should have the same label and that the two commits in the definition of Del should have the same label. In module P4, the labels have been standardized apart in preparation for the unfolding step.

```
MODULE       P4.

IMPORT       Lists.


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

Perm([],[]) <-
             {True}_1.
Perm([x|y],[u|v]) <-
             {True}_1 &
             Del(u,[x|y],z) &
             Perm(z,v).


PREDICATE    Del : Integer * List(Integer) * List(Integer).
DELAY        Del(x,[u|_],_) UNTIL NONVAR(x) & NONVAR(u).

Del(x,[x|y],y) <-
             {True}_2.
Del(x,[y|z],[y|w]) <-
             {x ~= y}_2 &
             Del(x,z,w).
```

Now the unfolding step at the call to Del in the second statement for Perm can be performed. The result of this is given in module P5. The brackets {...} of a commit {*Formula*}_*n* indicate the scope of the commit inside a statement. The label *n* indicates the scope of the commit over the statements in a definition. When the formula *Formula* succeeds, all other statements in the definition which contain a commit labelled *n* are pruned. The other pruning which takes place is that only one solution is found for *Formula*. The commits which have been introduced into module P5 by the unfolding step are such that an answer computed for a goal to module P5 can also be computed for the same goal to module P4. Thus the transformation is sound for the procedural semantics. This follows from a theorem about partial evaluation in [12]. (See theorem 3.1.)

Note that the DELAY declaration for Perm in module P5 has been strengthened compared to the one in module P4. This is to ensure correct commitment. For example, if the DELAY declaration for Perm in P5 is replaced by the weaker one for Perm in P4, then the goal

```
<- Perm([1,2,3],x) & x = [2,1,3].
```

```
MODULE       P5.

IMPORT       Lists.


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

Perm([],[]) <-
             {True}_1.
Perm([x|y],[x|v]) <-
             {True}_1 &
             {True}_2 &
             Perm(y,v).
Perm([x|y],[u|v]) <-
             {True}_1 &
             {x ~= u}_2 &
             Del(u,y,w) &
             Perm([x|w],v).


PREDICATE    Del : Integer * List(Integer) * List(Integer).
DELAY        Del(x,[u|_],_) UNTIL NONVAR(x) & NONVAR(u).

Del(x,[x|y],y) <-
             {True}_2.
Del(x,[y|z],[y|w]) <-
             {x ~= y}_2 &
             Del(x,z,w).
```

could fail unexpectedly by committing to the second statement.

The unfolding in module `P4` was carried out at an atom which was not inside the scope of the commit in the statement. For the case when the unfolding is carried out at an atom in the scope of all the commits in a statement (that is, the regularity condition in [12] is satisfied), a stronger soundness result applies. This is that, for any *set* of answers computed by the program resulting from the unfolding, the same set can also be computed by the original program. (See theorem 3.2 in [12].)

The one-solution commit `{...}` is regarded as a special case of the commit `{...}_`n in that `{...}` is assumed to have a label that does not occur elsewhere in the definition in which it appears. Another aspect of the commit worth noting is that an implementation employing negation as failure (and variations of it) must disable all pruning inside a negated call. This ensures the soundness of the implementation. (See [12].)

We now describe the (top-level) syntax for statements and goals containing commits. For this, we need to give the (top-level) syntax for a body, which may be either the body of a goal or a statement. The grammar for a (non-empty) body is as follows.

```
Body           ⟶   [CFormula(f)] <‘|’> [CFormula(f1)]
                  | CFormula(f)

CFormula(0)    ⟶   <‘(’> CFormula(f) <‘)’>
                  | <‘{’> CFormula(f) <‘}’ [‘_’ Label]>
CFormula(2)    ⟶   CFormula(f) <‘&’> CFormula(f1)
        Condition: f ≤ 1, f1 ≤ 2.
CFormula(f)    ⟶   Formula(f)
```

where the notation [...] indicates 0 or 1 occurrences of the included item, Label is a positive integer, and Formula is a formula not involving commits.

Next, we describe how the bar commits and the one-solution commits are preprocessed away. The preprocessed versions of bodies will be useful for the ground representation. According to the above grammar, only one bar commit is allowed in a body and it must not appear inside the scope of any other commit. So a body of the form V | W (resp., V |, | W, |) is replaced by {V}_n & W (resp., {V}_n, {*True*}_n & W, {*True*}_n), for a suitable positive integer n. If the bar commit is in the body of a goal, then n must be different from any other label in the goal. For bar commits appearing in a definition, n must be the same for all bar commits in the definition and different from the label of any other commits in the definition. Finally, any one-solution commits are given a label unique to the goal or definition in which they appear. After this preprocessing, all bodies contain only commits of the form {...}_n.

Finally, we give a somewhat more formal description of the procedural semantics of commit. (More details for the case of SLDNF-resolution can be found in [12].) First we need to introduce the concept of a search tree corresponding to some proof procedure. We avoid a formal definition here, but the idea is intuitively clear. Nodes in a search tree are labelled by goals, the root node being labelled by the top-level goal. Each node in a search tree has zero or more children, where each child corresponds to a goal obtained from its parent by a (positive or negative) derivation step according to the proof procedure. The definition of a search tree is deliberately left vague since we want to encompass the concept of a search tree for as many proof procedures as possible. We define an l-child as follows.

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

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

2. G_1 is derived from G_0 using an input statement which contains a commit labelled l (after standardisation apart of the commit labels).

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

Note the following properties of l-children.

1. A node can be an l-child of the first or second kind, but not both.
2. The l-children of a node are either all of the first kind or all of the second kind.
3. If one child of a node is an l-child of the first kind, then all children of the node are l-children.
4. A node can be an l-child and an m-child, where $l \neq m$.

The ideas of the previous definition are illustrated by module `P6`, for which the following is a search tree (using SLDNF-resolution, say).

$$\begin{tikzcd}
 & & \texttt{<- P} \arrow[dll] \arrow[d] \arrow[drr] & & \\
\texttt{<- \{Q\}\_1 \& R} \arrow[d] & & \texttt{<- \{S\}\_1 \& T} \arrow[dl] \arrow[dr] & & \texttt{<- U \& \{V\}\_2} \arrow[d] \\
\texttt{<- R} \arrow[d] & \texttt{<- \{B\}\_1 \& T} & & \texttt{<- T} & \texttt{<- W \& Z \& \{V\}\_2} \\
\square & & & &
\end{tikzcd}$$

Then `<- {Q}_1 & R` and `<- {S}_1 & T` are 1-children and `<- U & {V}_2` is a 2-child of `<- P` of the second kind. The nodes `<- {B}_1 & T` and `<- T` are 1-children of `<- {S}_1 & T` of the first kind. The node `<- W & Z & {V}_2` is not a 2-child of `<- U & {V}_2`, as the selected literal is not in the scope of the commit.

Now we can define the concept of a pruning step, which gives the procedural meaning of the Gödel commit.

Definition Let S be a subtree of a search tree. The tree S' is obtained from S by a *pruning step* in S at G_0 if the following conditions are satisfied.

1. S has a node G_0 with distinct l-children G_1 and G_2, and there is an l-free node G_2' in S which is either equal to or below G_2.
2. S' is obtained from S by removing the subtree of S rooted at G_1.

G_1 is the *cut node* and the pair (G_2, G_2') is an *explanation* for the pruning step.

Consider the preceding search tree. We can apply a pruning step to this tree to obtain the subtree S_1

```
MODULE        P6.

PROPOSITION   P,Q,R,S,U,B,W,Z,T,V.

P <- {Q}_1 & R.
P <- {S}_1 & T.
P <- U & {V}_2.
Q.
S <- B.
S.
U <- W & Z.
T.
R.
```

$$\begin{tikzcd}
 & & \texttt{<- P} \arrow[dl] \arrow[dr] & \\
 & \texttt{<- \{S\}\_1 \& T} \arrow[dl] \arrow[dr] & & \texttt{<- U \& \{V\}\_2} \arrow[d] \\
\texttt{<- \{B\}\_1 \& T} & & \texttt{<- T} & \texttt{<- W \& Z \& \{V\}\_2}
\end{tikzcd}$$

where the explanation for pruning the leftmost branch is the pair (`<- {S}_1 & T`, `<- T`). We can apply another pruning step to S_1 to obtain the subtree S_2

$$\begin{tikzcd}
 & \texttt{<- P} \arrow[dl] \arrow[dr] & \\
\texttt{<- \{S\}\_1 \& T} \arrow[d] & & \texttt{<- U \& \{V\}\_2} \arrow[d] \\
\texttt{<- T} & & \texttt{<- W \& Z \& \{V\}\_2}
\end{tikzcd}$$

where the explanation for pruning the leftmost child of `<- {S}_1 & T` is the pair (`<- T`, `<- T`). No further pruning is possible.

We have defined a pruning step to be the smallest amount of pruning that could take place at any time. However, an implementation of Gödel may apply several pruning steps together. For example, if a pruning step at a node G_0 with cut node G_1 is possible because there is an explanation (G_2, G_2'), where G_1 and G_2 are l-children of G_0 and G_2' is l-free, then it would also be possible to prune all other subtrees rooted at l-children of G_0 using the same explanation (G_2, G_2'). In the case when all children of G_0 are l-children, this amounts to removing the choice point at G_0. It is also possible to remove the choice points at each node on the branch to G_2' at which the selected literal is in the scope of an l-commit. In fact, an implementation may do pruning *eagerly*, in the sense that when the scope of a commit is solved the system would perform as much pruning as is possible as a result of solving that scope.

Chapter 8

Input/Output

The Gödel input/output facilities are provided by the system modules `IO`, `NumbersIO`, `FlocksIO`, `ProgramsIO`, `ScriptsIO`, and `TheoriesIO`. The first three modules are discussed in this chapter. The last three are relevant for meta-programming applications and so the discussion of these is postponed to chapter 9.

8.1 `IO`

The basic input/output module is `IO`, the export part (except for the `DELAY` declarations) of which is given below. Two types, `InputStream` and `OutputStream`, are provided which are the types of input streams and output streams, respectively. To open a file for input, the predicate `FindInput` is used. Its first argument is the name of a file (given as a string) and the second argument contains either a term of the form `In(`*stream*`)`, where *stream* is the new input stream, or the constant `NotFound` depending on whether the attempt to open the file was successful or not. Similarly, `FindOutput` is used to open a file for output. The read predicates are `Get`, which returns the ASCII code of the next character read from the open input stream, and `ReadChar`, which returns a string of length 1 containing the next character read from the open input stream, assuming the end of the stream has not yet been reached. The write predicates are `Put`, which writes the character whose ASCII code is the second argument to the open output stream, and `WriteString`, which writes the string of characters in the second argument to the open output stream. The predicate `EndInput` closes an open input stream and `EndOutput` closes an open output stream. As an example of the use of the module `IO`, the module `DisplayFile` below reads a file as input and displays it on standard output.

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

```
EXPORT          IO.

IMPORT          Strings.

BASE            InputStream, OutputStream, ResultOfFind.

CONSTANT        StdIn : InputStream;
                StdOut, StdErr : OutputStream;
                NotFound : ResultOfFind.

FUNCTION        In : InputStream -> ResultOfFind;
                Out : OutputStream -> ResultOfFind.

PREDICATE       FindInput : String * ResultOfFind;
                FindOutput : String * ResultOfFind;
                FindUpdate : String * ResultOfFind;
                EndInput : InputStream;
                EndOutput : OutputStream;
                Get : InputStream * Integer;
                ReadChar : InputStream * String;
                Put : OutputStream * Integer;
                WriteString : OutputStream * String;
                NewLine : OutputStream;
                Flush : OutputStream.
```

```
EXPORT      DisplayFile.

IMPORT      Strings.

PREDICATE   Display : String.

DELAY       Display(x) UNTIL GROUND(x).

% Display displays on standard output the file whose name appears in its
% argument.
```

```
LOCAL       DisplayFile.

IMPORT      IO.

% PREDICATE   Display : String.

Display(file) <-
            FindInput(file, result) &
            ProcessStream(result).


PREDICATE   ProcessStream : ResultOfFind.

ProcessStream(In(stream)) <-
            Get(stream, c) &
            DisplayStream(stream, c) &
            EndInput(stream).

ProcessStream(NotFound) <-
            WriteString(StdOut, "File not found") &
            NewLine(StdOut).


PREDICATE   DisplayStream : InputStream * Integer.

DisplayStream(stream, c) <-
            IF c ~= -1
            THEN
              Put(StdOut, c) &
              Get(stream, c1) &
              DisplayStream(stream, c1).
```

to adopt this kind of module structure, since it is only the modules that are not input/output ones to which the purest forms of program transformation, declarative debugging, and so on, apply. Since the order of input/output calls is important, an implementation should indicate in what order such calls will be made.

8.2 NumbersIO

The system module `NumbersIO` provides input/output facilities for integers, rationals, and floating-point numbers. It also provides conversion between a number and its representation as a string which is often useful for input/output. The export part (except for the `DELAY` declarations) of

NumbersIO is given below.

The predicate ReadInteger skips over layout characters to find the next token which should be an integer. If the next token is not an integer, ReadInteger fails. If end of file has already been reached (or if there are only layout characters before the end of file), ReadInteger succeeds with 0 in the second argument and the constant EOF in the third. If end of file has not been reached and ReadInteger succeeds, then NotEOF is returned in the third argument. The predicates ReadRational and ReadFloat behave similarly.

The predicates WriteInteger, WriteRational, and WriteFloat write numbers of the appropriate type to an output stream. The predicates IntegerString, RationalString, and FloatString provide conversion between numbers and their string representations. These predicates are useful, for example, for user-defined procedures for reading numbers from a file.

```
EXPORT       NumbersIO.

IMPORT       IO, Numbers.

BASE         FileInfo.

CONSTANT     EOF, NotEOF : FileInfo.

PREDICATE    ReadInteger : InputStream * Integer * FileInfo;
             ReadRational : InputStream * Rational * FileInfo;
             ReadFloat : InputStream * Float * FileInfo;
             WriteInteger : OutputStream * Integer;
             WriteRational : OutputStream * Rational;
             WriteFloat : OutputStream * Float;
             IntegerString : Integer * String;
             RationalString : Rational * String;
             FloatString : Float * String.
```

8.3 FlocksIO

So that a program can access a term representing a flock, Gödel provides a system module, called FlocksIO. The export part of FlocksIO is given below. FlocksIO imports IO so that the predicates there can be used to open and close files. The predicate GetFlock is intended to be called with its first argument ground and instantiates the second argument to the term representing the flock in the file with extension .flk corresponding to the input stream in the first argument. The predicate PutFlock is intended to be called with both arguments ground. It puts the term in the second argument into the file with extension .flk corresponding to the output stream in the first argument. The output stream must have been found using FindOutput, so that the term being written overwrites any existing term in this file. Note that FlocksIO is an input/output module so it should be handled like the module `IO`. In particular, it is preferable to arrange the module structure of the program so that only its main module depends upon `FlocksIO`. A program illustrating the use of `FlocksIO` is given in section 10.3.

```
EXPORT       FlocksIO.

IMPORT       IO, Flocks.

PREDICATE    GetFlock : InputStream * Flock;
             PutFlock : OutputStream * Flock.
```

Chapter 9

Meta-Programming

In the design of Gödel, particular attention has been paid to its meta-logical facilities. As we have already indicated, using the ground representation, we can write in a declarative way many important kinds of meta-programs, such as program transformers, compilers, debuggers, abstract interpreters, program synthesizers, and theorem provers. We now turn attention to the the various modules which provide meta-programming facilities via the ground representation.

9.1 `Syntax`

The ground representation is a scheme for representing object programs, modules, goals, theories, declarations, terms, and so on, as terms in a meta-language. The details of the ground representation are not made explicit. (For background on the ground representation, see [3], [11], or [10], and for a detailed discussion of a particular implementation of the ground representation for Gödel itself, see [4].) Instead, following an abstract data type approach, Gödel provides, via the system modules `Syntax`, `Programs`, `Scripts`, and `Theories`, a set of predicates which allow a meta-program to access and manipulate terms representing object expressions. The constants and functions actually used in the representation are hidden in the local part of these modules. However, we emphasize strongly that, even though the details of the ground representation are hidden, all the predicates exported by these modules have declarative definitions.

Gödel supports the ground representation of (object) programs, scripts, and theories. The first of these is given by the module `Programs`, the second by the module `Scripts`, and the third by the module `Theories`. We discuss these three modules in subsequent sections. In this section, we discuss the module `Syntax` which provides some facilities for the manipulation of the representations of (object) expressions common to all three of these ground representations.

First, `Syntax` imports all the symbols exported by the modules `Integers`, `Lists`, and `Strings`. It also declares (amongst others) the following bases, which are required by the ground representation of expressions of various kinds.

```
BASE        Name, Type, Term, Formula, TypeSubst, TermSubst, VarTyping.
```

`Name` is the type of a term representing the name of a symbol, `Type` is the type of a term representing a type, `Term` is the type of a term representing a term, `Formula` is the type of a term representing a formula, `TypeSubst` is the type of a term representing a type substitution, `TermSubst` is the type of a term representing a term substitution, and `VarTyping` is the type of a term representing a variable typing.

Note that two kinds of substitutions need to be represented. The first is the usual *term substitution* consisting of a set of term bindings, which are pairs of variables and terms. The other kind is a *type substitution* consisting of a set of type bindings, which are pairs of parameters and types. All the usual operations on term substitutions are also provided for type substitutions. A *variable typing* is a set of bindings where each binding consists of a variable together with the type assigned to that variable.

We now discuss in some detail a representative selection of predicates provided by `Syntax`. The complete export part of `Syntax` is given in chapter 13.

The first collection of predicates is mainly concerned with the representation of the connectives and quantifiers.

```
PREDICATE  And : Formula * Formula * Formula;
           All : List(Term) * Formula * Formula;
           Commit : Integer * Formula * Formula.
```

`And` is intended to be true when its first and second arguments are the representations of formulas, and its third argument is the representation of the formula which is the conjunction of the formulas in the first and second arguments. `All` is intended to be true when its first argument is a list of the representations of variables, its second argument is the representation of a formula, and its third argument is the representation of the formula which is obtained by taking the universal quantification over the set of variables in the first argument of the formula in the second argument. `Commit` is intended to be true when its first argument is a commit label, its second argument is the representation of a formula, and its third argument is the representation of the formula obtained by enclosing the formula in the second argument using commits with this label.

The next four predicates are concerned with the representation of variables and parameters.

```
PREDICATE  Variable : Term;
           VariableName : Term * String * Integer;
           Parameter : Type;
           ParameterName : Type * String * Integer.
```

`Variable` is intended to be true when its argument is the representation of a variable. `Variable` can only be used to *check* whether or not its argument is the representation of a variable. Variable names have the form `name_n`, where `name` is the *root* of the name of the variable and the non-negative integer `n` is called the *index* of the variable.[1] `VariableName` is intended to be true when its first argument is the representation of a variable, its second argument is the root of the name of the variable, and its third argument is the index of the variable. `VariableName` is used to create the representations of new variables and also to split a variable name into its root and index. Parameter names are analogous to variable names and have a root and index. `Parameter` is intended to be true when its argument is the representation of a parameter. `ParameterName`

[1]As an implementation point, we note that variable names of the form `name_n` allow a convenient and efficient way of creating new variables, renaming variables, and so on, both for the user and the system. Furthermore, an implementation must give an index to *all* object variables, even those not created by `VariableName`. This greatly increases the usefulness of the predicates `FormulaMaxVarIndex` and `TermMaxVarIndex` (see chapter 13), which are often needed for renaming.

is intended to be true when its first argument is the representation of a parameter, its second argument is the root of the name of the parameter, and its third argument is the index of the parameter.

The next collection of predicates is mainly concerned with ensuring types, terms, and formulas have certain properties.

```
PREDICATE  NonVarTerm : Term;
           Atom : Formula;
           Statement : Formula;
           EmptyFormula : Formula.
```

`NonVarTerm` is intended to be true when its argument is the representation of a non-variable term. `Atom` is intended to be true when its argument is the representation of an atom. `Statement` is intended to be true when its argument is the representation of a statement. `EmptyFormula` is intended to be true when its argument is the representation of the empty formula.

The next two predicates are used to construct and pull apart the representations of terms and atoms.

```
PREDICATE  FunctionTerm : Term * Name * List(Term);
           PredicateAtom : Formula * Name * List(Term).
```

`FunctionTerm` is intended to be true when its first argument is the representation of a non-opaque term with a function at the top-level, its second argument is the representation of the name of this function, and its third argument is the list of representations of the top-level subterms of this term. 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 declared in the local part of a closed module; otherwise, it is *non-opaque*. A predicate `OpaqueTerm` is provided which can be used to find out whether a term is opaque or not. Opaque types and atoms are defined analogously. `PredicateAtom` is intended to be true when its first argument is the representation of a non-opaque atom with a predicate at the top-level, its second argument is the representation of the name of this predicate, and its third argument is the list of representations of the top-level terms of this atom.

The final collection of predicates is useful for a variety of standard meta-programming tasks.

```
PREDICATE  RestrictSubstToTerm : Term * TermSubst * TermSubst;
           BindingToTermSubst : Term * Term * TermSubst;
           BindingInTypeSubst : TypeSubst * Type * Type;
           UnifyTerms : Term * Term * TermSubst * TermSubst;
           RenameFormulas : List(Formula) * List(Formula) * List(Formula);
           VariantFormulas : List(Formula) * List(Formula);
           Derive : Formula * Formula * Formula * Formula * Formula *
                    TermSubst * Formula.
```

`RestrictSubstToTerm` is intended to be true when its first argument is the representation of a term, its second argument is the representation of a term substitution, and its third argument is the representation of this term substitution restricted to the variables in this term. `BindingToTermSubst` is intended to be true when its first argument is the representation of a variable, its second argument is the representation of a term, and its third argument is the representation of the term substitution containing just the binding in which this variable is bound to this term. `BindingInTypeSubst` is intended to be true when its first argument is the representation of a type substitution, its second argument is the representation of a parameter in a binding in this substitution, and its third argument is the representation of the type to which this parameter is bound in this binding. `UnifyTerms` is intended to be true when its first and second arguments are the representations of terms, its third argument is the representation of a term substitution, and its fourth argument is the representation of the term substitution obtained by composing the term substitution in the third argument with a specific, unique mgu for the terms which are obtained by applying the term substitution in the third argument to the terms in the first two arguments. `RenameFormulas` is intended to be true when its first and second arguments are lists of representations of formulas and its third argument is the list of representations of the formulas obtained by a specific, unique renaming of the free variables of the formulas in the second argument with the property that they become distinct from the free variables in the formulas in the first argument. `VariantFormulas` is intended to be true when its first argument is a list of representations of formulas and its second argument is a list of representations of formulas which are variants of the formulas in the first argument. `Derive` is intended to be true when its first argument is the representation of the head of a normal resultant [10], its second argument is the representation of the body to the left of selected atom in the resultant, its third argument is the representation of the selected atom in the resultant, its fourth argument is the representation of the body to the right of selected atom in the resultant, its fifth argument is the representation of a normal statement whose head unifies with the selected atom in the resultant, its sixth argument is the representation of a specific, unique mgu of the head of the selected atom and the head of the statement, and its seventh argument is the representation of the derived resultant.

As an illustration of the use of the above meta-level predicates, we give below a definition of the predicate `MyUnifyTerms`, which is essentially a special case of the system predicate `UnifyTerms`, using other predicates provided by `Syntax`.

```
EXPORT     Unify.

IMPORT     Syntax.


PREDICATE  MyUnifyTerms :

  Term          % Representation of a term.
* Term          % Representation of a term.
* TermSubst.    % Representation of an mgu for these terms.
```

```
LOCAL     Unify.


% PREDICATE  MyUnifyTerms : Term * Term * TermSubst.

MyUnifyTerms(t,t1,s) <-
           EmptyTermSubst(e) &
           UnifyingSubst(t,t1,e,s).


PREDICATE  UnifyingSubst : Term * Term * TermSubst * TermSubst.

UnifyingSubst(t,t1,s,s1) <-
           ApplySubstToTerm(t,s,ts) &
           ApplySubstToTerm(t1,s,t1s) &
           UnifyingSubst1(ts,t1s,s,s1).


PREDICATE  UnifyingSubst1 : Term * Term * TermSubst * TermSubst.

UnifyingSubst1(t,t,s,s).
UnifyingSubst1(t1,v,s,s1) <-
           Variable(v) &
           NotOccur(v,t1) &
           BindingToTermSubst(v,t1,r) &
           ComposeTermSubsts(r,s,s1).
UnifyingSubst1(v,t1,s,s1) <-
           Variable(v) &
           NotOccur(v,t1) &
           BindingToTermSubst(v,t1,r) &
           ComposeTermSubsts(r,s,s1).
UnifyingSubst1(t,t1,s,s1) <-
           FunctionTerm(t,x,tas) &
           FunctionTerm(t1,x,tas1) &
           UnifyingSubst2(tas,tas1,s,s1).


PREDICATE  UnifyingSubst2 : List(Term) * List(Term) * TermSubst * TermSubst.

UnifyingSubst2([],[],s,s).
UnifyingSubst2([t|ts],[t1|ts1],s,s1) <-
           UnifyingSubst(t,t1,s,s2) &
           UnifyingSubst2(ts,ts1,s2,s1).
```

```
PREDICATE  NotOccur : Term * Term.

NotOccur(s,t) <-
           s ~= t &
           ALL [t1] (NotOccur(s,t1) <- Subterm(t,t1)).


PREDICATE  Subterm : Term * Term.

Subterm(t,t1) <-
           FunctionTerm(t,_,ts) &
           Member(t1,ts).
```

9.2 Programs

We will turn shortly to a discussion of the predicates provided by `Programs`. But before this, we need to introduce the concepts of the flat name of a symbol and the flat form of a program.

For clarity in this discussion, what we called before the *name* of a symbol, we will now call the *declared name* of the symbol. Since module condition M3 is enforced, a symbol 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. (For bases, constants, and propositions, the arity is extraneous, of course. We include it in these cases for uniformity.) Such a quadruple (`M`, `S`, `C`, `A`), where `M` is the name of a module, `S` is the declared name of a symbol which is declared in the 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 the "program" 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` *wrt a program* `P` is the subset of the flat language of `P` given by the set of language declarations of all symbols which have a language declaration in the language of `M` wrt `P`. The *flat export language of a module* `M` *wrt a program* `P` is the subset of the flat language of `P` given by the set of language declarations of all symbols which have a language declaration in the export language of `M` wrt `P`.

Note carefully that, for the ground representation, it is the *flat form* of a program which is represented rather than the program itself. The reason is that the flat form of a program is much more convenient because it does not have any name clashes. Hence, such things as explicit derivation steps in an interpreter can be achieved by taking the current resultant [10], choosing an appropriate input statement, and doing a derivation step, without having to be concerned with name clashes.

Another point to note is that bar commits and one-solution commits are preprocessed away, as explained in section 7.2, so that only the representations of commits with labels are present in the ground representation of a program. Also module declarations, language declarations, control declarations, and statements appear in some definite, but unspecified, order in the ground representation of a program. The only time this order is visible to a programmer is when the predicate `DefinitionInProgram` is called. This predicate returns the collection of statements in the definition of a proposition or predicate in this order.

A typical meta-program is a debugger. Taking the declarative debugger in [15, page 124] as an example, we see that the debugger switches between manipulating formulas in the bodies of program statements and running goals to the program being debugged. The manipulation of the formulas takes place in the flat language of the appropriate module of the program. The running of a goal takes place in the flat language of the program.

The module `Programs` imports the module `Syntax`. It also declares the following bases.

```
BASE       Program, ModulePart, Condition.
```

`Program` is the type of a term representing (the flat form of) a program. `ModulePart` is the type of constants representing the keywords `EXPORT`, `LOCAL`, `CLOSED`, and `MODULE`. Thus there are four constants of type `ModulePart` with declarations as follows.

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

`Condition` is the type of a term representing a condition in a `DELAY` declaration.

We now discuss in some detail a representative selection of predicates provided by `Programs`, the complete export part of which is given in chapter 13. The first of these predicates is `TermInProgram`, which has the following declaration.

```
PREDICATE  TermInProgram : Program  * VarTyping * Term * Type * VarTyping.
```

`TermInProgram` is intended to be true when its first argument is the representation of a program, the second argument is the representation of a variable typing in the flat language of this program, its third argument is the representation of a term in this language, the fourth is the representation of the type of this term with respect to this variable typing, and the fifth is the representation of the variable typing obtained by combining the variable typing in the second argument with the types of all variables occurring in the term.

The next collection of predicates in `Programs` is concerned with the representation of types, terms, and formulas.

```
PREDICATE  StringToProgramType : Program * String * String * List(Type);
           StringToProgramTerm : Program * String * String * List(Term);
           StringToProgramFormula : Program * String * String * List(Formula).
```

`StringToProgramType` is intended to be 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 a string, and its fourth argument is the list (in a definite order) of representations of types in the flat language of this module wrt this program whose string representation is the third argument. There may be more than one such type due to overloading. Thus the call `StringToProgramType(`P`,` M`,"List(Integer)",x)`, where P is a term representing a program, M is the name of a module in this program, and the constructor `List` and the base `Integer` are in the language of M wrt P, would bind `x` to the list consisting of the term representing the type `List(Integer)` (assuming no overloading). Similarly, `StringToProgramTerm` is intended to be 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 a string, and its fourth argument is the list (in a definite order) of representations of terms in the flat language of this module wrt to this program whose string representation is the third argument. Thus the call `StringToProgramTerm(`*P*`,`*M*`,"v1",x)`, where *P* is a term representing a program and *M* is the name of a module in this program, would bind `x` to the list consisting of the term representing the variable `v1`. Note that the ground representation represents actual names of variables, which can be recovered by calling the companion predicate `ProgramTermToString` (see chapter 13) with the first argument instantiated to a term representing a program, the second argument to the name of a module in this program, the third argument to a variable, and the fourth argument instantiated to the term representing the variable whose name is to be recovered. `StringToProgramFormula` is intended to be 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 string, and its fourth argument is the list (in a definite order) of representations of formulas in the flat language of this module wrt this program whose string representation is the third argument. `StringToProgramFormula` provides a convenient way of giving a meta-program the representation of a goal to an object program. For example, the call `StringToProgramFormula(`*P*`,`*M*`,"<-Append(x,y,[1,2,3])",z)`, where *P* is a term representing a program and *M* is the name of a module in this program, would bind `z` to the list consisting of the term representing the goal `<- Append(x,y,[1,2,3])`.

Of course, facilities for creating the representation of object *programs* have also to be provided. Since object programs are usually of a substantial size, Gödel handles the creation of their representation differently from the representation of types, terms, and so on. The discussion of how this is done is postponed to later in this section.

The next collection of predicates is concerned with modules and accessing statements in open modules.

```
PREDICATE  OpenModule : Program * String;
           DeclaredInOpenModule : Program * String * Formula;
           StatementInModule : Program * String * Formula;
           StatementMatchAtom : Program * String * Formula * Formula.
```

`OpenModule` is intended to be true when its first argument is the representation of a program and its second argument is the name of an open module in this program. `DeclaredInOpenModule` is intended to be true when its first argument is the representation of a program, its second argument is the name of an open module in this program, and its third argument is the representation of an atom in the flat language of this program whose proposition or predicate is declared in this module. `StatementInModule` is intended to be true when its first argument is the representation of a program, its second argument is the name of an open module in this program, and its third argument is the representation of a statement in this module. `StatementMatchAtom` is intended to be true when its first argument is the representation of a program, its second argument is the name of an open module in this program, its third argument is the representation of an atom in the flat language of this program, and its fourth argument is the 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.

For the sake of uniformity, a statement with an empty body is represented, not as an atom, but as a formula with top-level connective `<-` and an empty formula to the right of the `<-`. Thus to access the head or body of a statement, `StatementInModule` or `StatementMatchAtom` are first used to access the statement and then `IsImpliedBy` from `Syntax` is used to access the head or body. An empty body can be checked for using `EmptyFormula` from `Syntax`.

The next three predicates are concerned with language and import declarations in a program.

```
PREDICATE  ConstructorInModule : Program * String * ModulePart * Name *
                                 Integer * String;
           ConstantInModule : Program * String * ModulePart * Name * Type *
                              String;
           ImportInModule : Program * String * ModulePart * String.
```

`ConstructorInModule` is intended to be 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 part keyword of this module which cannot be the local part if this module is closed, its fourth argument is the representation of the flat name of a constructor accessible to this part of this module, its fifth argument is the arity of this constructor, and its sixth argument is the name of the module in which the constructor is declared. Thus the call `ConstructorInModule(`P`, "M6", Module, `N`, 1, "Lists")`, where P is the term representing the program {`M6`, `M5`, `Lists`, `Integers`} and N is the term representing the flat name of `List`, would succeed. `ConstantInModule` is intended to be 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 part keyword of this module which cannot be the local part if this module is closed, its fourth argument is the representation of the flat name of a constant accessible to this part of this module, its fifth argument is the representation of the type of this constant, and its sixth argument is the name of the module in which the constant is declared. `ImportInModule` is intended to be 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 part keyword of this module which cannot be the local part if this module is closed, and its fourth argument is the name of a module appearing in an `IMPORT` declaration in this part of this module. Note that none of these predicates gives access to declarations or statements in the local part of a closed module.

The next two predicates are needed for dynamic meta-programming.

```
PREDICATE  InsertProgramBase : Program * String * ModulePart * Name * Program;
           DeleteStatement : Program * String * Formula * Program.
```

`InsertProgramBase` is intended to be true when its first argument is the representation of a program, its second argument is the name of an open module in this program, its third argument is the representation of a part keyword of this module, its fourth argument is the representation of the flat name of a base not declared in this part of this module, and its fifth argument is the 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. `DeleteStatement` is intended to be true when its first argument is the representation of a program, its second argument is the name of an open module in this program, its third argument is the representation of a statement in the flat language of this module wrt this program appearing in this module, and its fourth argument is the representation of a program which differs from the program in the first argument only in that it does not contain this statement in this part of this module.

The final two predicates are concerned with running goals for a program.

```
PREDICATE  Succeed : Program * Formula * TermSubst;
           Fail : Program * Formula.
```

`Succeed` is intended to be true when the first argument is the representation of a program, its second argument is the representation of the body of a goal in the flat language of this program, and its third argument is the representation of a computed answer for this goal and the flat form of this program. The predicate `Succeed` is the Gödel equivalent of the standard *demo* predicate. Note that, in particular, `Succeed` can be used to run goals containing predicates in the export parts of closed modules. `Fail` is intended to be true when its first argument is the representation of a program and its second argument is the representation of the body of a goal in the flat language of this program such that this goal and the flat form of this program have a finitely failed search tree. Note that `Succeed` and `Fail` (and their companions `SucceedAll`, `Compute`, and `ComputeAll`) are the only predicates in `Programs` which have access to statements in closed modules.

Next, we discuss how Gödel handles the creation of the ground representation of object programs. There are two utilities, `program-compile` and `program-decompile`. The command

```
program-compile Main
```

takes the program with main module `Main` in the current directory, produces the ground representation of this program, which is a term of type `Program`, and puts this term into a file called `Main.prm` in the current directory, overwriting any previously existing file with this name. The command

```
program-decompile Main
```

finds in the current directory the file `Main.prm` which contains a term of type `Program` representing an object program, creates the user-defined modules of this object program, and puts files containing these modules in the current directory, overwriting any previously existing files with the same names. The original files are saved with a `.old` extension.

As an example of the use of the module `Programs`, we give below a module containing the predicate `RemoveAll`, which removes statements whose heads unify with a given atom.

9.3 ProgramsIO

So that a meta-program can access a term representing a program, Gödel provides a system module, called `ProgramsIO`, the export part of which is given below. `ProgramsIO` imports `IO` so that the predicates there can be used to open and close files. The predicate `GetProgram` is intended to be called with its first argument ground and instantiates the second argument to the term representing the program in the file with extension `.prm` corresponding to the input stream in the first argument. The predicate `PutProgram` is intended to be called with both arguments ground. It puts the term in the second argument into the file with extension `.prm` corresponding to the output stream in the first argument. The output stream must have been found using `FindOutput`, so that the term being written overwrites any existing term in this file. Note that `ProgramsIO` is an input/output module so it should be handled like the module `IO`. In particular, it is preferable to arrange the module structure of the meta-program so that only its main module depends upon `ProgramsIO`.

```
EXPORT      Dynamic.

IMPORT      Programs.

PREDICATE   RemoveAll :

  Program       % The representation of a program P.
* String        % The name of an open module in P.
* Formula       % The representation of an atom in the flat language of
                % this module wrt P.
* Program.      % The representation of the program obtained from P by
                % deleting all statements in this module whose heads unify
                % with this atom (after standardization apart).

DELAY       RemoveAll(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).


PREDICATE   Remove :

  Program       % The representation of a program P.
* String        % The name of an open module in P.
* Formula       % The representation of an atom in the flat language of
                % this module wrt P.
* Program.      % The representation of the program obtained from P by
                % deleting a statement in this module whose head unifies
                % with this atom (after standardization apart).

DELAY       Remove(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).
```

9.4 Scripts

The module `Scripts` discussed in this section is a rather specialised meta-programming module whose main purpose is to support partial evaluation. We begin by explaining the need for such a module.

Partial evaluation and module systems do not fit easily together. The reason for this is that the partial evaluation process naturally “promotes” symbols to a module higher in the module hierarchy than where they are accessible and thus the module system is broken. A natural solution to this problem is to flatten out the module structure of a program before any partial evaluation takes place. We are thus led to the concept of a script which is essentially the flat form of a program with the module structure removed. We now turn to the appropriate definitions.

A *standard script* is what is obtained by removing all module declarations from the flat form of a program and concatenating what remains of the modules making up the program. The program

```
LOCAL       Dynamic.

% PREDICATE   RemoveAll : Program * String * Formula * Program.

RemoveAll(program,module,atom,new_program) <-
            IF SOME [program1] Remove(program,module,atom,program1)
            THEN
              RemoveAll(program1,module,atom,new_program)
            ELSE
              new_program = program.


% PREDICATE   Remove : Program * String * Formula * Program.

Remove(program,module,atom,new_program) <-
            StatementMatchAtom(program,module,atom,statement) &
            IsImpliedBy(head,_,statement) &
            RenameFormulas([head],[atom],[renamed_atom]) &
            EmptyTermSubst(empty_subst) &
            UnifyAtoms(head,renamed_atom,empty_subst,_) &
            DeleteStatement(program,module,statement,new_program).
```

```
EXPORT       ProgramsIO.

IMPORT       IO, Programs.

PREDICATE    GetProgram : InputStream * Program.
DELAY        GetProgram(x,_) UNTIL GROUND(x).

PREDICATE    PutProgram : OutputStream * Program.
DELAY        PutProgram(x,y) UNTIL GROUND(x) & GROUND(y).
```

from which the script is obtained is called the *associated program* and the script is called the *associated script* (for this program). The predicate `ProgramToScript` discussed below performs this operation. More generally, a *script* is what is obtained from a standard script by a (possibly empty) sequence of applications of the insert and delete predicates in the module `Scripts` (which we discuss below). Thus scripts do not have a fully independent existence but instead can only be created initially from a program and then can only be modified in the restricted ways allowed by the insert and delete predicates. Note also that scripts do not have a textual format as programs do – they only exist in "flattened" form.

A script is divided into two parts. The *closed part* of a script consists of (the flattened forms of) all language declarations, control declarations, and statements which appeared in the local parts of the closed modules of the associated program from which the script was originally constructed. The *open part* of a script is the complement in the script of the closed part. Note that only the open part of a script can be modified.

There are several languages associated with a script. The *language of a script* is the polymorphic many-sorted language given by all the language declarations in the script. The language of a script is naturally "flat", but we omit this qualifier since scripts do not exist in non-flattened form. The *open language of a script* is the sublanguage of the language of the script given by the declarations which appear in the open part of the script. The *goal language* of a standard script is the (flattened form of) the goal language of the associated program. For a script in general, the *goal language* is modified as follows. Any symbol inserted into the script is added to the goal language of the script. If a symbol which appears in the goal language is deleted from the script, then this symbol is also deleted from the goal language.

Now let us turn to the module `Scripts` itself. First, `Scripts` imports `Programs` into its export part. Next, we describe a representative collection of predicates in `Scripts`.

The first predicate `ProgramToScript` has the following declaration.

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

`ProgramToScript` is intended to be true when its first argument is the representation of a program and its second argument is the representation of the associated script obtained from this program. This predicate has a `DELAY` declaration which requires the first argument be given, so that it can only be used to construct the associated script from the given program (or check a given script is associated with the given program). Typically, `ProgramToScript` would be used during the running of a partial evaluator to produce a standard script from the program to be partially evaluated. The appropriate resultants would be produced by the partial evaluator which would then replace (some of) the statements in the standard script. This modified script is now the partially evaluated form of the program and can be script compiled by the system and run (analogously to the way programs are compiled and run). Legal goals to the script are goals whose flattened form is a formula in the goal language of the script.

The next two predicates are concerned with accessing the declarations of symbols and statements in a script.

```
PREDICATE  BaseInScript : Script * Name;
           StatementInScript : Script * Formula.
```

`BaseInScript` is intended to be true when its first argument is the representation of a script and its second argument is the representation of the flat name of a base in the open language of the script. `StatementInScript` is intended to be true when its first argument is the representation of a script and its second argument is the representation of a statement in the open part of this script.

Finally, we give two predicates needed for dynamic meta-programming.

```
PREDICATE  DeleteScriptProposition : Script * Name * Script;
           InsertStatement : Script * Formula * Script.
```

`DeleteScriptProposition` is intended to be true when its first argument is the representation of a script, its second argument is the representation of the flat name of a proposition whose module name component is not the name of a closed module and which is declared in the open part of this script, and its third argument is the representation of a script which differs from the script in the first argument only in that it does not contain the declaration of this proposition in its open part. `InsertStatement` is intended to be true when its first argument is the representation of a script, its second argument is the representation of a statement in the open language of this script, and its third argument is the representation of a script which differs from the script in the first argument only in that it also contains this statement in the open part of the script.

For a more detailed description of the use of the meta-programming modules, including `Scripts`, for partial evaluation, the reader is referred to [8] and [9]. Many important issues relevant to partial evaluation, such as the use of the ground representation and self-applicability, are discussed in these two papers.

9.5 ScriptsIO

To allow a meta-program to access a term representing a script, Gödel provides a system module, called `ScriptsIO`. The predicate `GetScript` is intended to be called with its first argument ground and instantiates the second argument to the term representing the script in the file with extension `.scr` corresponding to the input stream in the first argument. The predicate `PutScript` is intended to be called with both arguments ground. It puts the term in the second argument into the file with extension `.scr` corresponding to the output stream in the first argument. The output stream must have been found using `FindOutput`, so that the term being written overwrites any existing term in this file.

```
EXPORT       ScriptsIO.

IMPORT       IO, Scripts.

PREDICATE    GetScript : InputStream * Script.
DELAY        GetScript(x,_) UNTIL GROUND(x).

PREDICATE    PutScript : OutputStream * Script.
DELAY        PutScript(x,y) UNTIL GROUND(x) & GROUND(y).
```

9.6 Theories

The ground representation in this section is a scheme for representing object (many-sorted) theories, theorems, declarations, terms, and so on, as terms in a meta-language. A theory is like the local part of a module having only a local part, except that it does not contain conditionals, commits, or control declarations, and it contains arbitrary first order formulas instead of statements. A theory can import system modules.

A typical application for the system module `Theories` could be a program synthesis system. For such a system, the specification for a program would be expressed as a theory, which perhaps imported some system modules. Then a meta-program would perform transformation on this theory until it took the form of (essentially) a module. After further modification (for example, adding control information and/or employing conditionals), the synthesized program would result.

Before we turn to the details of the module `Theories`, we give some definitions. The *language of a theory* is the polymorphic many-sorted language given by the language declarations of all symbols accessible to the theory. The *flat form* of a theory `T` is the "theory" obtained from `T` by replacing each occurrence of the declared name of a symbol in `T` by the flat name of the symbol. The *flat language of a theory* is the polymorphic many-sorted language obtained from the language of the theory by replacing the declared name of each symbol appearing in a declaration by its flat name.

The module `Theories` imports the module `Syntax`. It also declares the following base.

```
BASE       Theory.
```

`Theory` is the type of a term representing (the flat form of) an (object) theory.

We now discuss in some detail a representative selection of predicates provided by `Theories`. The complete export part of `Theories` is given in chapter 13. The first of the predicates provided by `Theories` which we discuss here is `TermInTheory`, which has the following declaration.

```
PREDICATE  TermInTheory : Theory * VarTyping * Term * Type * VarTyping.
```

`TermInTheory` is intended to be true when its first argument is the representation of a theory, the second argument is the representation of a variable typing in the flat language of this theory, its third argument is the representation of a term in this language, the fourth is the representation of the type of this term with respect to this variable typing, and the fifth is the representation of the variable typing obtained by combining the variable typing in the second argument with the types of all variables occurring in the term.

The next collection of predicates in `Theories` is concerned with the representation of types and formulas.

```
PREDICATE  StringToTheoryType : Theory * String * List(Type);
           StringToTheoryFormula : Theory * String * List(Formula).
```

`StringToTheoryType` is intended to be true when its first argument is the representation of a theory, its second argument is a string, and its third argument is the list (in a definite order) of representations of types in the language of this theory whose string representation is the second argument. Thus the call `StringToTheoryType(`T`,"Pair(Person)",x)`, where T is a term representing a theory, and the constructor `Pair` and the base `Person` are in the language of

T, would bind `x` to the list consisting of the term representing the type `Pair(Person)` (assuming no overloading). `StringToTheoryFormula` is intended to be true when its first argument is the representation of a theory, its second argument is a string, and its third argument is the list (in a definite order) of representations of formulas in the language of this theory whose string representation is the second argument.

The next predicate is concerned with accessing axioms in theories.

```
PREDICATE  AxiomInTheory : Theory * Formula.
```

`AxiomInTheory` is intended to be true when its first argument is the representation of a theory and its second argument is the representation of an axiom in this theory.

The next two predicates are concerned with language declarations in a theory.

```
PREDICATE  ConstructorInTheory : Theory * Name * Integer * String;
           ConstantInTheory : Program * Name * Type * String.
```

`ConstructorInTheory` is intended to be true when its first argument is the representation of a theory, its second argument is the representation of the flat name of a constructor accessible to this theory, its third argument is the arity of this constructor, and its fourth argument is the name of the theory or the module in which this constructor is declared. `ConstantInTheory` is intended to be true when its first argument is the representation of a theory, its second argument is the representation of the flat name of a constant accessible to this theory, its third argument is the representation of the type of this constant, and its fourth argument is the name of the theory or the module in which this constant is declared.

The next two predicates are needed for dynamic meta-programming.

```
PREDICATE  InsertTheoryBase : Theory * Name * Theory;
           DeleteAxiom : Theory * Formula * Theory.
```

`InsertTheoryBase` is intended to be true when its first argument is the representation of a theory, its second argument is the representation of the flat name of a base not declared in this theory, and its third argument is the representation of a theory which differs from the theory in the first argument only in that it also contains the declaration of this base. `DeleteAxiom` is intended to be true when its first argument is the representation of a theory, its second argument is the representation of a formula in the flat language of this theory in this theory, and its third argument is the representation of a theory which differs from the theory in the first argument only in that it does not contain this formula as an axiom.

The final predicate is concerned with proving theorems.

```
PREDICATE  Prove : Theory * Formula.
```

`Prove` is intended to be true when the first argument is the representation of a theory and its second argument is the representation of a theorem of this theory.

Next, we discuss how Gödel handles the creation of the ground representation of theories. There are two utilities, `theory-compile` and `theory-decompile`. The command

```
theory-compile Name
```

takes the theory with name `Name` in the current directory, produces the ground representation of this theory, which is a term of type `Theory`, and puts this term into a file called `Name.thy` in the current directory, overwriting the contents of any previously existing file with this name. The command

```
theory-decompile Name
```

finds in the current directory the file `Name.thy` which contains a term of type `Theory` representing a theory, creates the corresponding theory, and puts a file containing this theory in the current directory, overwriting any previously existing file with the same name. The original file is saved with a `.old` extension.

9.7 TheoriesIO

So that a meta-program can access a term representing a theory, Gödel provides a system module, called `TheoriesIO`. `TheoriesIO` imports `IO` so that the predicates there can be used to open and close files. The predicate `GetTheory` is intended to be called with its first argument ground and instantiates the second argument to the term representing the theory in the file with extension `.thy` corresponding to the input stream in the first argument. The predicate `PutTheory` is intended to be called with both arguments ground. It puts the term in the second argument into the file with extension `.thy` corresponding to the output stream in the first argument. The output stream must have been found using `FindOutput`, so that the term being written overwrites any existing term in this file.

```
EXPORT       TheoriesIO.

IMPORT       IO, Theories.

PREDICATE    GetTheory : InputStream * Theory.
DELAY        GetTheory(x,_) UNTIL GROUND(x).

PREDICATE    PutTheory : OutputStream * Theory.
DELAY        PutTheory(x,y) UNTIL GROUND(x) & GROUND(y).
```

Chapter 10

Example Programs

In this chapter, we give programs to further illustrate the Gödel programming style.

10.1 Fibonacci

The first example program is an iterative Fibonacci program. Given the first argument, `Fib` finds the Fibonacci number having rank equal to the first argument. `Fib` can be run in reverse to find the rank of a Fibonacci number. It can also be run with both arguments variables to generate all Fibonacci numbers and their ranks. For example, the goal

```
<- Fib(235,y).
```

gives the answer

```
y = 5789092068864820527338372482892113982249794889765
```

the goal

```
<- Fib(x,5789092068864820527338372482892113982249794889765).
```

gives the answer

```
x = 235
```

and the goal

```
<- Fib(x,y).
```

gives the answers

```
x = 0
y = 0

x = 1
y = 1

x = 2
y = 1

x = 3
y = 2

x = 4
y = 3

x = 5
y = 5
```

and so on.

```
MODULE    Fibonacci.

IMPORT    Integers.


PREDICATE Fib : Integer * Integer.

% Fib(k,n) <-> n is the Fibonacci number F_{k} of rank k.

Fib(0,0).
Fib(1,1).
Fib(k,n) <-
          k > 1 &
          FibIt(k-2,1,1,n).


PREDICATE FibIt : Integer * Integer * Integer * Integer.

% FibIt(k,f,g,n) <-> n  =  F_{k} * f  +  F_{k+1} * g.

FibIt(0,_,g,g).
FibIt(k,f,g,n) <-
          k > 0 &
          g < n &
          FibIt(k-1,g,f+g,n).
```

10.2 WolfGoatCabbage

The next program illustrates the use of extensional set terms. It is a solution of the well-known wolf-goat-cabbage problem, as described in [19, pages 285–287]. The module `Search` provides a depth-first state-transition search framework for problem solving, similar to the one in [19, page 285] and adapted here for the wolf-goat-cabbage problem.

The program is invoked with the goal

```
<- Run(x).
```

and produces the answers

```
x =
[LeftToRight({Farmer,Goat}),RightToLeft({Farmer}),LeftToRight({Cabbage,Farmer}),
RightToLeft({Farmer,Goat}),LeftToRight({Farmer,Wolf}),RightToLeft({Farmer}),
LeftToRight({Farmer,Goat})]

x =
[LeftToRight({Farmer,Goat}),RightToLeft({Farmer}),LeftToRight({Farmer,Wolf}),
RightToLeft({Farmer,Goat}),LeftToRight({Cabbage,Farmer}),RightToLeft({Farmer}),
LeftToRight({Farmer,Goat})].
```

```
MODULE       Search.

% A depth-first state-transition search framework for problem solving, similar
% to the one on page 285 of Sterling and Shapiro, The Art of Prolog. It is used
% here to solve the wolf-goat-cabbage problem, also described in Sterling and
% Shapiro, pages 285-287.

IMPORT       Lists, WolfGoatCabbage.


PREDICATE    Solve : State * List(State) * List(Move).

Solve(current_state, _, []) <-
             Final(current_state).
Solve(current_state, history, [move|moves]) <-
             Applicable(move, current_state) &
             ApplyMove(move, current_state, new_state) &
             Legal(new_state) &
             NoLoops(new_state, history) &
             Solve(new_state, [new_state|history], moves).


PREDICATE  NoLoops : State * List(State).

NoLoops(_, []).
NoLoops(st, [st1|rest]) <-
             st ~= st1 &
             NoLoops(st, rest).


PREDICATE    Run : List(Move).

Run(moves) <-
             Initial(initial_state) &
             Solve(initial_state, [initial_state], moves).
```

```
EXPORT     WolfGoatCabbage.

% The wolf-goat-cabbage problem, as described in Sterling and Shapiro,
% The Art of Prolog, pages 285-287.

IMPORT     Lists, Sets.

BASE       Object, State, Move.

CONSTANT   Farmer, Wolf, Goat, Cabbage : Object.

FUNCTION   St : Set(Object) * Set(Object) -> State;
           LeftToRight : Set(Object) -> Move;
           RightToLeft : Set(Object) -> Move.

PREDICATE  Initial, Final : State;
           Applicable : Move * State;
           ApplyMove : Move * State * State;
           Legal : State.
```

```
LOCAL WolfGoatCabbage.


% PREDICATE  Initial : State.

Initial(St({Farmer,Wolf,Goat,Cabbage}, {})).


% PREDICATE  Final : State.

Final(St({}, {Farmer,Wolf,Goat,Cabbage})).


% PREDICATE  Applicable : Move * State.

Applicable(LeftToRight({Farmer}), St(left,_)) <-
               Farmer In left.

Applicable(RightToLeft({Farmer}), St(_,right)) <-
               Farmer In right.

Applicable(LeftToRight({Farmer,x}), St(left,_)) <-
               {Farmer,x} Subset left &
               x In {Wolf,Goat,Cabbage}.

Applicable(RightToLeft({Farmer,x}), St(_,right)) <-
               {Farmer,x} Subset right &
               x In {Wolf,Goat,Cabbage}.


% PREDICATE  ApplyMove : Move * State * State.

ApplyMove(LeftToRight(cargo), St(left,right), St(left\cargo,right+cargo)).

ApplyMove(RightToLeft(cargo), St(left,right), St(left+cargo,right\cargo)).


% PREDICATE  Legal : State.

Legal(St(left,right)) <-
               ~ Illegal(left) &
               ~ Illegal(right).
```

```
PREDICATE Illegal : Set(Object).

Illegal(bank) <-
               ~ Farmer In bank &
               ({Goat,Cabbage} Subset bank \/ {Wolf,Goat} Subset bank).
```

10.3 Tableau

The next program, which is a tableau propositional theorem prover, illustrates the use of the system modules `Units` and `Flocks`. In this program, a propositional theory is represented as a flock. For example, the theory

```
(A & B) -> ~~E.
E -> (C & D).
A -> F.
F -> A.
(E \/ ~F) -> D.
```

would appear in a flock as

```
->(&(A,B),~(~(E))).
->(E,&(C,D)).
->(A,F).
->(F,A).
->(\/(E,~(F)),D).
```

This flock would be flock-compiled. Then typical queries to the program could be

```
<- TestProve("theory", "->(B,\\/(C,D))", indicator).
```

to which the answer is `indicator = YES`, and

```
<- TestProve("theory", "&(A,B)", indicator).
```

to which the answer is `indicator = NO`.

We remark on two points of interest in this program. The first is that the theorem prover has been written in a "non-failing" style so that the predicate `Prove` succeeds whether the formula is a theorem or not. This is achieved by adding an extra argument to `Prove` which records whether there was a proof or not. This style has some important advantages over the more usual one whereby exceptions of one kind or another are captured by failure. These are that the code which results from the "non-failing" style tends to have a clearer declarative reading and that there is usually much less need for pruning. For example, in the following program, no pruning is needed at all.

The other point is that the use of abstract data types means that all the heads of the statements in the predicate `Apply` are most general and it is only in the body by using `UnitParts` that we determine which case applies. Having such general heads could mean that no indexing could be done which leads to inefficiencies when the program is run. However, partial evaluation of the calls to `UnitParts` can be used to push the structure of the first argument of `Apply` back into the head. This process can be made invisible to the programmer so that we end up with the advantages of using abstract data types and also the efficiency obtained from having as much structure in the heads of statements as possible.

```
MODULE      TestTableau.

IMPORT      FlocksIO, Tableau.


PREDICATE   TestProve :

  String      % The name of a file (without the .flk extension) containing the
              % flock representation of a propositional theory.
* String      % The string representation of a propositional formula.
* Indicator.  % YES, if the formula is a theorem, and NO, if it is not.

TestProve(theory_string, formula_string, indicator) <-
    FindInput(theory_string ++ ".flk", In(stream)) &
    GetFlock(stream, theory) &
    EndInput(stream) &
    StringToUnit(formula_string, formula) &
    Prove(theory, formula, indicator).

DELAY       TestProve(x,y,_) UNTIL GROUND(x) & GROUND(y).
```

```
EXPORT      Tableau.

% This module contains an implementation of a tableau propositional theorem
% prover as described, for example, in "Logic for Computer Science", S. Reeves
% and M. Clarke, Addison-Wesley, 1990, pages 64-75. The connectives admitted
% are &, \/, ->, and ~.

IMPORT      Flocks.

BASE        Indicator.
%
% The type of the constants YES, indicating a formula is a theorem of a
% theory, and NO, indicating that it is not.


CONSTANT    YES, NO : Indicator.


PREDICATE   Prove :

  Flock          % A propositional theory.
* Unit           % A propositional formula.
* Indicator.     % YES, if the formula is a theorem of the theory, and NO,
                 % if it is not.

DELAY       Prove(x,y,_) UNTIL GROUND(x) & GROUND(y).
```

```
LOCAL      Tableau.


IMPORT     Sets.


BASE       Leaf, Node, Tableau.


FUNCTION   Nd :

  Integer         % A pointer.
* Unit            % The propositional formula in the node pointed to by this
                  % pointer.
-> Node.

FUNCTION   Lf :

  Integer         % A pointer to an open leaf node. (A leaf node is open if it
                  % is at the end of a non-closed branch in a tableau.)
* Set(Integer)    % The set of pointers to the ancestors of this node plus the
                  % pointer to the node itself.
-> Leaf.

FUNCTION   Tb :

  Set(Leaf)       % The set of open leaf nodes in a tableau.
* Set(Node)       % The set of all nodes in this tableau.
* Set(Integer)    % The set of pointers to nodes in this tableau which have yet
                  % to be expanded and lie on non-closed branches.
* Integer         % The maximum value of pointers in this tableau.
-> Tableau.


% PREDICATE  Prove : Flock * Unit * Indicator.

Prove(theory, theorem, indicator) <-
    UnitParts(neg_thm, "~", [theorem]) &
    (IF Literal(neg_thm) THEN exp = {} ELSE exp = {1}) &
    InitialiseTableau(theory,Tb({Lf(1,{1})},{Nd(1,neg_thm)},exp,1), tableau) &
    Unsatisfiable(tableau, indicator).


PREDICATE  InitialiseTableau :
  Flock           % A propositional theory.
* Tableau         % A tableau containing only the negation of the formula to be
                  % proved.
* Tableau.        % The tableau containing just the axioms in this theory and the
                  % negation of the formula to be proved.

InitialiseTableau(theory, Tb(leaves,nodes,exp,max), tableau) <-
    IF SOME [f,n] UnitInFlock(theory, f, n)
    THEN
      DeleteUnit(theory, f, n, new_theory) &
      Lf(max,anc) In leaves &
      (IF Closed(f,anc,nodes)
       THEN
         tableau = Tb({},{},{},0)            % Tableau is closed.
       ELSE
         (IF Literal(f) THEN e = {} ELSE e = {max+1}) &
         InitialiseTableau(new_theory,
           Tb({Lf(max+1,anc+{max+1})},nodes+{Nd(max+1,f)},exp+e,max+1),
           tableau)
      )
    ELSE
      tableau = Tb(leaves,nodes,exp,max).


PREDICATE  Unsatisfiable :

  Tableau         % A tableau.
* Indicator.      % YES, if the tableau can be expanded to an unsatisfiable one,
                  % and NO, otherwise.

Unsatisfiable(Tb({},_,_,_), YES).

Unsatisfiable(Tb(leaves,nodes,exp,max), indicator) <-
    leaves ~= {} &
    IF  SOME [n] n In exp
    THEN
      new_exp = exp\{n} &
      Nd(n,f) In nodes &
      leaf_ptrs = {m : SOME [anc] ((Lf(m,anc) In leaves) & (n In anc))} &
      (IF leaf_ptrs = {}
       THEN
         tableau = Tb(leaves,nodes,new_exp,max)
       ELSE
         Apply(f, leaf_ptrs, Tb(leaves,nodes,new_exp,max), tableau)
      ) &
      Unsatisfiable(tableau, indicator)
    ELSE
      indicator = NO.


PREDICATE  Closed :

  Unit           % A formula closed wrt to the formulas in the nodes in the
                 % third argument.
* Set(Integer)   % A set of pointers to nodes.
* Set(Node).     % A set of nodes.

Closed(f, ptrs, nodes) <-
    m In ptrs &
    Nd(m,f1) In nodes &
    (UnitParts(f, "~", [f1]) \/ UnitParts(f1, "~", [f])).


PREDICATE  Literal :

  Unit.          % A propositional formula which is a literal.

Literal(f) <-
    UnitParts(f, _, []).

Literal(f) <-
    UnitParts(f, "~", [f1]) &
    UnitParts(f1, _, []).


PREDICATE  Apply :

  Unit           % A propositional formula which is not a literal.
* Set(Integer)   % A set of pointers to open leaf nodes.
* Tableau        % A tableau containing these open leaf nodes.
* Tableau.       % The tableau obtained by extending this tableau at each of
                 % these open leaf nodes according to this formula.

Apply(f, leaf_ptrs, Tb(leaves,nodes,exp,max), tableau) <-
    UnitParts(f, "\\/", [f1, f2]) &             % f = f1 \/ f2
    ExpandOr(f1,f2,leaf_ptrs,Tb(leaves,nodes,exp,max),tableau).

Apply(f, leaf_ptrs, Tb(leaves,nodes,exp,max), tableau) <-
    UnitParts(f, "->", [f1, f2]) &              % f = f1 -> f2
    UnitParts(notf1, "~", [f1]) &
    ExpandOr(notf1,f2,leaf_ptrs,Tb(leaves,nodes,exp,max),tableau).

Apply(f, leaf_ptrs, Tb(leaves,nodes,exp,max), tableau) <-
    UnitParts(f, "&", [f1, f2]) &               % f = f1 & f2
    ExpandAnd(f1,f2,leaf_ptrs,Tb(leaves,nodes,exp,max),tableau).

Apply(f, leaf_ptrs, Tb(leaves,nodes,exp,max), tableau) <-
    UnitParts(f, "~", [f1]) &
    UnitParts(f1, "&", [f2, f3]) &              % f = ~(f2 & f3)
    UnitParts(notf2, "~", [f2]) &
    UnitParts(notf3, "~", [f3]) &
    ExpandOr(notf2,notf3,leaf_ptrs,Tb(leaves,nodes,exp,max),tableau).

Apply(f, leaf_ptrs, Tb(leaves,nodes,exp,max), tableau) <-
    UnitParts(f, "~", [f1]) &
    UnitParts(f1, "\\/", [f2, f3]) &            % f = ~(f2 \/ f3)
    UnitParts(notf2, "~", [f2]) &
    UnitParts(notf3, "~", [f3]) &
    ExpandAnd(notf2,notf3,leaf_ptrs,Tb(leaves,nodes,exp,max),tableau).

Apply(f, leaf_ptrs, Tb(leaves,nodes,exp,max), tableau) <-
    UnitParts(f, "~", [f1]) &
    UnitParts(f1, "->", [f2, f3]) &             % f = ~(f2 -> f3)
    UnitParts(notf3, "~", [f3]) &
    ExpandAnd(f2,notf3,leaf_ptrs,Tb(leaves,nodes,exp,max),tableau).

Apply(f, leaf_ptrs, Tb(leaves,nodes,exp,max), tableau) <-
    UnitParts(f, "~", [f1]) &
    UnitParts(f1, "~", [f2]) &                  % f = ~~f2
    ExpandNegNeg(f2,leaf_ptrs,Tb(leaves,nodes,exp,max),tableau).


PREDICATE  ExpandOr :

  Unit            % A propositional formula.
* Unit            % A propositional formula.
* Set(Integer)    % A set of pointers to open leaf nodes.
* Tableau         % A tableau containing these open leaf nodes.
* Tableau.        % The tableau obtained by extending this tableau at each of
                  % these open leaf nodes with two "or" children corresponding
                  % to each of these formulas.

ExpandOr(f1,f2,leaf_ptrs,Tb(leaves,nodes,exp,max),new_tableau) <-
    IF SOME [n]  n In leaf_ptrs
    THEN
      Lf(n,anc) In leaves &
      max1 = max + 1 &
      max2 = max + 2 &
      (IF Literal(f1) THEN e1 = {} ELSE e1 = {max1}) &
      (IF Literal(f2) THEN e2 = {} ELSE e2 = {max2}) &
      (IF Closed(f1,anc,nodes) THEN lf1 = {} ELSE lf1 = {Lf(max1,anc+{max1})}) &
      (IF Closed(f2,anc,nodes) THEN lf2 = {} ELSE lf2 = {Lf(max2,anc+{max2})}) &
      ExpandOr(f1, f2, leaf_ptrs\{n},
         Tb((leaves\{Lf(n,anc)})+lf1+lf2, nodes+{Nd(max1,f1),Nd(max2,f2)},
            exp+e1+e2,max2),
         new_tableau)
    ELSE
      new_tableau = Tb(leaves,nodes,exp,max).


PREDICATE  ExpandAnd :

  Unit           % A propositional formula.
* Unit           % A propositional formula.
* Set(Integer)   % A set of pointers to open leaf nodes.
* Tableau        % A tableau containing these open leaf nodes.
* Tableau.       % The tableau obtained by extending this tableau at each of
                 % these open leaf nodes with two "and" children corresponding
                 % to each of these formulas.

ExpandAnd(f1,f2,leaf_ptrs,Tb(leaves,nodes,exp,max),new_tableau) <-
    IF SOME [n]  n In leaf_ptrs
    THEN
      Lf(n,anc) In leaves &
      max1 = max + 1 &
      max2 = max + 2 &
      (IF Literal(f1) THEN e1 = {} ELSE e1 = {max1}) &
      (IF Literal(f2) THEN e2 = {} ELSE e2 = {max2}) &
      (IF (Closed(f1,anc,nodes) \/ Closed(f2,anc+{max1},nodes+{Nd(max1,f1)}))
       THEN lf2 = {} ELSE lf2 = {Lf(max2,anc+{max1,max2})}) &
      ExpandAnd(f1, f2, leaf_ptrs\{n},
         Tb((leaves\{Lf(n,anc)})+lf2, nodes+{Nd(max1,f1),Nd(max2,f2)},
            exp+e1+e2,max2),
         new_tableau)
    ELSE
      new_tableau = Tb(leaves,nodes,exp,max).


PREDICATE  ExpandNegNeg :
  Unit            % A propositional formula.
* Set(Integer)    % A set of pointers to open leaf nodes.
* Tableau         % A tableau containing these open leaf nodes.
* Tableau.        % The tableau obtained by extending this tableau at each of
                  % these open leaf nodes with a node containing this formula.

ExpandNegNeg(f1,leaf_ptrs,Tb(leaves,nodes,exp,max),new_tableau) <-
    IF SOME [n]  n In leaf_ptrs
    THEN
      Lf(n,anc) In leaves &
      max1 = max + 1 &
      (IF Literal(f1) THEN e1 = {} ELSE e1 = {max1}) &
      (IF Closed(f1,anc,nodes) THEN lf1 = {} ELSE lf1 = {Lf(max1,anc+{max1})}) &
      ExpandNegNeg(f1, leaf_ptrs\{n},
         Tb((leaves\{Lf(n,anc)}) + lf1, nodes+{Nd(max1,f1)}, exp+e1, max1),
         new_tableau)
    ELSE
      new_tableau = Tb(leaves,nodes,exp,max).
```

10.4 Vanilla

The module `Vanilla` below contains a `solve` interpreter written using a non-ground representation. For this example, the object program is the program consisting of the module `M2` (with some added control on `Append`).

We adopt the convention that an object level symbol is represented by a meta-level symbol with a name that consists of the name of the object level symbol prefixed by an `O` (for object). In the non-ground representation, parameters at the object level are represented by parameters at the meta-level, bases by bases, constructors by constructors, variables by variables, constants by constants, functions by functions, propositions by constants, predicates by functions, and connectives by functions. The type `OFormula` is used for the type of meta-level terms representing object-level formulas. The connective `&` is represented by the function `And`, the connective `~` is represented by the function `Not`, and the connective `<-` is represented by the function `If`. Note that object level predicates are represented by meta-level *functions*, so that, for example, the predicate `Append` is represented by the function `OAppend` with declaration

```
FUNCTION      OAppend : OList(ODay) * OList(ODay) * OList(ODay) -> OFormula.
```

Statements in the object program `M2` are represented in `OM2` using the predicate `Statement` and the constant `Empty`, which is used to represent the empty body.

Note how control at the object level can be pushed to the meta-level. In `M2`, we assume that there is a `DELAY` declaration for `Append` as follows.

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

This `DELAY` declaration is represented by the `DELAY` declaration for `Statement` given in module `OM2`. Then a goal of the form

`<- Solve(OAppend3(`r',s',t'`)).`

where r', s', and t' are the representations of r, s, and t, respectively, will run with the same control as the object level goal

`<- Append3(`r,s,t`).`

It is straightforward to extend the non-ground representation of programs consisting of one module to programs consisting of a number of modules. Note, however, that a programmer has to provide the non-ground representation of object level modules, as there is no utility provided to do this. Also there are limitations on what can be represented. For example, it is not possible to form the non-ground representation of a module containing a polymorphic predicate (for example, `Append` in module `M3`). The reason is that the function which represents such a polymorphic predicate has a non-transparent language declaration. Furthermore, there is no way in the non-ground representation to represent quantifiers.

The program {`Vanilla, OM2`} shows that it is possible to use Gödel to illustrate the basics of the non-ground representation and to write the vanilla interpreter and simple extensions of it, as given in standard Prolog textbooks. However, as we pointed out above, there are restrictions on what can be represented and, in any case, it is not possible to do serious meta-programming in a declarative way with the non-ground representation. These are the main reasons why Gödel does not provide any special support for the non-ground representation, but, instead, provides considerable support for the ground representation.

```
MODULE         Vanilla.

IMPORT         OM2.

PREDICATE      Solve : OFormula.
DELAY          Solve(x) UNTIL NONVAR(x).

Solve(Empty).

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

Solve(Not x) <-
               ~ Solve(x).

Solve(x) <-
               Statement(x If y) &
               Solve(y).
```

```
EXPORT        OM2.

BASE          OFormula, ODay.
CONSTRUCTOR   OList/1.

CONSTANT      Empty : OFormula;
              ONil : OList(ODay);
              OMonday, OTuesday, OWednesday, OThursday, OFriday, OSaturday,
              OSunday : ODay.

FUNCTION      And : xFy(110) : OFormula * OFormula -> OFormula;
              Not : Fy(120) : OFormula -> OFormula;
              If : xFx(100) : OFormula * OFormula -> OFormula;
              OCons : ODay * OList(ODay) -> OList(ODay);
              OAppend : OList(ODay) * OList(ODay) * OList(ODay) -> OFormula;
              OAppend3 : OList(ODay) * OList(ODay) * OList(ODay) * OList(ODay)
                                                                   -> OFormula.

PREDICATE     Statement : OFormula.

DELAY         Statement(OAppend(x,_,z) If _) UNTIL NONVAR(x) \/ NONVAR(z).
```

```
LOCAL         OM2.

Statement(OAppend(ONil,x,x) If Empty).
Statement(OAppend(OCons(u,x),y,OCons(u,z)) If OAppend(x,y,z)).
Statement(OAppend3(x,y,z,u) If OAppend(x,y,w) And OAppend(w,z,u)).
```

10.5 ThreeWisemen

The next program implements a solution to the three wisemen puzzle. The meta-level architecture chosen is a simple one that captures the reasoning of the third wiseman in a way which is semantically defensible. However, the main purpose of the program is not to make a contribution to the study of meta-reasoning, but rather to illustrate how dynamic meta-programming can be done in a declarative way.

```
MODULE ThreeWisemen.

% The Three Wisemen puzzle is as follows:
% A king, wishing to find out which of his three wisemen is the wisest, puts
% a hat on each of their heads, and tells them that each hat is either black
% or white and at least one of the hats is white. The king does this in such a
% way that each wiseman can see the hats of the other wisemen, but not his own.
% In fact, each wiseman has a white hat on. The king then successively asks
% each one of wisemen whether he knows the colour of his own hat. The first
% wiseman answers "I don't know", as does the second. Then the third announces
% that his hat is white.
%
% The reasoning of the third wiseman is as follows:
% "Suppose my hat is black. Then the second wiseman would see a black hat and
% a white hat, and would reason that, if his hat is black, the first wiseman
% would see two black hats and hence would conclude that his hat is white on
% the basis of the king's assurance that at least one of the hats is white.
% But the second wiseman said he didn't know the colour of his hat. Hence my
% hat must be white."
%
% This program directly simulates the reasoning of the third wiseman. The
% module Wiseman3KB contains the knowledge of the third wiseman. The module
% Reasoner contains two reasoning procedures in the form of the two statements
% for the predicate Solve. The first statement embodies the principle: "if I
% can see the other two hats are black, then mine must be white". The second
% embodies a proof by contradiction: "if I assume my hat is a particular colour
% and thereby derive a contradiction of the fact that one of the other wisemen
% didn't know the colour of his hat, then my hat must be the other colour".
%
% The module Wiseman3KB needs to be program-compiled first.
%
% Query:  <- Go(colour).

IMPORT ProgramsIO, Reasoner.

PREDICATE  Go : Colour.

Go(colour) <-
  FindInput("Wiseman3KB.prm", In(stream)) &
  GetProgram(stream, wiseman3_kb) &
  EndInput(stream) &
  Solve(W3,wiseman3_kb, colour).
```

```
EXPORT       Reasoner.

% This module contains the two reasoning procedures for the Three Wisemen
% puzzle.

IMPORT       Programs.

BASE         Wiseman, Colour.

CONSTANT     W1, W2, W3 : Wiseman;
             Black, White : Colour.


PREDICATE  Solve :

  Wiseman          % A wiseman.
* Program          % The knowledge known, or assumed to be known, to this
                   % wiseman by the third wiseman, W3.
* Colour.          % The colour of this wiseman's hat.
```

```
LOCAL        Reasoner.

IMPORT       Sets.


% PREDICATE    Solve : Wiseman * Program * Colour.

Solve(wiseman, kb, White) <-
      {Enumerate(wiseman, next_wiseman, last_wiseman)} &
      FormBodyString1(wiseman, next_wiseman, Black, body2_string) &
      FormBodyString1(wiseman, last_wiseman, Black, body3_string) &
      Prove(kb, body2_string) &
      Prove(kb, body3_string).

Solve(wiseman, kb, colour) <-
      Enumerate(wiseman, next_wiseman, last_wiseman) &
      FormBodyString2(wiseman, next_wiseman, body_string) &
      Prove(kb, body_string)  &
      AssumeHatColour(wiseman, next_wiseman, Black, kb, kb2) &
      AssumeHatColour(wiseman, last_wiseman, Black, kb2, new_kb) &
      IF Solve(next_wiseman, new_kb, _)
      THEN colour = White
      ELSE
         AssumeHatColour(wiseman, next_wiseman, White, kb, kb3) &
         AssumeHatColour(wiseman, last_wiseman, White, kb3, newer_kb) &
         Solve(next_wiseman, newer_kb, _) &
         colour = Black.


PREDICATE    Enumerate :

  Wiseman           % A wiseman.
* Wiseman           % A wiseman different from the previous one.
* Wiseman.          % The remaining wiseman.

Enumerate(x, y, z) <-
      x In {W1, W2, W3} &
      y In {W1, W2, W3}\{x} &
      z In {W1, W2, W3}\{x,y}.


PREDICATE    AssumeHatColour :

  Wiseman           % A wiseman.
* Wiseman           % Another wiseman.
* Colour            % A colour.
* Program           % A knowledge base.
* Program.          % The knowledge base obtained from this knowledge base by
                    % adding the fact that the second wiseman knows that this
                    % colour is the colour of the first wiseman's hat.

AssumeHatColour(wiseman, other_wiseman, colour, kb, new_kb) <-
      MainModuleInProgram(kb,module) &
      FormBodyString1(other_wiseman, wiseman, colour, string) &
      StringToProgramFormula(kb, module, string, [atom]) &
      EmptyFormula(empty_formula) &
      IsImpliedBy(atom, empty_formula, fact) &
      InsertStatement(kb, module, fact, new_kb).


PREDICATE    Prove :

  Program           % A knowledge base.
* String.           % The string representation of the body of a goal which
                    % succeeds for this knowledge base.
```

```
Prove(kb,body_string) <-
      MainModuleInProgram(kb,module) &
      StringToProgramFormula(kb, module, body_string, [body]) &
      Succeed(kb,body,_).


PREDICATE    FormBodyString1 :

  Wiseman           % A wiseman.
* Wiseman           % Another wiseman.
* Colour            % A colour.
* String.           % The string representation of the body of the fact that
                    % the first wiseman knows that this is the colour of the
                    % hat of the second wiseman.

FormBodyString1(W1, W2, White, "Knows(W1, Hat(W2, White))").
FormBodyString1(W1, W3, White, "Knows(W1, Hat(W3, White))").
FormBodyString1(W2, W1, White, "Knows(W2, Hat(W1, White))").
FormBodyString1(W2, W3, White, "Knows(W2, Hat(W3, White))").
FormBodyString1(W3, W1, White, "Knows(W3, Hat(W1, White))").
FormBodyString1(W3, W2, White, "Knows(W3, Hat(W2, White))").
FormBodyString1(W1, W2, Black, "Knows(W1, Hat(W2, Black))").
FormBodyString1(W1, W3, Black, "Knows(W1, Hat(W3, Black))").
FormBodyString1(W2, W1, Black, "Knows(W2, Hat(W1, Black))").
FormBodyString1(W2, W3, Black, "Knows(W2, Hat(W3, Black))").
FormBodyString1(W3, W1, Black, "Knows(W3, Hat(W1, Black))").
FormBodyString1(W3, W2, Black, "Knows(W3, Hat(W2, Black))").


PREDICATE    FormBodyString2 :

  Wiseman           % A wiseman.
* Wiseman           % Another wiseman.
* String.           % The string representation of the body of the fact that
                    % the first wiseman knows that the second wiseman doesn't
                    % know the colour of his own hat.

FormBodyString2(W1, W2, "Knows(W1, DoesntKnow(W2))").
FormBodyString2(W1, W3, "Knows(W1, DoesntKnow(W3))").
FormBodyString2(W2, W1, "Knows(W2, DoesntKnow(W1))").
FormBodyString2(W2, W3, "Knows(W2, DoesntKnow(W3))").
FormBodyString2(W3, W1, "Knows(W3, DoesntKnow(W1))").
FormBodyString2(W3, W2, "Knows(W3, DoesntKnow(W2))").
```

```
MODULE       Wiseman3KB.

% The knowledge base of the third wiseman, W3.

BASE         Wiseman, Colour, Fact.

CONSTANT     W1, W2, W3 : Wiseman;
             Black, White : Colour.

FUNCTION     DoesntKnow : Wiseman -> Fact;
             Hat : Wiseman * Colour -> Fact.


PREDICATE    Knows :

  Wiseman          % A wiseman.
* Fact.            % Some knowledge which W3 knows to be known to this wiseman.

Knows(W3, DoesntKnow(W1)).
Knows(W3, DoesntKnow(W2)).
Knows(W3, Hat(W1,White)).
Knows(W3, Hat(W2,White)).
Knows(W2, DoesntKnow(W1)).
Knows(W2, Hat(W1,White)).
Knows(W1, Hat(W2,White)).
```

10.6 Interpreter

The module `Interpreter` below shows how a programmer might make use of the predicates provided by the module `Programs`. `Interpreter` contains the definitions of the predicate `MySucceed`, which is intended to be true when its first argument is the representation of a normal program (without commits or conditionals), its second argument is the representation of the body of a normal goal to this program, and its third argument is the representation of a computed answer for this goal and program using SLDNF-resolution. The predicate `MyFail` is intended to be true when its first argument is the representation of a normal program, and its second argument is the representation of the body of a normal goal in the flat language of the program which has a finitely failed SLDNF-tree wrt this program. The predicate `Select` implements the safe "leftmost literal" computation rule. `Select` is intended to be true when its first argument represents a non-empty conjunction of literals Q in this language, its second argument represents the conjunction of literals of Q which is the largest prefix of Q containing only non-ground negative literals, its third argument is of the form `Pos(`E`)` or `Neg(`E`)`, where E represents the first atom or ground negative literal appearing in Q (and where `Pos` indicates that it is a positive literal and `Neg` indicates that it is a ground negative literal), and its fourth argument represents the conjunction of literals in Q to the right E.

To run the interpreter, a goal such as

```
<- Go1("M4", "Append([Monday, Tuesday], [Wednesday], x)", answer).
```

could be used. This would produce the answer

```
answer = "{x/[Monday,Tuesday,Wednesday]}".
```

Similarly, a goal such as

```
<- Go2("M4", "Append([Monday, Tuesday], [Wednesday], [])").
```

gives the answer `Yes`.

Note the use `Interpreter` makes of the system predicates `Succeed` and `Fail` in `Programs` for running literals whose proposition or predicate is not defined in an open module. The code in `Interpreter` could be used as the basis for an interpreter which required a more sophisticated computation rule using coroutining, for example.

Not surprisingly, `Interpreter` runs rather slowly (in fact, around 2 orders of magnitude slower than running the object program directly). However, it is important to appreciate that, by rewriting parts of the interpreter and employing partial evaluation techniques, it is possible to significantly improve its speed. A similar approach can be applied to other meta-programs. The details of this may be found in [8] and [9].

```
MODULE   TestInterpreter.

IMPORT   Interpreter, ProgramsIO, Answers.


PREDICATE Go1 : String * String * String.

Go1(prog_string, goal_string, answer_string) <-
    FindInput(prog_string ++ ".prm", In(stream)) &
    GetProgram(stream, program) &
    MainModuleInProgram(program, module) &
    StringToProgramFormula(program, module, goal_string, [goal]) &
    NormalBody(goal) &
    CommitFreeFormula(goal) &
    MySucceed(program, goal, answer) &
    RestrictSubstToFormula(goal, answer, computed_answer) &
    AnswerString(program, module, computed_answer, answer_string).


PREDICATE Go2 : String * String.

Go2(prog_string, goal_string) <-
    FindInput(prog_string ++ ".prm", In(stream)) &
    GetProgram(stream, program) &
    MainModuleInProgram(program, module) &
    StringToProgramFormula(program, module, goal_string, [goal]) &
    NormalBody(goal) &
    CommitFreeFormula(goal) &
    MyFail(program, goal).
```

```
EXPORT     Answers.

IMPORT     Programs.

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

```
LOCAL      Answers.

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

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

PREDICATE  AnswerString1 : Program * String * TermSubst * String.

AnswerString1(program, module, answer, string) <-
    {IF SOME [var, term, answer1]
          DelBindingInTermSubst(answer, var, term, answer1)
     THEN
       ProgramTermToString(program, module, var, var_string) &
       ProgramTermToString(program, module, term, term_string) &
       AnswerString2(program, module, answer1, var_string ++ "/" ++ term_string,
                                                                   string)
     ELSE
       string = ""
    }.

PREDICATE  AnswerString2 : Program * String * TermSubst * String * String.

AnswerString2(program, module, answer, so_far, string) <-
    {IF SOME [var, term, answer1]
          DelBindingInTermSubst(answer, var, term, answer1)
     THEN
       ProgramTermToString(program, module, var, var_string) &
       ProgramTermToString(program, module, term, term_string) &
       AnswerString2(program, module, answer1,
                     so_far ++ "," ++ var_string ++ "/" ++ term_string, string)
     ELSE
       string = so_far
    }.
```

```
EXPORT     Interpreter.

% This module implements an interpreter for SLDNF-resolution using the ground
% representation. The computation rule employed is the safe "leftmost literal"
% one, that is, the selected literal is the leftmost literal which is either an
% atom or a ground negative literal. This module assumes the existence of one
% above it which handles the conversion of goals and answers between their
% string representation and their ground representation.

IMPORT     Programs.

PREDICATE  MySucceed :

  Program       % Representation of a normal program (excluding commits and
                % conditionals).
* Formula       % Representation of the body of a normal goal to this program.
* TermSubst.    % Representation of a computed answer for this goal and
                % program using SLDNF-resolution and the safe leftmost literal
                % computation rule.

DELAY      MySucceed(x,y,_) UNTIL GROUND(x) & GROUND(y).


PREDICATE  MyFail :

  Program       % Representation of a normal program (excluding commits and
                % conditionals).
* Formula.      % Representation of the body of a normal goal to this program
                % such that this goal and this program have a finitely failed
                % SLDNF-tree using the safe leftmost literal computation rule.

DELAY      MyFail(x,y) UNTIL GROUND(x) & GROUND(y).
```

```
LOCAL      Interpreter.

BASE       SignedFormula.

FUNCTION   Pos, Neg : Formula  -> SignedFormula.


% PREDICATE   MySucceed : Program  * Formula * TermSubst.

MySucceed(program, body, computed_answer) <-
    EmptyTermSubst(empty_subst) &
    IF EmptyFormula(body)
    THEN
      computed_answer = empty_subst
    ELSE
      Select(body, left, selected, right) &
      MySucceed1(program, body, left, selected, right, empty_subst, answer) &
      RestrictSubstToFormula(body, answer, computed_answer).



PREDICATE  MySucceed1 :

  Program         % Representation of a normal program.
* Formula         % Representation of the head of a resultant.
* Formula         % Representation of the body of the resultant to the left of
                  % the selected literal.
* SignedFormula   % Representation of the selected literal in the body.
* Formula         % Representation of the body of the resultant to the right of
                  % the selected literal.
* TermSubst       % Representation of a term substitution.
* TermSubst.      % Representation of the term substitution obtained by
                  % composing the term substitution in the sixth argument
                  % with the computed answer obtained for this program and
                  % resultant.


MySucceed1(program, head, left, Pos(selected), right, answer_so_far, answer) <-
    (IF SOME [module] DeclaredInOpenModule(program, module, selected)
     THEN
       StatementMatchAtom(program, module, selected, stat) &
       RenameFormulas([head,left,selected,right], [stat], [new_stat]) &
       Derive(head, left, selected, right, new_stat, mgu, new_resultant) &
       ComposeTermSubsts(answer_so_far, mgu, new_answer) &
       IsImpliedBy(new_head, new_body, new_resultant)
     ELSE
       FormulaMaxVarIndex([head,left,selected,right],varindex) &
       EmptyTermSubst(empty) &
       Compute(program, selected, varindex, _, empty, subst, last_goal) &
       EmptyFormula(last_goal) &
       ApplySubstToFormula(head, subst, new_head) &
       AndWithEmpty(left, right, body1) &
       ApplySubstToFormula(body1, subst, new_body) &
       ComposeTermSubsts(answer_so_far, subst, new_answer)
    ) &
    IF EmptyFormula(new_body)
    THEN
      answer = new_answer
    ELSE
      Select(new_body, new_left, new_selected, new_right) &
      MySucceed1(program, new_head, new_left, new_selected, new_right,
                                                         new_answer, answer).

MySucceed1(program, head, left, Neg(selected), right, answer_so_far, answer) <-
    (IF DeclaredInOpenModule(program, _, selected)
     THEN
       EmptyFormula(empty_formula) &
       MyFail1(program, empty_formula, Pos(selected), empty_formula)
     ELSE
       Fail(program, selected)
    ) &
    AndWithEmpty(left, right, new_body) &
    IF EmptyFormula(new_body)
    THEN
      answer = answer_so_far
    ELSE
      Select(new_body, new_left, new_selected, new_right) &
      MySucceed1(program, head, new_left, new_selected, new_right,
                                                      answer_so_far, answer).


% PREDICATE   MyFail : Program  * Formula.

MyFail(program, body) <-
    Select(body, left, selected, right) &
    MyFail1(program, left, selected, right).


PREDICATE  MyFail1 :
  Program         % Representation of a normal program for which there exists a
                  % finitely failed SLDNF-tree for a goal, using the safe
                  % "leftmost literal" computation rule.
* Formula         % Representation of the body of the goal to the left of the
                  % selected literal.
* SignedFormula   % Representation of the selected literal in the body of the
                  % goal.
* Formula.       % Representation of the body of the goal to the right of the
                  % selected literal.

MyFail1(program, left, Pos(selected), right) <-
    EmptyFormula(empty_formula) &
    IF SOME [module] DeclaredInOpenModule(program, module, selected)
    THEN
      ALL [new_body]
        (SOME [l, s, r] (Select(new_body, l, s, r) & MyFail1(program, l, s, r))
         <-
            SOME [stat, new_stat, new_goal]
            (StatementMatchAtom(program, module, selected, stat) &
             RenameFormulas([left,selected,right], [stat], [new_stat]) &
             Derive(empty_formula, left, selected, right, new_stat, _, new_goal) &
             IsImpliedBy(empty_formula, new_body, new_goal)
            )
        )
    ELSE
      IF Fail(program, selected)
      THEN
        True
      ELSE
        FormulaMaxVarIndex([left,selected,right],varindex) &
        EmptyTermSubst(empty) &
        Compute(program, selected, varindex, _, empty, subst, last_goal) &
        EmptyFormula(last_goal) &
        AndWithEmpty(left, right, body1) &
        ApplySubstToFormula(body1, subst, new_body) &
        Select(new_body, new_left, new_selected, new_right) &
        MyFail1(program, new_left, new_selected, new_right).


MyFail1(program, _, Neg(selected), _) <-
    EmptyFormula(empty_formula) &
    EmptyTermSubst(empty_subst) &
    MySucceed1(program, selected, empty_formula, Pos(selected), empty_formula,
                                                               empty_subst, _) |.
```

```
MyFail1(program, left, Neg(selected), right) <-
    EmptyFormula(empty_formula) &
    MyFail1(program, empty_formula, Pos(selected), empty_formula) |
    AndWithEmpty(left, right, new_body) &
    Select(new_body, new_left, new_selected, new_right) &
    MyFail1(program, new_left, new_selected, new_right).


PREDICATE  Select :

  Formula          % Representation of a (non-empty) conjunction of literals.
* Formula          % Representation of the conjunction of literals to the left
                   % of the selected literal.
* SignedFormula    % Term with function Pos or Neg at the top-level and
                   % containing the representation of the atom in the selected
                   % literal (according to the safe "leftmost literal"
                   % computation rule and where Pos is used if the selected
                   % literal is an atom and Neg otherwise).
* Formula.         % Representation of the conjunction of literals to the right
                   % of the selected literal.

Select(atom, empty_formula, Pos(atom), empty_formula) <-
    Atom(atom) &
    EmptyFormula(empty_formula).

Select(negative_literal, empty_formula, Neg(atom), empty_formula) <-
    Not(atom, negative_literal) &
    GroundAtom(atom) &
    EmptyFormula(empty_formula).

Select(body, left, selected, right) <-
    And(l, r, body) &
    IF SOME [l1, s1, r1] Select(l, l1, s1, r1)
    THEN
      left = l1 &
      selected = s1 &
      AndWithEmpty(r1, r, right)
    ELSE
      Select(r, l1, selected, right) &
      AndWithEmpty(l, l1, left).
```

Part II

Definition of Gödel

Chapter 11

Syntax

In this chapter, we present the syntax of Gödel. Our main objective is to define precisely the concepts of program, goal, theory, and flock.

11.1 Notation

We first describe the notation in which the grammar is defined. A grammar rule takes the following form:

non-terminal $\longrightarrow$ sequence of terminals and non-terminals

There may be more than one way of decomposing a non-terminal, in which case the alternatives are separated by $|$. A terminal is distinguished from a non-terminal by being placed within angle-brackets $< >$. Some non-terminals have arguments: a lower case letter indicates that the argument is a variable. A sequence X of terminals and non-terminals may be enclosed in brackets:

$\{X\}$ means 0 or more occurrences of X.

$\{X\}^{(n)}$ means exactly n occurrences of X.

$[X]$ means 0 or 1 occurrence of X.

For some rules, extra *conditions* determine whether a particular rule can be applied. These conditions are given in italic text following the rule to which they apply.

11.2 Tokens

A program, theory, and so on, can be regarded as text consisting of a string of characters. This string is parsed and substrings of characters are identified as *tokens*. Thus the text is first transformed to a sequence of these tokens. It is this sequence of tokens and not the original text that will be used in the grammars given in subsequent sections. The text includes special sequences of characters, called *layout items*, which improve the layout and readability of the text. In addition, these layout items are used, where necessary, to indicate the separation of tokens. As a guide to the use of layout items, note that if a sequence of characters would form a token but the sequence is included in a larger sequence which would also form a token, then the larger token is preferred. The layout items are not tokens, but so that they can be included in the text, we define the grammar rules for them as well.

The terminals of this grammar are the individual characters of the text. Note that the token Identifier appears only in the grammar for flocks.

Token ⟶ BigName | LittleName | GraphicName | String | Bracket | Comma
| Semicolon | Underscore | Terminator | Number | Float | Identifier

BigName ⟶ BigLetter {NameCharacter}

LittleName ⟶ LittleLetter {NameCharacter}

GraphicName ⟶ GraphicCharacter {GraphicCharacter}

String ⟶ <"> {StringCharacter} <">

Bracket ⟶ <{> | <}> [<_> Label] | <(> | <)> | <[> | <]>

Comma ⟶ <,>

Semicolon ⟶ <;>

Underscore ⟶ <_>

Terminator ⟶ <.>

Number ⟶ Zero {Zero} | PositiveNumber

Float ⟶ Decimal
| Decimal <E> <+> Number
| Decimal <E> <-> Number

Decimal ⟶ Number <.> Number

Label ⟶ PositiveNumber

NameCharacter ⟶ BigLetter | LittleLetter | Digit | Underscore

GraphicCharacter ⟶ <\> | NonBSGraphicChar

NonBSGraphicChar ⟶ <+> | <-> | <*> |</> | <^> | <#> | <<> | <>> | <=> | <~>
| <&> | <?> | <‘> | <@> | <!> | <$> | <:> | <|> | <’>

BigLetter ⟶ <A> | <B> | <C> | <D> | <E> | <F> | <G> | <H> | <I>

| <J> | <K> | <L> | <M> | <N> | <O> | <P> | <Q> | <R>
| <S> | <T> | <U> | <V> | <W> | <X> | <Y> | <Z>

LittleLetter $\longrightarrow$ <a> | <b> | <c> | <d> | <e> | <f> | <g> | <h> | <i>
| <j> | <k> | <l> | <m> | <n> | <o> | <p> | <q> | <r>
| <s> | <t> | <u> | <v> | <w> | <x> | <y> | <z>

PositiveNumber $\longrightarrow$ {Zero} NonZero {Digit}

Digit $\longrightarrow$ Zero | NonZero

NonZero $\longrightarrow$ <1> | <2> | <3> | <4> | <5> | <6> | <7> | <8> | <9>

Zero $\longrightarrow$ <0>

StringCharacter $\longrightarrow$ <\> <\> | <\> <"> | [<\>] NonEscCharacter

LayoutItem $\longrightarrow$ LayoutCharacter | Comment

LayoutCharacter $\longrightarrow$ < *space* > | < *tab* > | < *newline* >

Comment $\longrightarrow$ <%> {Character}

Condition: The sequence {Character} *does not contain a newline, but is followed by a newline.*

Character $\longrightarrow$ <\> | <"> | NonEscCharacter

NonEscCharacter $\longrightarrow$ LayoutCharacter | NonBSGraphicChar | NameCharacter
| Semicolon | Comma | Terminator | <%> | <{> | <}>
| <(> | <)> | <[> | <]>

Identifier $\longrightarrow$ QuotedIdent | DbleQuotedIdent | OrdinaryIdent

QuotedIdent $\longrightarrow$ <'> {QuoteChar} <'>

QuoteChar $\longrightarrow$ <'><'> | Character

Condition: Character *may not be* '.

DbleQuotedIdent $\longrightarrow$ <"> {DbleQuoteChar} <">

DbleQuoteChar $\longrightarrow$ <"><"> | Character

Condition: Character *may not be* ".

OrdinaryIdent ⟶ OrdinaryChar {OrdinaryChar}

OrdinaryChar ⟶ NameCharacter | GraphicCharacter | Semicolon | Terminator
| <`[`> | <`]`> | <`{`> | <`}`>

Condition: GraphicCharacter *may not be* `'`.

11.3 Programs

The terminals of the grammars in this and subsequent sections are the tokens defined previously. For some rules it is necessary to know, not just the name of the token, but also the internal structure of the token. In this case, the internal structure is given in place of the token name. For example: if, in a rule, the token 'Number' has to be a positive number, then this is denoted by <PositiveNumber>. If the token has to include a particular string of terminals from the previous grammar, then this string is enclosed in single quotes. For example: <'`BASE`'> indicates that the token is a BigName consisting of the string '`BASE`'. Similarly, <'`}`' ['`_`' Label]> indicates that the token is a Bracket consisting of '`}`' followed by an optional '`_`' Label. The token name is easily inferred from the given structure.

Program ⟶ Module {Module}

Module ⟶ ExportPart LocalPart
| ExportPart
| LocalPart

Condition: If a module has a LocalPart *and an* ExportPart*, then the* ModuleName *of each part must be the same.*

ExportPart ⟶ ExportKind ModuleName <Terminator> {ExportItem}

LocalPart ⟶ LocalKind ModuleName <Terminator> {LocalItem}

Condition: The LocalPart *of a module has the* LocalKind `MODULE` *if and only if the module has no* ExportPart*.*

ExportKind ⟶ <'`EXPORT`'> | <'`CLOSED`'>

Condition: Only system modules can have ExportKind `CLOSED`*.*

LocalKind ⟶ <'`LOCAL`'> | <'`MODULE`'>

ModuleName ⟶ UserBigName

Condition: ModuleName *for a user-defined module cannot be any of* `Integers`*,* `Rationals`*,* `Floats`*,* `Numbers`*,* `Lists`*,* `Sets`*,* `Strings`*,* `Tables`*,* `Units`*,* `Flocks`*,* `Syntax`*,* `Programs`*,* `Scripts`*,* `Theories`*,* `IO`*,* `NumbersIO`*,* `FlocksIO`*,* `ProgramsIO`*,* `ScriptsIO`*, or* `TheoriesIO`*.*

```
ExportItem        ⟶   ImportDecl <Terminator>
                     | LanguageDecl <Terminator>
                     | ControlDecl <Terminator>

LocalItem         ⟶   ImportDecl <Terminator>
                     | LiftDecl <Terminator>
                     | LanguageDecl <Terminator>
                     | ControlDecl <Terminator>
                     | Statement <Terminator>

ImportDecl        ⟶   <'IMPORT'> ModuleName {<Comma> ModuleName}

LiftDecl          ⟶   <'LIFT'> ModuleName {<Comma> ModuleName}
```

Condition: A `LIFT` *declaration can only appear in the local part of a module which has an export part.*

At this stage, the overall module structure of a (potential) program can be discerned. To explain this, we introduce the following definitions.

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

```
IMPORT    N.
```

or

```
LIFT      N.
```

A module `M` *refers to* a module `N` if either the local or export part of `M` refers to `N`.

The relation *depends upon* between modules is defined to be the transitive closure of the relation refers to.

Then the (potential) program must consist of a set $\{\texttt{Mi}\}_{\texttt{i}=0}^{\texttt{n}}$ of modules with the property that $\{\texttt{Mi}\}_{\texttt{i}=1}^{\texttt{n}}$ is the set of modules upon which the distinguished module `M0` depends. We say `M0` is the *main module* of the program.

Furthermore, the following module condition can now be checked.

M1: No module may depend upon itself.

Language Declarations

```
LanguageDecl      ⟶   BaseDecl
                     | ConstructorDecl
                     | ConstantDecl
                     | FunctionDecl
                     | PropositionDecl
                     | PredicateDecl
```

BaseDecl $\longrightarrow$ <‘`BASE`’> UserNameSeq

ConstructorDecl $\longrightarrow$ <‘`CONSTRUCTOR`’> ConstrDecl {<Comma> ConstrDecl}

ConstrDecl $\longrightarrow$ UserName <‘`/`’> <PositiveNumber>

ConstantDecl $\longrightarrow$ <‘`CONSTANT`’> ConstDecl {<Semicolon> ConstDecl}

ConstDecl $\longrightarrow$ UserNameSeq <‘`:`’> Type

Note that the declaration of integer constants and floating-point constants is treated specially by the system. For these, UserNameSeq *is a sequence of* Number*s (not* UserName*s).*

FunctionDecl $\longrightarrow$ <‘`FUNCTION`’> FuncDecl {<Semicolon> FuncDecl}

FuncDecl $\longrightarrow$ UserNameSeq
[<‘`:`’> FunctionSpec(n) <‘`(`’> <PositiveNumber> <‘`)`’>]
<‘`:`’> Type {<‘`*`’> Type}$^{(n-1)}$ <‘`->`’> Type

Condition: The declaration must be transparent*, that is, every parameter appearing in the declaration must also appear in the range type.*

FunctionSpec(1) $\longrightarrow$ <‘`Fx`’> | <‘`Fy`’> | <‘`xF`’> | <‘`yF`’>

FunctionSpec(2) $\longrightarrow$ <‘`xFx`’> | <‘`xFy`’> | <‘`yFx`’>

PropositionDecl $\longrightarrow$ <‘`PROPOSITION`’> UserNameSeq

PredicateDecl $\longrightarrow$ <‘`PREDICATE`’> PredDecl {<Semicolon> PredDecl}

PredDecl $\longrightarrow$ UserNameSeq [<‘`:`’> PredicateSpec(n)]
<‘`:`’> Type {<‘`*`’> Type}$^{(n-1)}$

PredicateSpec(1) $\longrightarrow$ <‘`Pz`’> | <‘`zP`’>

PredicateSpec(2) $\longrightarrow$ <‘`zPz`’>

UserNameSeq $\longrightarrow$ UserName {< Comma > UserName}

UserName $\longrightarrow$ UserBigName | UserGraphicName

UserBigName $\longrightarrow$ < BigName >

Condition: BigName *is not one of the reserved words:* `EXPORT`, `CLOSED`, `LOCAL`, `MODULE`, `IMPORT`, `LIFT`, `THEORY`, `BASE`, `CONSTRUCTOR`, `CONSTANT`, `FUNCTION`, `PROPOSITION`, `PREDICATE`, `DELAY`, `UNTIL`, `GROUND`, `NONVAR`, `TRUE`, `ALL`, `SOME`, `IF`, `THEN`, `ELSE`.

UserGraphicName $\longrightarrow$ < GraphicName>

Condition: GraphicName *is not one of the reserved words:* `:`, `<-`, `->`, `<->`, `&`, `~`, `\/`, `|`.

At this point, the symbols which have been declared in, or imported into, the various modules can be determined. We now give the appropriate definitions.

A *symbol* is a base, constructor, constant, function, proposition, or predicate.

A *type symbol* is a base or constructor. Other symbols are *non-type symbols*.

The local part of a module `M` *1-imports* a symbol `S` *via* a module `N` if `S` has a declaration in the export part of `N` and the local part of `M` refers to `N`.

The export part of a module `M` *1-imports* a symbol `S` *via* a module `N` if `S` has a declaration in the export part of `N` and either (i) the export part of `M` refers to `N`, or (ii) the local part of `M` refers to `N` via a `LIFT` declaration, `S` is a type symbol, and there is a declaration for `S` in the export part of `M`. In the latter case, the export part of `M` also *1-redeclares* `S`.

The local part of a module `M` *n-imports*, $n > 1$, a symbol `S` *via* a module `N` if there is a module `L` such that the export part of `L` $(n-1)$-imports `S` via `N` and the local part of `M` refers to `L`.

The export part of a module `M` *n-imports*, $n > 1$, a symbol `S` *via* a module `N` if there is a module `L` such that the export part of `L` $(n-1)$-imports `S` via `N` and either (i) the export part of `M` refers to `L`, or (ii) the local part of `M` refers to `L` via a `LIFT` declaration, `S` is a type symbol, and there is a declaration for `S` in the export part of `M`. In the latter case, the export part of `M` also *n-redeclares* `S`.

A part of a module `M` *imports* a symbol `S` *via* a module `N` if the part of `M` n-imports `S` via `N`, for some $n \geq 1$.

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

The export part of a module *redeclares* a type symbol `S` if it n-redeclares `S`, for some $n \geq 1$.

The export part of a module *declares* a type symbol if it contains a declaration for, but does not redeclare, the symbol. It *declares* a non-type symbol if it contains a declaration for the symbol.

The local 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 part of `M` imports `S` via `N` and the export part of `N` declares `S`.

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 is *accessible to* the local (resp., export) part of a module if it is either declared in, or imported into, the module (resp., export part of the module).

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

The following module conditions can now be checked.

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.

Types

Type $\longrightarrow$ Parameter
| Base
| Constructor(n) <‘(’> TypeSeq(n) <‘)’>

TypeSeq(n) $\longrightarrow$ Type {<Comma> Type}$^{(n-1)}$

Base $\longrightarrow$ UserName

Condition: A symbol with name UserName *is declared or imported as a base.*

Constructor(n) $\longrightarrow$ UserName

Condition: A symbol with name UserName *is declared or imported as a constructor of arity* n.

Parameter $\longrightarrow$ < LittleName >

The following module condition can now be checked.

M4: In a module, a 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.

Control Declarations

ControlDecl $\longrightarrow$ <‘`DELAY`’> ContDecl {<Semicolon> ContDecl}

Condition: A `DELAY` *declaration for a predicate can only appear in the module in which the predicate is declared.*

ContDecl $\longrightarrow$ Atom <‘`UNTIL`’> Cond

Condition: Atom *cannot be a proposition. No pair of* Atom*’s in the set of* `DELAY` *declarations for a predicate can have a common instance.*

Cond $\longrightarrow$ Cond1
| Cond1 <‘`&`’> AndSeq
| Cond1 <‘`\/`’> OrSeq

Cond1 $\longrightarrow$ <‘`NONVAR`’> <‘(’> Variable <‘)’>
| <‘`GROUND`’> <‘(’> Variable <‘)’>
| <‘`TRUE`’>
| <‘(’> Cond <‘)’>

AndSeq $\longrightarrow$ Cond1 | Cond1 <‘`&`’> AndSeq

OrSeq $\longrightarrow$ Cond1 | Cond1 <‘`\/`’> OrSeq

Statements

At this point, the overall module structure of the (potential) program has been determined, as have the language and control declarations that each module contains. However, before we go on to give the grammar for statements, we must pause and give some more definitions. The reason is that statements are formulas in certain polymorphic many-sorted languages and we need to define these languages first.

A module is *closed* if its export part contains a `CLOSED` declaration and *open* if it is not closed. Let `M` be a module in a (potential) program `P`.

1. The *language of* `M` *wrt* `P` is the polymorphic many-sorted language given by the language declarations of all symbols accessible to the local part of `M`.

2. The *export language of* `M` *wrt* `P` is the polymorphic many-sorted language given by the language declarations of all symbols accessible to the export part of `M`.

3. The *goal language of* `P` is the language of the main module `M0`, if `M0` is open, or the export language of `M0`, if `M0` is closed.

Statements and goals are written in a language more general in several ways than that considered in appendix A. They may contain commits. These can be ignored for the purposes of language checking. Conditionals and intensional set terms are handled by replacing each occurrence of them by the formulas which give their meaning. Any expressions involving occurrences of operators are rewritten into the standard logical syntax. After this preprocessing, statements and goals are now in a suitable form for checking to see if they are in the appropriate language.

Statement $\longrightarrow$ Atom [<‘`<-`’> Body]

Condition: A statement (ignoring any commits) must be a formula in the language of the module in which it occurs.
A statement must satisfy the head condition, *that is, the tuple of types of the arguments of the head in the statement must be a variant of the type declared for the predicate in the head.*

Body $\longrightarrow$ [CFormula(f)] <‘`|`’> [CFormula(f1)]
| CFormula(f)

CFormula(0) $\longrightarrow$ <‘`(`’> CFormula(f) <‘`)`’>
| <‘`{`’> CFormula(f) <‘`}`’ [‘`_`’ Label]>
CFormula(2) $\longrightarrow$ CFormula(f) <‘`&`’> CFormula(f1)

Condition: $f \leq 1$, $f1 \leq 2$.

CFormula(f) $\longrightarrow$ Formula(f)

Formula(0) $\longrightarrow$ Atom
| RangeFormula
| <‘(’> Formula(f) <‘)’>
Formula(1) $\longrightarrow$ <‘~’> Formula(f)
| <‘SOME’> <‘[’> VariableSeq <‘]’> Formula(f)
| <‘ALL’> <‘[’> VariableSeq <‘]’> Formula(f)

Condition: $f \leq 1$.

Formula(2) $\longrightarrow$ Formula(f) <‘&’> Formula(f1)
| <‘IF’> [<‘SOME’> <‘[’> VariableSeq <‘]’>] Formula(f2)
<‘THEN’> ThenPart [<‘ELSE’> Formula(f1)]

Condition: $f \leq 1$, $f1 \leq 2$.

Formula(3) $\longrightarrow$ Formula(f) <‘\/’> Formula(f1)

Condition: $f \leq 2$, $f1 \leq 3$.

Formula(4) $\longrightarrow$ Formula(f) <‘<-’> Formula(f1)
| Formula(f) <‘->’> Formula(f1)
| Formula(f) <‘<->’> Formula(f)

Condition: $f \leq 3$, $f1 \leq 4$.

ThenPart $\longrightarrow$ Formula(f)
| Formula(f) <‘&’> ThenPart
| <‘IF’> Formula(f1) <‘THEN’> ThenPart

Condition: $f \leq 1$.

Atom $\longrightarrow$ Proposition
| PredicateNoInd(n) <‘(’> Term {<Comma> Term}$^{(n-1)}$ <‘)’>
| PredicateInd(1,‘Pz’) Term
| Term PredicateInd(1,‘zP’)
| Term PredicateInd(2,‘zPz’) Term

RangeFormula $\longrightarrow$ Term Comparator Term Comparator Term

Comparator $\longrightarrow$ <‘<’> | <‘=<’>

Term $\longrightarrow$ Term(p)

Term(∞) $\longrightarrow$ Variable
| Constant
| <Number>
| <Float>
| <String>
| List
| Set

| <‘`(`’> Term <‘`)`’>
| FunctionNoInd(n) <‘`(`’> Term {<Comma> Term}$^{(n-1)}$ <‘`)`’>

Term(p) ⟶ FunctionInd(1,‘`Fx`’,p) Term(q)
| FunctionInd(1,‘`Fy`’,p) Term(r)
| Term(q) FunctionInd(1,‘`xF`’,p)
| Term(r) FunctionInd(1,‘`yF`’,p)
| Term(q) FunctionInd(2,‘`xFx`’,p) Term(q1)
| Term(q) FunctionInd(2,‘`xFy`’,p) Term(r)
| Term(r) FunctionInd(2,‘`yFx`’,p) Term(q)

Condition: $p < q$, $p < q1$*, and* $p \leq r$.

Constant ⟶ UserName

Condition: A symbol with name UserName *is declared or imported as a constant.*

FunctionInd(n,i,p) ⟶ UserName

Condition: A symbol with name UserName *is declared or imported as a function with arity* n *and indicator* i(p)*.*

FunctionNoInd(n) ⟶ UserName

Condition: A symbol with name UserName *is declared or imported as a function with arity* n *and no indicator.*

Proposition ⟶ UserName

Condition: Either (i) a symbol with name UserName *is declared or imported as a proposition, or (ii)* UserName *is* `True` *or* `False`*.*

PredicateInd(n,i) ⟶ UserName

Condition: Either (i) a symbol with name UserName *is declared or imported as a predicate with arity* n *and indicator* i*, or (ii)* UserName *is* `=` *or* ∼`=`*,* n *is* 2*, and* i *is* `zPz`*.*

PredicateNoInd(n) ⟶ UserName

Condition: A symbol with name UserName *is declared or imported as a predicate with arity* n *and no indicator.*

List ⟶ <‘`[`’> [ListExpr] <‘`]`’ >

ListExpr ⟶ Term
| Term <Comma> ListExpr
| Term <‘`|`’> List
| Term <‘`|`’> Variable

Set $\longrightarrow$ <'`{`'> [SetExpr] <'`}`' >
| <'`{`'> Term <':'> Formula(f) <'`}`' >

SetExpr $\longrightarrow$ Term
| Term <Comma> SetExpr
| Term <'`|`'> Set
| Term <'`|`'> Variable

VariableSeq $\longrightarrow$ <LittleName> {<Comma> <LittleName>}

Variable $\longrightarrow$ <LittleName> | <Underscore> [<LittleName>]

The final module condition can now be checked.

M5: A module must declare every proposition or predicate defined in that module.

11.4 Goals

Goal $\longrightarrow$ <'`<-`'> GoalBody

Condition: A goal (ignoring any commits) must be a formula in the goal language of the program.

GoalBody $\longrightarrow$ Body
| <'`:`'> Body
| VariableSeq <'`:`'> Body

Condition: Each variable in VariableSeq *must occur in* Body*.*

11.5 Theories

Theory $\longrightarrow$ <'`THEORY`'> TheoryName <Terminator> {TheoryItem}

TheoryName $\longrightarrow$ <UserBigName>

Condition: TheoryName *cannot be any of* `Integers`, `Rationals`, `Floats`, `Numbers`, `Lists`, `Sets`, `Strings`, `Tables`, `Units`, `Flocks`, `Syntax`, `Programs`, `Scripts`, `Theories`, `IO`, `NumbersIO`, `FlocksIO`, `ProgramsIO`, `ScriptsIO`, *or* `TheoriesIO`*.*

TheoryItem $\longrightarrow$ ImportDecl <Terminator>
| LanguageDecl <Terminator>
| FirstOrderFormula(f) <Terminator>

Condition: A FirstOrderFormula *must be a formula in the language of the theory, which is the polymorphic many-sorted language given by the language declarations of all symbols accessible to the theory.*

FirstOrderFormula(0) ⟶ Atom
| <‘(’> FirstOrderFormula(f) <‘)’>

FirstOrderFormula(1) ⟶ <‘~’> FirstOrderFormula(f)
| <‘SOME’> <‘[’> VariableSeq <‘]’> FirstOrderFormula(f)
| <‘ALL’> <‘[’> VariableSeq <‘]’> FirstOrderFormula(f)

Condition: f $\leq$ 1.

FirstOrderFormula(2) ⟶ FirstOrderFormula(f) <‘&’> FirstOrderFormula(f1)

Condition: f $\leq$ 1, f1 $\leq$ 2.

FirstOrderFormula(3) ⟶ FirstOrderFormula(f) <‘\/’> FirstOrderFormula(f1)

Condition: f $\leq$ 2, f1 $\leq$ 3.

FirstOrderFormula(4) ⟶ FirstOrderFormula(f) <‘<-’> FirstOrderFormula(f1)
| FirstOrderFormula(f) <‘->’> FirstOrderFormula(f1)
| FirstOrderFormula(f) <‘<->’> FirstOrderFormula(f)

Condition: f $\leq$ 3, f1 $\leq$ 4.

11.6 Flocks

Flock ⟶ {Unit <Terminator>}

Unit ⟶ <Identifier>
| <Identifier> <‘(’> <‘)’>
| <Identifier> <‘(’> UnitSeq <‘)’>

UnitSeq ⟶ Unit {<Comma> Unit}

11.7 Definitions for the Ground Representation

In this section, we give some auxiliary definitions for the ground representation. These definitions complete the specification of 12 associated predicates in the module `Syntax`. To give these definitions compactly, it is convenient to use a simplified grammar similar to the one previously used to give the syntax of Gödel. In particular, we continue to use | to denote alternatives and [...] to mean zero or one occurrence.

StandardStatement ⟶ Atom `<-` [StandardBody]

NormalStatement ⟶ Atom `<-` [NormalBody]

```
DefiniteStatement    ⟶   Atom <- [DefiniteBody]

StandardResultant    ⟶   [StandardBody] <- [StandardBody]

NormalResultant      ⟶   [NormalBody] <- [NormalBody]

DefiniteResultant    ⟶   [DefiniteBody] <- [DefiniteBody]

StandardGoal         ⟶   <- [StandardBody]

NormalGoal           ⟶   <- [NormalBody]

DefiniteGoal         ⟶   <- [DefiniteBody]

StandardBody         ⟶   StandardFormula
                        | {StandardBody}_Label
                        | StandardBody & StandardBody

StandardFormula      ⟶   Atom
                        | ~ StandardFormula
                        | SOME [VariableSeq] StandardFormula
                        | ALL [VariableSeq] StandardFormula
                        | StandardFormula & StandardFormula
                        | IF StandardFormula THEN StandardFormula [ELSE StandardFormula]
                        | StandardFormula \/ StandardFormula
                        | StandardFormula <- StandardFormula
                        | StandardFormula -> StandardFormula
                        | StandardFormula <-> StandardFormula

NormalBody           ⟶   NormalFormula
                        | {NormalBody}_Label
                        | NormalBody & NormalBody

NormalFormula        ⟶   Atom
                        | ~ Atom
                        | NormalFormula & NormalFormula
                        | IF NormalFormula THEN NormalFormula [ELSE NormalFormula]

DefiniteBody         ⟶   DefiniteFormula
                        | {DefiniteBody}_Label
                        | DefiniteBody & DefiniteBody

DefiniteFormula      ⟶   Atom
                        | DefiniteFormula & DefiniteFormula
```

Chapter 12

Semantics

In this chapter, we define the declarative and procedural semantics of Gödel programs.[1]

12.1 Declarative Semantics

We begin with some definitions.

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

The *(polymorphic many-sorted) program underlying* `P` is obtained by using all the language declarations of the flat form of `P` as the language of the (polymorphic many-sorted) program and using all the statements of the flat form of `P` as its statements, except that all commits are removed and all conditionals and intensional set terms are replaced by the formulas giving their meanings, as discussed earlier. *(Polymorphic many-sorted) goals* are formulas in the language of the (polymorphic many-sorted) program underlying `P` of the form `<- W`, where `W` does not contain commits, conditionals, or intensional set terms. Any goal to `P` can be transformed to a goal for the program underlying `P` by deleting all commits, replacing any conditionals or intensional set terms by the formulas giving their meanings, and replacing the declared names of all symbols in the goal by their flat names.

Now let us turn to the issue of giving a declarative semantics to Gödel programs. The most direct and satisfactory way of giving a declarative semantics to a Gödel program is to give its intended interpretation. For the system modules of Gödel, this is what is done.[2] While it would be better to use a formal language for the purpose of specifying the intended interpretation, the more informal presentation adopted in this book serves almost as well. However, for user-defined modules, matters are more complicated as there is no direct way to state the intended interpretation for these (nor is there in any other existing logic programming language). In fact, given a Gödel program `P`, what is actually specified by the programmer is a theory which is the union of the completions of all the user-defined predicates in the (polymorphic many-sorted)

[1]This chapter is incomplete.

[2]Actually, since Gödel uses abstract data types, not all of the intended interpretation can be given, since symbols declared in the local parts of closed modules are hidden. For example, for the module `Strings`, the constants and functions used to represent strings are declared in the local part of the module and are never seen by the programmer. Thus, in general, all symbols and their meaning in the intended interpretation, except those symbols that are declared in the local parts of closed modules, can be specified.

program underlying `P` and all the equaliy axioms for user-defined types. (See appendix A.) This theory indirectly specifies the intended interpretation of the user-defined symbols since the intended interpretation is known (or at least assumed) to be one of the models of this theory.

Thus the declarative semantics of a Gödel program is defined to be the set of interpretations such that each interpretation is an extension of the intended interpretation of the system modules imported into the program and is also a model of the theory associated with the program, as defined above. An implementation of Gödel must be sound, that is, only compute answers which are correct wrt these interpretations. It is preferable that an implementation of Gödel be as complete as possible, that is, compute as many answers which are correct wrt these interpretations as possible.

12.2 Procedural Semantics

First, we discuss `DELAY` declarations. In the grammar for *Cond* given in chapter 11, the reserved word `&` stands for conjunction, `\/` 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. Now suppose an atom A is an instance by a substitution θ of an *Atom* (that is, $A = Atom\theta$) in a `DELAY` declaration. Then we say 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, we say A does *not satisfy* the corresponding condition.

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

- 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*.

Next we discuss pruning.

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

1. G_0 contains a commit labelled l and the selected literal in G_0 is in the scope of this commit, **or**
2. G_1 is derived from G_0 using an input statement which contains a commit labelled l (after standardisation apart of the commit labels).

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

Now we can define the concept of a pruning step, which gives the procedural meaning of the commit.

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

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

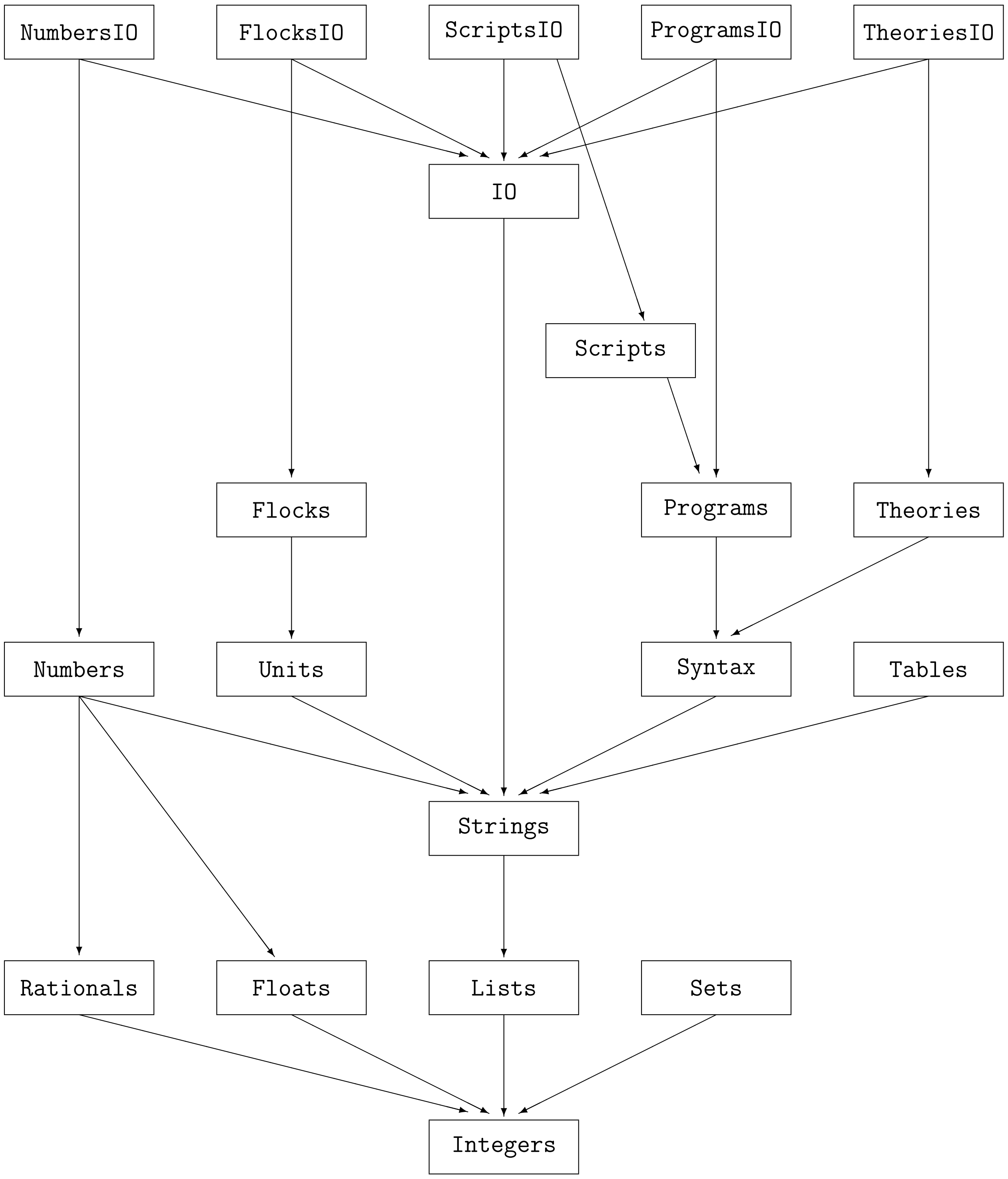
2. S' is obtained from S by removing the subtree of S rooted at G_1.

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

Chapter 13

System Modules and Utilities

The system modules `Integers`, `Rationals`, `Floats`, `Numbers`, `Lists`, `Sets`, `Strings`, `Tables`, `Units`, `Flocks`, `Syntax`, `Programs`, `Scripts`, `Theories`, `IO`, `NumbersIO`, `FlocksIO`, `ProgramsIO`, `ScriptsIO`, and `TheoriesIO` are provided by Gödel. The export parts of these are given in this chapter. The figure over the page shows the relationships between the system modules. An arrow from one module to another means the first module refers to the second.


NumbersIO
FlocksIO
ScriptsIO
ProgramsIO
TheoriesIO
IO
Scripts
Flocks
Programs
Theories
Numbers
Units
Syntax
Tables
Strings
Rationals
Floats
Lists
Sets
Integers

13.1 Integers

```
EXPORT      Integers.


% Module providing the integers and some standard functions and predicates
% with integer arguments.
%
% This module conforms to the standard for the data type Integer in Version
% 4.0 (August 1992) of the Language Independent Arithmetic Standard (LIAS)
% ISO/IEC CD 10967-1:1992 (JTC1/SC22/WG11 N318, ANSI X3T2 92-064).
%
% The intended interpretation of the symbols in this module is as follows.
%
% The intended domain of the interpretation is the integers Z. The constant
% 0 is interpreted as the integer 0, the constant 1 is interpreted as the
% integer 1, and so on. The various functions, such as +, -, etc., have their
% usual interpetation as mappings from  Z x Z (or Z, as appropriate) into Z.
% Similarly, the various predicates, such as >, <, etc., have their usual
% interpretation on Z x Z. The details are given below.
%
% The LIAS boolean bounded is false. Thus infinite precision integer
% arithmetic is provided.


BASE        Integer.
%
% Type of the integers.


% CONSTANT     0, 1, 2, ... : Integer.

FUNCTION    ^ : yFx(540) : Integer * Integer -> Integer.
%
% Exponentiation.
%
% ^ is defined by
%          x^y  = x raised to the power y,      if y >= 0
%               = 0,                            if y < 0.
%
% The function ^ is defined to be 0 when y < 0 to make it a total function.
% This result will never be used. If exponentiation with y < 0 is attempted,
% the computation will halt with an error message.
```

```
FUNCTION     - : Fy(530) : Integer -> Integer.
%
% Unary minus.
%
% The function - corresponds to the LIAS function neg.

FUNCTION     * : yFx(520) : Integer * Integer -> Integer.
%
% Multiplication.
%
% The function * corresponds to the LIAS function mul.

FUNCTION     Div : yFx(520) : Integer * Integer -> Integer.
%
% Integer division.
%
% The function Div corresponds to the LIAS functions rem and div^1. The LIAS
% rounding function rnd used is "round towards minus infinity", that is,
% rnd(x) = [x], where [] is the floor function.

FUNCTION     Mod : yFx(520) : Integer * Integer -> Integer.
%
% Modulus.
%
% Div and Mod have the defining properties given by the division algorithm:
%
% if y ~= 0, then x = (x Div y)*y + (x Mod y)
% and
%      0 =< x Mod y < y,   if y > 0
%      0 >= x Mod y > y,   if y < 0
%
% if y = 0, then x Div y = 0 and x Mod y = 0.
%
% The functions Div and Mod are defined to be 0 when y = 0 to make them total
% functions. These results will never be used.  If division by 0 is attempted,
% the computation will halt with an error message.
%
% The function Mod corresponds to the LIAS function mod.

FUNCTION     + : yFx(510) : Integer * Integer -> Integer.
%
% Addition.
%
% The function + corresponds to the LIAS function add.
```

```
FUNCTION     - : yFx(510) : Integer * Integer -> Integer.
%
% Subtraction.
%
% The function - corresponds to the LIAS function sub.


FUNCTION     Abs : Integer -> Integer.
%
% Absolute value.
%
% The function Abs corresponds to the LIAS function abs.


FUNCTION     Max : Integer * Integer -> Integer.
%
% Maximum.


FUNCTION     Min : Integer * Integer -> Integer.
%
% Minimum.


PREDICATE    > : zPz :

  Integer     % An integer greater than the integer in the second argument.
* Integer.    % An integer.

% The predicate > corresponds to the LIAS predicate gtr.


PREDICATE    < : zPz :

  Integer     % An integer less than the integer in the second argument.
* Integer.    % An integer.

% The predicate < corresponds to the LIAS predicate lss.


PREDICATE    >= : zPz :

  Integer     % An integer greater than or equal to the integer in the second
              % argument.
* Integer.    % An integer.

% The predicate >= corresponds to the LIAS predicate geq.
```

```
PREDICATE    =< : zPz :

  Integer     % An integer less than or equal to the integer in the second
              % argument.
* Integer.    % An integer.

% The predicate =< corresponds to the LIAS predicate leq.


PREDICATE    Interval :

  Integer     % An integer less than or equal to the integer in the second
              % argument.
* Integer     % An integer less than or equal to the integer in the third
              % argument.
* Integer.    % An integer.

DELAY        Interval(x,_,z) UNTIL GROUND(x) & GROUND(z).
```

13.2 Rationals

```
EXPORT      Rational.
```

```
EXPORT      Rationals.


% Module providing the rationals and some standard functions and predicates
% with rational arguments.
%
% The intended interpretation of the symbols in this module is as follows.
%
% The intended domain is the rationals Q. The various functions, such as  +, -,
% etc., have their usual interpetation as mappings from Q x Q (or Q, as
% appropriate) into Q. Similarly, the various predicates, such as >, <, etc.,
% have their usual interpretation on Q x Q. The details are given below.
%
% Note that infinite precision rational arithmetic is provided.


IMPORT      Integers.


BASE        Rational.
%
% Type of the rationals.

FUNCTION    // : yFx(520) : Integer * Integer -> Rational.
%
% For integers x and y, x//y is the rational obtained from the quotient of x by
% y.
%
% Note that // gives conversion from Integer to Rational in that x//1 is the
% rational corresponding to the integer x. Conversion from Rational to Integer
% can be achieved with a call of the form x//1 = y, where y is a rational
% reducible to the form N/1, for some integer N.
%
% // is defined to be 0 when the second argument is 0 to make it a total
% function. This result will never be used. If // is called with the second
% argument 0, the computation will halt with an error message.


FUNCTION    ^ : yFx(540) : Rational * Integer -> Rational.
%
% Exponentiation.
```

```
%
% ^ is defined to be 0 when the first argument is 0 and the second argument
% is negative to make it a total function. This result will never be used. If
% such an exponentiation is attempted, the computation will halt with an error
% message.


FUNCTION     - : Fy(530) : Rational -> Rational.
%
% Unary minus.


FUNCTION     * : yFx(520) : Rational * Rational -> Rational.
%
% Multiplication.


FUNCTION     / : yFx(520) : Rational * Rational -> Rational.
%
% Division.
%
% / is defined to be 0 when the second argument is 0 to make it a total
% function. This result will never be used. If division by 0 is attempted,
% the computation will halt with an error message.


FUNCTION     + : yFx(510) : Rational * Rational -> Rational.
%
% Addition.


FUNCTION     - : yFx(510) : Rational * Rational -> Rational.
%
% Subtraction.


FUNCTION     Abs : Rational -> Rational.
%
% Absolute value.


FUNCTION     Max : Rational * Rational -> Rational.
%
% Maximum.
```

```
FUNCTION     Min : Rational * Rational -> Rational.
%
% Minimum.


PREDICATE    > : zPz :

  Rational      % A rational greater than the rational in the second argument.
* Rational.     % A rational.


PREDICATE    < : zPz :

  Rational      % A rational less than the rational in the second argument.
* Rational.     % A rational.


PREDICATE    >= : zPz :

  Rational      % A rational greater than or equal to the rational in the second
                % argument.
* Rational.     % A rational.


PREDICATE    =< : zPz :

  Rational      % A rational less than or equal to the rational in the second
                % argument.
* Rational.     % A rational.


PREDICATE    StandardRational :

  Rational      % A rational.
* Integer       % The numerator of the rational in standard form equal to this
                % rational.
* Integer.      % The denominator of the rational in standard form equal to this
                % rational.

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

13.3 Floats

```
EXPORT      Floats.


% Module providing floating-point numbers and some standard functions and
% predicates with floating-point arguments.
%
% This module conforms to the standard for the data type Floating-Point in
% Version 4.0 (August 1992) of the Language Independent Arithmetic Standard
% (LIAS) ISO/IEC CD 10967-1:1992 (JTC1/SC22/WG11 N318, ANSI X3T2 92-064).
% It also conforms to the ANSI/IEEE Standard for Binary Floating-Point
% Arithmetic 754-1985.
%
% The four IEEE Standard 754-1985 rounding functions (round toward nearest,
% round toward plus infinity, round toward minus infinity, and round toward
% zero) are provided by compiler options. The default rounding function is
% round toward nearest.
%
% The intended interpretation of the symbols in this module is as follows.
%
% The intended domain of the interpretation is the finite set F of floating-
% point numbers characterised by a fixed radix, a fixed precision, and fixed
% smallest and largest exponent. Thus F is the finite set of numbers of the
% form either 0 or s0.f1...fp * r^e, where r is the radix, p is the precision,
% e is the exponent, s is either + or -, and each fi satisfies 0 =< fi < r.
% Note that the LIAS boolean denorm is true. Thus denormalised floating-point
% numbers are provided.
%
% The language contains finitely many constants, exactly one corresponding to
% each floating-point number in F. However, for the convenience of the user,
% there is some syntactic sugar used instead of the names of these constants.
% This is the usual decimal number notation, with or without an exponent.
% Typical decimal numbers without exponent are 3.1416 and 0, and typical
% decimal numbers with exponent are -2.345619E-12 and 674328.89E+2. Such decimal
% numbers are converted (according to the ANSI/IEEE standard 754-1985) by the
% system to floating-point numbers in the form above. Then the convention is
% that a decimal number is syntactic sugar for the constant whose interpretation
% is the floating-point number obtained from the decimal number. This means that
% there is more than one way of denoting each of these constants. For example,
% both 3.1416 and 314.16E-2 denote the same constant. Similarly, when answers
% are displayed by the system, floating-point numbers are converted back to the
% more convenient decimal form.
```

```
%
% The various functions, such as +, -, etc., have their usual interpretation
% as mappings from  F x F (or F, as appropriate) into F. Similarly, the various
% predicates, such as >, <, etc., have their usual interpretation on F x F.
% The details are given below.


IMPORT       Integers.


BASE         Float.
%
% Type of the floating-point numbers.


% CONSTANT    Finitely many constants, one for each number in the finite set F
% of floating-point numbers determined by the radix, precision, and smallest
% and largest exponent.

FUNCTION    ^ : yFx(540) : Float * Float -> Float.
%
% Exponentiation.
%
% ^ is defined by
%      x^y  = Exp(y*Log(x)),     if x>0 and no underflow or overflow occurs
%           = 0,                 otherwise.
% The function ^ is defined to be 0 when x =< 0 or underflow or overflow occurs
% to make it a total function. This result will never be used. If exponentiation
% is attempted with x =< 0 or underflow or overflow occurs, the computation
% will halt with an appropriate error message.


FUNCTION     - : Fy(530) : Float -> Float.
%
% Unary minus.
%
% The function - corresponds to the LIAS function neg.


FUNCTION     * : yFx(520) : Float * Float -> Float.
%
% Multiplication.
%
% * can overflow or underflow. In either case, the multiplication is defined to
% be 0 to make * total. Such a result will never be used. If a multiplication
% leads to an underflow or overflow, the computation will halt with an
% appropriate error message.
%
% The function * corresponds to the LIAS function mul.

FUNCTION     / : yFx(520) : Float * Float -> Float.
%
% Division.
%
% / can overflow, underflow, or have a zero divisor. In all such cases, the
% division is defined to be 0 to make / total. Such a result will never be used.
% If a division leads to an underflow, overflow, or a zero divisor, the
% computation will halt with an appropriate error message.
%
% The function / corresponds to the LIAS function div.

FUNCTION     + : yFx(510) : Float * Float -> Float.
%
% Addition.
%
% + can overflow or underflow. In either case, the addition is defined to be
% 0 to make + total. Such a result will never be used. If an addition leads to
% an underflow or overflow, the computation will halt with an appropriate error
% message.
%
% The function + corresponds to the LIAS function add. The LIAS approximate
% addition function add* is true (exact) addition.

FUNCTION     - : yFx(510) : Float * Float -> Float.
%
% Subtraction.
%
% - can overflow or underflow. In either case, the subtraction is defined to
% be 0 to make - total. Such a result will never be used. If a subtraction
% leads to an underflow or overflow, the computation will halt with an
% appropriate error message.
%
% The function - corresponds to the LIAS function sub.

FUNCTION     Abs : Float -> Float.
%
% Absolute value.
%
% The function Abs corresponds to the LIAS function abs.
```

```
FUNCTION     Max : Float * Float -> Float.
%
% Maximum.


FUNCTION     Min : Float * Float -> Float.
%
% Minimum.


FUNCTION     Sqrt : Float -> Float.
%
% Square root.
%
% Sqrt is defined to be 0 when its argument is negative to make it total. Such
% a result will never be used. In such a case, the computation will halt with
% an appropriate error message.
%
% The function Sqrt corresponds to the LIAS function sqrt.


FUNCTION     Sign : Float -> Float.
%
% Sign(x) = 1,   if x>=0
%          -1,   otherwise.
%
% The function Sign corresponds to the LIAS function sign.


FUNCTION     Fraction : Float -> Float.
%
% The fraction part s0.f1...fp of a non-zero floating-point number
% s0.f1...fp * r^e or 0 for the floating-point number 0.
%
% The function Fraction corresponds to the LIAS function fraction.


FUNCTION     Scale : Float * Integer -> Float.
%
% Scale scales a floating-point number in the first argument by an integer
% power of the radix, where the integer is in the second argument. Scale is
% defined to be 0 when underflow or overflow occurs to make it a total function.
% Such a result will never be used. If underflow or overflow occurs, the
% computation will halt with an error message.
%
% The function Scale corresponds to the LIAS function scale.
```

```
FUNCTION     Successor : Float -> Float.
%
% Successor returns the closest number in F greater than its argument.
% Successor is defined to be 0 when overflow occurs to make it a total function.
% Such a result will never be used. If overflow occurs, the computation will
% halt with an error message.
%
% The function Successor corresponds to the LIAS function succ.


FUNCTION     Predecessor : Float -> Float.
%
% Predecessor returns the closest number in F less than its argument.
% Predecessor is defined to be 0 when overflow occurs to make it a total
% function. Such a result will never be used. If overflow occurs, the
% computation will halt with an error message.
%
% The function Predecessor corresponds to the LIAS function pred.


FUNCTION     UnitInLastPlace : Float -> Float.
%
% UnitInLastPlace gives the weight of the least significant digit of a non-zero
% argument. UnitInLastPlace is defined to be 0 when its argument is 0 or
% underflow occurs to make it a total function. Such a result will never be
% used. If its argument is 0 or underflow occurs, the computation will halt
% with an error message.
%
% The function UnitInLastPlace corresponds to the LIAS function ulp.


FUNCTION     Truncate : Float * Integer -> Float.
%
% Truncate zeros out the low (p - n) digits of the argument, where p is the
% precision and n is the second argument. Truncate is defined to be 0 when its
% second argument is non-positive to make it a total function. Such a result
% will never be used. If its second argument is non-positive, the  computation
% will halt with an error message.
%
% The function Truncate corresponds to the LIAS function trunc.


FUNCTION     Round : Float * Integer -> Float.
%
% Round rounds its argument to n significant digits; the low (p - n) digits
% are all zeros, where p is the precision and n is the second argument. Round
% is defined to be 0 when its second argument is non-positive or underflow
% occurs to make it a total function. Such a result will never be used. If its
% second argument is non-positive or overflow occurs, the computation will halt
% with an error message.
%
% The function Round corresponds to the LIAS function round.


FUNCTION     IntegerPart : Float -> Float.
%
% IntegerPart returns (in floating-point form) the integer part of a floating
% point number.
%
% The function IntegerPart corresponds to the LIAS function intpart.


FUNCTION     FractionalPart : Float -> Float.
%
% FractionalPart returns the value of its argument minus its integer part.
%
% The function FractionalPart corresponds to the LIAS function fractpart.


FUNCTION     Sin : Float -> Float.
%
% Sine.


FUNCTION     Cos : Float -> Float.
%
% Cosine.


FUNCTION     Tan : Float -> Float.
%
% Tangent.
%
% Tan is defined to be 0 when overflow occurs to make it total. Such a result
% will never be used.  If overflow occurs, the computation will halt with an
% error message.


FUNCTION     ArcSin : Float -> Float.
%
% ArcSine.
%
% ArcSin is defined to be 0 when its argument is outside the range [-1,1] to
% make it total. Such a result will never be used.  If ArcSin is called with
% such an argument, the computation will halt with an error message.
```

```
FUNCTION     ArcCos : Float -> Float.
%
% ArcCosine.
%
% ArcCos is defined to be 0 when its argument is outside the range [-1,1] to
% make it total. Such a result will never be used.  If ArcCos is called with
% such an argument, the computation will halt with an error message.

FUNCTION     ArcTan : Float -> Float.
%
% ArcTangent.

FUNCTION     Exp : Float -> Float.
%
% Exponential.
%
% Exponential is defined to be 0 when underflow or overflow occurs to make it
% total. Such a result will never be used.  If underflow or overflow occurs,
% the computation will halt with an error message.

FUNCTION     Log : Float -> Float.
%
% Natural logarithm.
%
% Log is defined to be 0 when its argument is not greater than 0 or underflow
% or overflow occurs to make it total. Such a result will never be used. In such
% cases, the computation will halt with an error message.

FUNCTION     Log10 : Float -> Float.
%
% Base 10 logarithm.
%
% Log10 is defined to be 0 when its argument is not greater than 0 or underflow
% or overflow occurs to make it total. Such a result will never be used. In such
% cases, the computation will halt with an error message.

PREDICATE    IntegerToFloat :

  Integer     % An integer.
* Float.      % The floating-point number resulting from the conversion of this
              % integer.

DELAY        IntegerToFloat(x,_) UNTIL GROUND(x).

% The predicate IntegerToFloat corresponds to the LIAS function cvt_{I->F}.
```

```
PREDICATE    TruncateToInteger :

  Float       % A floating-point number.
* Integer.    % The integer resulting from the conversion of this floating-point
              % number using the function x -> sign(x).[|x|], where sign gives
              % the sign of its argument, [] is the floor function, and . is
              % multiplication.

DELAY        TruncateToInteger(x,_) UNTIL GROUND(x).

% The predicate TruncateToInteger corresponds to an LIAS function cvt_{F->I}.


PREDICATE    RoundToInteger :

  Float       % A floating-point number.
* Integer.    % The integer resulting from the conversion of this floating-point
              % number using the function x -> sign(x).[|x| + 1/2], where sign
              % gives the sign of its argument, [] is the floor function, and .
              % is multiplication.

DELAY        RoundToInteger(x,_) UNTIL GROUND(x).

% The predicate Nearest corresponds to an LIAS function cvt_{F->I}.


PREDICATE    Floor :

  Float       % A floating-point number.
* Integer.    % The integer resulting from the conversion of this floating-point
              % number using the floor function.

DELAY        Floor(x,_) UNTIL GROUND(x).

% The predicate Floor corresponds to an LIAS function cvt_{F->I}.


PREDICATE    Ceiling :

  Float       % A floating-point number.
* Integer.    % The integer resulting from the conversion of this floating-point
              % number using the ceiling function.

DELAY        Ceiling(x,_) UNTIL GROUND(x).

% The predicate Ceiling corresponds to an LIAS function cvt_{F->I}.
```

```
PREDICATE    Exponent :

  Float       % A (non-zero) floating-point number s0.f1...fp * r^e.
* Integer.    % The exponent e of this number.

DELAY        Exponent(x,_) UNTIL GROUND(x).

% The predicate Exponent corresponds to the LIAS function exponent.


PREDICATE    Radix :

  Integer.    % The radix 2 used in the representation of floating-point
              % numbers.

% The predicate Radix gives the LIAS parameter r.


PREDICATE    Precision :

  Integer.    % The maximum number 24 of radix digits allowed in the
              % representation of floating-point numbers.

% The predicate Precision gives the LIAS parameter p.


PREDICATE    MaxExponent :

  Integer.    % The maximum exponent 128 allowed in the representation of
              % floating-point numbers.

% The predicate MaxExponent gives the LIAS parameter emax.


PREDICATE    MinExponent :

  Integer.    % The minimum exponent -125 allowed in the representation of
              % floating-point numbers.

% The predicate MinExponent gives the LIAS parameter emin.


PREDICATE    MaxFloat :

  Float.      % The largest floating-point number 3.402823466E+38 (approx.).

% The predicate MaxFloat gives the LIAS constant fmax.
```

```
PREDICATE    MinNormFloat :

  Float.      % The normalised floating-point number 1.175494351E-38 (approx.)
              % with the smallest magnitude.

% The predicate MinNormFloat gives the LIAS constant fmin_{N}.


PREDICATE    MinFloat :

  Float.      % The denormalised floating-point number 1.401298464E-45 (approx.)
              % with the smallest magnitude.

% The predicate MinNormFloat gives the LIAS constant fmin_{D}.


PREDICATE    Epsilon :

  Float.      % Maximum relative spacing 1.192092896E-7 (approx.) in F.

% The predicate Epsilon gives the LIAS constant epsilon.


PREDICATE    > : zPz :

  Float       % A floating-point number greater than the number in the second
              % argument.
* Float.      % A floating-point number.

DELAY        x > y UNTIL GROUND(x) & GROUND(y).

% The predicate > corresponds to the LIAS predicate gtr.


PREDICATE    < : zPz :

  Float       % A floating-point number less than the number in the second
              % argument.
* Float.      % A floating-point number.

DELAY        x < y UNTIL GROUND(x) & GROUND(y).

% The predicate < corresponds to the LIAS predicate lss.
```

```
PREDICATE    >= : zPz :

  Float       % A floating-point number greater than or equal to the number in
              % the second argument.
* Float.      % A floating-point number.

DELAY        x >= y UNTIL GROUND(x) & GROUND(y).

% The predicate >= corresponds to the LIAS predicate geq.


PREDICATE    =< : zPz :

  Float       % A floating-point number less than or equal to the number in the
              % second argument.
* Float.      % A floating-point number.

DELAY        x =< y UNTIL GROUND(x) & GROUND(y).

% The predicate =< corresponds to the LIAS predicate leq.
```

13.4 Numbers

```
EXPORT       Numbers.

% Module providing various conversion predicates for integer, rational, and
% floating-point numbers.


IMPORT       Rationals, Floats, Strings.

PREDICATE    RationalToFloat :

  Rational    % A rational.
* Float.      % The floating-point number nearest the rational. In the case of
              % a value exactly half-way between two neighbouring values in F,
              % the one selected will be the one with the least significant bit
              % zero. If underflow or overflow occurs, the computation will halt
              % with an error message.

DELAY        RationalToFloat(x,_) UNTIL GROUND(x).

PREDICATE    FloatToRational :

  Float       % A floating-point number.
* Rational.   % The rational resulting from the exact conversion of this
              % floating-point number.

DELAY        FloatToRational(x,_) UNTIL GROUND(x).

PREDICATE    IntegerString :

  Integer        % An integer.
* String.        % The string consisting of the characters in the integer.

DELAY        IntegerString(x,y) UNTIL GROUND(x) \/ GROUND(y).

PREDICATE    RationalString :

  Rational       % A rational.
* String.        % The string consisting of the characters in the rational.

DELAY        RationalString(x,y) UNTIL GROUND(x) \/ GROUND(y).
```

```
PREDICATE    FloatString :

  Float          % A floating-point number.
* String.        % The string consisting of the characters in the floating-point
                 % number.

DELAY        FloatString(x,y) UNTIL GROUND(x) \/ GROUND(y).
```

13.5 Lists

```
EXPORT       Lists.


% Module providing a collection of standard list processing predicates.


IMPORT       Integers.


CONSTRUCTOR  List/1.
%
% List constructor.


CONSTANT     Nil : List(a).
%
% Empty list.


FUNCTION     Cons : a * List(a) -> List(a).
%
% Cons function.


PREDICATE    Member :

  a                % An element.
* List(a).         % A list containing this element.

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


PREDICATE    MemberCheck :

  a                % An element.
* List(a).         % A list containing this element.
                   %
                   % MemberCheck is a version of Member which efficiently checks
                   % whether a given element is a member of a given list. It
                   % prunes the search space so that MemberCheck succeeds at
                   % most once.
                   %
                   % If s is the first argument of a call to MemberCheck and any
                   % answers to the call s = t, where t ranges over all elements
                   % of the list, are variants of one another, then no
                   % non-redundant answers to MemberCheck will be pruned.
                   % (In particular, if the call to MemberCheck is ground, this
                   % condition is automatically satisfied.)

DELAY         MemberCheck(_, []) UNTIL TRUE;
              MemberCheck(x, [y|_]) UNTIL NONVAR(x) & NONVAR(y).

PREDICATE     Append :

  List(a)          % A list.
* List(a)          % A list.
* List(a).         % The list obtained by appending the lists in the first
                   % and second arguments.

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

PREDICATE     Permutation :

  List(a)          % A list.
* List(a).         % A list which is a permutation of the list in the first
                   % argument.

DELAY         Permutation(x,y) UNTIL NONVAR(x) \/ NONVAR(y).

PREDICATE     Delete :

  a                % An element.
* List(a)          % A list.
* List(a).         % A list which differs from the list in the second argument
                   % only in that exactly one occurrence of the element in the
                   % first argument is missing.

DELAY         Delete(_,y,z) UNTIL NONVAR(y) \/ NONVAR(z).

PREDICATE     DeleteFirst :

  a                % An element.
* List(a)          % A list.
* List(a).         % A list which differs from the list in the second argument
                   % only in that the first occurrence of the element in the
                   % first argument is missing.

DELAY         DeleteFirst(x,y,z) UNTIL NONVAR(x) & (NONVAR(y) \/ NONVAR(z)).
```

```
PREDICATE    Reverse :

  List(a)          % A list.
* List(a).         % The list consisting of the elements of the list in the
                   % first argument in reverse order.

DELAY        Reverse(x,y) UNTIL NONVAR(x) \/ NONVAR(y).

PREDICATE    Prefix :

  List(a)          % A list.
* Integer          % A non-negative integer less than or equal to the length
                   % of the list in the first argument.
* List(a).         % The prefix of the list in the first argument having length
                   % equal to the integer in the second argument.

DELAY        Prefix(x,y,z) UNTIL (NONVAR(x) & NONVAR(y))
                                 \/ (NONVAR(x) & NONVAR(z)).

PREDICATE    Suffix :

  List(a)          % A list.
* Integer          % A non-negative integer less than or equal to the length of
                   % the list in the first argument.
* List(a).         % The suffix of the list in the first argument having length
                   % equal to the integer in the second argument.

DELAY        Suffix(x,y,z) UNTIL (NONVAR(x) & NONVAR(y))
                                 \/ (NONVAR(x) & NONVAR(z)).

PREDICATE    Length :

  List(a)          % A list.
* Integer.         % The length of the list in the first argument.

DELAY        Length(x,y) UNTIL NONVAR(x) \/ NONVAR(y).

PREDICATE    Sorted :

  List(Integer).   % A list of integers in non-decreasing order.

DELAY        Sorted([]) UNTIL TRUE;
             Sorted([_]) UNTIL TRUE;
             Sorted([x,y|_]) UNTIL NONVAR(x) & NONVAR(y).
```

```
PREDICATE    Sort :

  List(Integer)    % A list of integers.
* List(Integer).   % The list consisting of the elements of the list in the
                   % first argument in non-decreasing order.

DELAY        Sort(x,_) UNTIL NONVAR(x).


PREDICATE    Merge :

  List(Integer)    % A list of integers.
* List(Integer)    % A list of integers.
* List(Integer).   % The list which is the result of merging the lists in the
                   % first and second arguments.

DELAY        Merge(x,y,z) UNTIL (NONVAR(x) & NONVAR(y)) \/ NONVAR(z).
```

13.6 Sets

```
EXPORT      Sets.


% Module providing finite sets and a collection of standard set processing
% functions and predicates.
%
% Also provided by this module are intensional set terms which have the form
%
% {T : W}
%
% where T is a term with free variables y1,...,yn, say, and W is a formula
% (not involving commits) which has y1,...,yn amongst its free variables. The
% variables y1,...,yn must be local to {T : W}. The free variables of {T : W}
% are the free variables of W other than y1,...,yn. (Note that T may itself be
% an intensional set term and that it is possible for n to be 0.)
%
% Intuitively, {T : W} means "the set of all instances of T corresponding to
% the instances of W which are true".


IMPORT      Integers.


CONSTRUCTOR Set/1.
%
% Set constructor.


CONSTANT    Null : Set(a).
%
% Empty set.

FUNCTION    Inc : a * Set(a) -> Set(a).
%
% The function Inc (short for Include) used to form sets. The intended meaning
% of Inc is the mapping Inc' such that Inc'(d,S) = {d} union S, where d is an
% element from the domain of type a and S is a set of elements of type a.

FUNCTION    *  : yFx(120) : Set(a) * Set(a) -> Set(a).
%
% Set-theoretic intersection.
```

```
FUNCTION     +  : yFx(110) : Set(a) * Set(a) -> Set(a).
%
% Set-theoretic union.


FUNCTION     \ : yFx(100) : Set(a) * Set(a) -> Set(a).
%
% Set-theoretic difference.


PREDICATE    In : zPz :

  a             % An element.
* Set(a).       % A set containing this element.

DELAY        _ In y UNTIL GROUND(y).


PREDICATE    Subset : zPz :

  Set(a)        % A subset of the set in the second argument.
* Set(a).       % A set.

DELAY        _ Subset y UNTIL GROUND(y).


PREDICATE    StrictSubset : zPz :

  Set(a)        % A strict subset of the set in the second argument.
* Set(a).       % A set.

DELAY        _ StrictSubset y UNTIL GROUND(y).


PREDICATE    Size :

  Set(a)        % A finite set.
* Integer.      % The number of elements in the set.

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

13.7 Strings

```
EXPORT     Strings.


% Module providing strings and a collection of standard string processing
% predicates.


IMPORT     Lists.


BASE       String.
%
% Type of a string.


FUNCTION   ++ : yFx(500) : String * String -> String.
%
% String concatenation.


PREDICATE  StringInts :

  String          % A string.
* List(Integer).  % The list of ASCII codes of the characters in this string
                  % in the order in which they appear there.

DELAY      StringInts(x,y) UNTIL GROUND(x) \/ GROUND(y).


PREDICATE  FirstSubstring :

  String          % A string.
* Integer         % A non-negative integer n less than or equal to the length of
                  % the string in the first argument.
* String.         % The substring consisting of the first n characters of the
                  % string in the first argument.

DELAY      FirstSubstring(x,y,z) UNTIL GROUND(x) & (GROUND(y) \/ GROUND(z)).
```

```
PREDICATE  LastSubstring :

  String          % A string.
* Integer         % A non-negative integer n less than or equal to the length of
                  % this string.
* String.         % The substring consisting of the last n characters of the
                  % string in the first argument.

DELAY      LastSubstring(x,y,z) UNTIL GROUND(x) & (GROUND(y) \/ GROUND(z)).


PREDICATE  Width :

  String          % A string.
* Integer.        % The number of characters in this string.

DELAY      Width(x,_) UNTIL GROUND(x).


PREDICATE  > : zPz :

  String          % A string lexically greater than the string in the second
                  % argument.
* String.         % A string.

DELAY      x > y UNTIL GROUND(x) & GROUND(y).


PREDICATE  < : zPz :

  String          % A string lexically less than the string in the second
                  % argument.
* String.         % A string.

DELAY      x < y UNTIL GROUND(x) & GROUND(y).


PREDICATE  >= : zPz :

  String          % A string lexically greater than or equal to the string in
                  % the second argument.
* String.         % A string.

DELAY      x >= y UNTIL GROUND(x) & GROUND(y).
```

```
PREDICATE  =< : zPz :

  String          % A string lexically less than or equal to the string in the
                  % second argument.
* String.         % A string.

DELAY      x =< y UNTIL GROUND(x) & GROUND(y).
```

13.8 Tables

```
EXPORT       Tables.

% Module providing tables, which are data structures consisting of an ordered
% collection of nodes, each of which has two components, a key and a value.
% The key must be a string, but the value can have any type. The ordering is
% the lexical ordering of the keys.

IMPORT       Strings.

CONSTRUCTOR  Table/1.


PREDICATE    EmptyTable :

  Table(a).         % An empty table.


PREDICATE    NodeInTable :

  Table(a)          % A table.
* String            % The key of a node in the table.
* a.                % The value in this node.

DELAY        NodeInTable(x,_,_) UNTIL NONVAR(x).


PREDICATE    InsertNode :

  Table(a)          % A table.
* String            % A key, not in this table.
* a                 % A value.
* Table(a).         % A table which differs from the table in the first
                    % argument only in that it also contains a node with this
                    % key and value.

DELAY        InsertNode(x,y,_,_) UNTIL NONVAR(x) & GROUND(y).


PREDICATE    DeleteNode :

  Table(a)          % A table.
* String            % A key in this table.
* a                  % A value.
* Table(a).          % A table which differs from the table in the first
                     % argument only in that it does not contain a node with
                     % this key and value.

DELAY        DeleteNode(x,_,_,_) UNTIL NONVAR(x).


PREDICATE    UpdateTable :

  Table(a)           % A table.
* String             % The key of a node in the table.
* a                  % A new value to be associated with the key.
* Table(a)           % The table with the node updated.
* a.                 % The old value that was associated with the key.

DELAY        UpdateTable(x,y,_,_,_) UNTIL NONVAR(x) & GROUND(y).


PREDICATE    AmendTable :

  Table(a)           % A table.
* String             % A key.
* a                  % A (new) value to be associated with the key.
* a                  % 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.)
* a.                 % The old value associated with the key if it was already
                     % present in the table, otherwise the value in the fourth
                     % argument.

DELAY        AmendTable(x,y,_,_,_,_) UNTIL NONVAR(x) & GROUND(y).


PREDICATE    JoinTables :

  Table(a)           % A table.
* Table(a)           % A table.
* Table(a).          % The table formed from the first argument by adding to
                     % it all the nodes in the second argument whose keys are
                     % not already present in the first argument.

DELAY        JoinTables(x,y,_) UNTIL NONVAR(x) & NONVAR(y).
```

```
PREDICATE    ListTable :

  Table(a)          % A table.
* List(String)      % The lexically ordered list of keys of nodes in the table.
* List(a).          % The corresponding list of values.

DELAY        ListTable(x,y,z) UNTIL NONVAR(x) \/ (NONVAR(y) & NONVAR(z)).

PREDICATE    FirstNode :

  Table(a)          % A table.
* String            % The key of the node which is lexically first amongst
                    % all keys in the table.
* a.                % The value of this node.

DELAY        FirstNode(x,_,_) UNTIL NONVAR(x).

PREDICATE    LastNode :

  Table(a)          % A table.
* String            % The key of the node which is lexically last amongst
                    % all keys in the table.
* a.                % The value of this node.

DELAY        LastNode(x,_,_) UNTIL NONVAR(x).

PREDICATE    NextNode :

  Table(a)          % A table.
* String            % The key of a node in the table.
* String            % The key of the node which is the lexical successor of
                    % the key in the second argument.
* a.                % The value of this node.

DELAY        NextNode(x,y,_,_) UNTIL NONVAR(x) & NONVAR(y).

PREDICATE    PreviousNode :

  Table(a)          % A table.
* String            % The key of a node in the table.
* String            % The key of the node which is the lexical predecessor
                    % of the key in the second argument.
* a.                % The value of this node.

DELAY        PreviousNode(x,y,_,_) UNTIL NONVAR(x) & NONVAR(y).
```

13.9 Units

```
EXPORT     Units.


% Module providing units and some predicates which process them.


IMPORT     Strings.


BASE       Unit.
%
% Type of a unit.


PREDICATE  StringToUnit :

  String           % String representation of a unit.
* Unit.            % This unit.

DELAY      StringToUnit(x,_) UNTIL GROUND(x).


PREDICATE  UnitToString :

  Unit             % A unit.
* String.          % String representation of this unit.

DELAY      UnitToString(x,_) UNTIL GROUND(x).


PREDICATE  UnitParts :

  Unit             % A unit.
* String           % String representation of the top-level identifier of this
                   % unit.
* List(Unit).      % List of top-level subunits of this unit. (The list is empty
                   % if the unit is just an identifier.)
```

```
PREDICATE  UnitArgument :

  Unit            % A unit.
* Integer         % A positive integer n.
* Unit.           % The nth top-level subunit of this unit.

DELAY      UnitArgument(x,_,_) UNTIL NONVAR(x).
```

13.10 Flocks

```
EXPORT     Flocks.


% Module providing flocks, which are ordered collections of units, and some
% predicates which process them.


IMPORT     Units.


BASE       Flock.
%
% Type of a flock.

PREDICATE  EmptyFlock :

  Flock.          % An empty flock.

PREDICATE  Extent :

  Flock           % A flock.
* Integer.        % The number of units in this flock.

DELAY      Extent(x,_) UNTIL GROUND(x).

PREDICATE  UnitInFlock :

  Flock           % A flock.
* Unit            % A unit in this flock.
* Integer.        % Position of this unit in this flock.

DELAY      UnitInFlock(x,_,_) UNTIL GROUND(x).

PREDICATE  UnitWithIdentifier :

  Flock           % A flock.
* String          % String representation of an identifier.
* Unit            % A unit in this flock whose top-level identifier is the
                  % identifier in the second argument.
* Integer.        % Position of this unit in this flock.

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

```
PREDICATE  InsertUnit :

  Flock           % A flock.
* Unit            % A unit.
* Integer         % A positive integer n.
* Flock.          % A flock which differs from the flock in the first argument
                  % only in that it also contains this unit in the nth position.

DELAY      InsertUnit(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).


PREDICATE  DeleteUnit :

  Flock           % A flock.
* Unit            % A unit in this flock.
* Integer         % The position of this unit in this flock.
* Flock.          % A flock which differs from the flock in the first argument
                  % only in that it does not contain this unit in this position.

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

13.11 Syntax

```
EXPORT     Syntax.


% Module containing predicates for manipulating (the ground representations of)
% object level expressions.


IMPORT     Strings.

BASE       Name,           % Type of a term representing the name of a symbol.
           Type,           % Type of a term representing a type.
           Term,           % Type of a term representing a 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.


FUNCTION   XFX, XFY, YFX, XF, FX, YF, FY : Integer -> FunctionInd.
%
% 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  W & V.

PREDICATE  AndWithEmpty :

  Formula         % Representation of possibly empty formula W.
* Formula         % Representation of possibly empty formula V.
* Formula.        % Representation of W & V, if W and V are non-empty; 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.
* Formula.        % Representation of the formula W \/ V.

PREDICATE  Not :

  Formula         % Representation of a formula W.
* Formula.        % Representation of the formula ~ W.

PREDICATE  Implies :

  Formula         % Representation of a formula W.
* Formula         % Representation of a formula V.
* Formula.        % Representation of the formula W -> V.

PREDICATE  IsImpliedBy :

  Formula         % Representation of a formula W.
* Formula         % Representation of a formula V.
* Formula.        % Representation of the formula W <- V.

PREDICATE  Equivalent :

  Formula         % Representation of a formula W.
* Formula         % Representation of a formula V.
* Formula.        % Representation of the formula W <-> V.
```

```
PREDICATE  Some :

  List(Term)      % List of representations of variables.
* Formula         % Representation of a formula.
* Formula.        % Representation of the formula obtained by taking the
                  % existential quantification over the set of 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.
* Formula.        % Representation of the IF-THEN construct which has the form
                  % IF Condition THEN Formula.


PREDICATE  IfSomeThen :

  List(Term)      % List of representations of variables, Listvar.
* Formula         % Representation of a formula, Condition.
* Formula         % Representation of a formula, Formula.
* Formula.        % Representation of the IF-THEN construct which has the form
                  % IF SOME Listvar Condition THEN Formula.


PREDICATE  IfThenElse :

  Formula         % Representation of a formula, Condition.
* Formula         % Representation of a formula, Formula1.
* Formula         % Representation of a formula, Formula2.
* Formula.        % Representation of the IF-THEN-ELSE construct which has the
                  % form IF Condition THEN Formula1 ELSE Formula2.
```

```
PREDICATE  IfSomeThenElse :

  List(Term)      % List of representations of variables, Listvar.
* Formula         % Representation of a formula, Condition.
* Formula         % Representation of a formula, Formula1.
* Formula         % Representation of a formula, Formula2.
* Formula.        % Representation of the IF-THEN-ELSE construct which has the
                  % form IF SOME Listvar Condition THEN Formula1 ELSE Formula2.

PREDICATE  Commit :

  Integer         % Commit label.
* Formula         % Representation of a formula.
* Formula.        % Representation of the formula obtained by enclosing the
                  % formula in the second argument using commits with this
                  % label.

PREDICATE  IntensionalSet :

  Term            % Representation of a term T.
* Formula         % Representation of a formula W.
* Term.           % Representation of the intensional set term {T : W}.

PREDICATE  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  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  TermMaxVarIndex :

  List(Term)      % List of representations of terms.
* Integer.        % One more than the maximum index of variables appearing in
                  % these terms. (If there are no such variables, this argument
                  % is 0.)

DELAY      TermMaxVarIndex(x,_) UNTIL GROUND(x).

PREDICATE  FormulaVariables :

  Formula         % Representation of a formula.
* List(Term).     % List (in some order) of the representations of the free
                  % variables occurring in this formula.

DELAY      FormulaVariables(x,_) UNTIL GROUND(x).

PREDICATE  TermVariables :

  Term            % Representation of a term.
* List(Term).     % List (in some order) of the representations of the (free)
                  % variables occurring in this term.

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

```
PREDICATE  TypeParameters :

  Type            % Representation of a type.
* List(Type).     % List (in some order) of the representations of the
                  % parameters occurring in this type.

DELAY      TypeParameters(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  NonParType :

  Type.           % Representation of a non-parameter type.

DELAY      NonParType(x) UNTIL GROUND(x).


PREDICATE  NonVarTerm :

  Term.           % Representation of a non-variable term.

DELAY      NonVarTerm(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).


PREDICATE  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 ground type.

DELAY      GroundType(x) UNTIL GROUND(x).


PREDICATE  GroundTerm :

  Term.           % Representation of a ground term.

DELAY      GroundTerm(x) UNTIL GROUND(x).


PREDICATE  GroundAtom :

  Formula.        % Representation of a ground atom.

DELAY      GroundAtom(x) UNTIL GROUND(x).
```

```
PREDICATE  ClosedFormula :

  Formula.        % Representation of a closed formula.

DELAY      ClosedFormula(x) UNTIL GROUND(x).


PREDICATE  Body :

  Formula.        % Representation of a standard body.

DELAY      Body(x) UNTIL GROUND(x).


PREDICATE  NormalBody :

  Formula.        % Representation of a normal body.

DELAY      NormalBody(x) UNTIL GROUND(x).


PREDICATE  DefiniteBody :

  Formula.        % Representation of a definite body.

DELAY      DefiniteBody(x) UNTIL GROUND(x).


PREDICATE  Goal :

  Formula.        % Representation of a standard 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.
* List(Term).     % List of representations of the top-level subterms 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.
* List(Term).     % List of representations of the top-level terms of 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.
* Type.           % Representation of the type obtained by applying 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.
* Term.           % Representation of the term obtained by applying this
                  % substitution to this term.

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.
* TypeSubst.      % Representation of the substitution obtained by composing
                  % these two substitutions (in the order 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.
* VarTyping.      % Representation of the variable typing obtained by combining
                  % these two variable typings (in the order that they appear as
                  % arguments).

DELAY      CombineVarTypings(x,y,_) UNTIL GROUND(x) & GROUND(y).


PREDICATE  RestrictSubstToType :

  Type            % Representation of a type.
* TypeSubst       % Representation of a type substitution.
* TypeSubst.      % Representation of the substitution obtained by 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 a term substitution.
* TermSubst.      % Representation of the substitution obtained by restricting
                  % this term substitution to the free variables in this term.

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.
* 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.
* VarTyping.      % Representation of the variable typing containing 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 a binding in this
                  % substitution.
* Type.           % Representation of the type to which this parameter 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 a binding in this
                  % substitution.
* Term.           % Representation of the term to which this variable is bound
                  % in this substitution.

DELAY      BindingInTermSubst(x,_,_) UNTIL GROUND(x).


PREDICATE  BindingInVarTyping :

  VarTyping       % Representation of a variable typing.
* Term            % Representation of a variable in a binding in this variable
                  % typing.
* Type.           % Representation of the type to which this variable 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.
* Type            % Representation of a type.
* TypeSubst.      % Representation of the type substitution obtained from the
                  % first argument by deleting the binding of this parameter
                  % to this type.

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

```
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.

DELAY      DelBindingInTermSubst(x,_,_,_) UNTIL GROUND(x).


PREDICATE  DelBindingInVarTyping :

  VarTyping       % Representation of a variable typing.
* Term            % Representation of a variable.
* Type            % Representation of a type.
* VarTyping.      % Representation of the variable typing obtained from the
                  % first argument by deleting the binding of this variable
                  % to this type.

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 :

  Term            % Representation of a term.
* Term            % Representation of a term.
* TermSubst       % Representation of a term substitution.
* TermSubst.      % Representation of the term substitution obtained by
```

```
                  % composing the term substitution in the third argument with
                  % a specific, unique mgu for the terms which are obtained by
                  % applying the term substitution in the third argument to
                  % the terms in the first two arguments. (The mgu binds a
                  % variable in the first argument to a variable in the second
                  % argument, not the other way around.)

DELAY      UnifyTerms(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).


PREDICATE  UnifyAtoms :

  Formula         % Representation of an atom.
* Formula         % Representation of an atom.
* TermSubst       % Representation of a term substitution.
* TermSubst.      % Representation of the term substitution obtained by
                  % composing the term substitution in the third argument with
                  % a specific, unique mgu for the atoms which are obtained by
                  % applying the term substitution in the third argument to
                  % the atoms in the first two arguments. (The mgu binds a
                  % variable in the first argument to a variable in the second
                  % argument, not the other way around.)

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 variables of the terms in the second argument by a
                  % specific, unique term substitution such that they become
```

```
                 % distinct from the 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.
* 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 normal 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 normal 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.)
* Formula.        % Representation of the derived resultant obtained from this
                  % resultant and this statement using this mgu.

DELAY      Derive(x,y,z,u,v,_,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z) &
                                                          GROUND(u) & GROUND(v).


PREDICATE  ResolveAll :

  Formula         % Representation of an atom.
* List(Formula)   % List of representations of normal statements.
* Integer         % A non-negative integer.
* Integer         % A non-negative integer.
* Integer         % A non-negative integer.
* Integer         % A non-negative 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 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 corresponding conjunctions
                    % of literals obtained by resolving the negation of the atom
                    % in the first argument with the substitution in the seventh
                    % argument applied and each renamed statement from the second
                    % argument and then negating the resolvents so obtained.

DELAY      ResolveAll(x,y,z,_,u,_,v,_,_) UNTIL GROUND(x) & GROUND(y) &
                                             GROUND(z) & GROUND(u) & GROUND(v).
```

13.12 Programs

```
EXPORT     Programs.


% Module containing predicates provided by the ground representation of (object)
% Goedel programs.


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.
* Type.            % Representation of a type in the flat language of this
                   % program.

DELAY      TypeInProgram(x,y) UNTIL GROUND(x) & GROUND(y).


PREDICATE  TermInProgram :

  Program          % Representation of a program.
* VarTyping        % Representation of a variable typing in the flat language of
                   % this program.
* Term             % Representation of a term in the flat language of this
                   % program.
* Type             % Representation of the type of this term with respect 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 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.
* Formula         % Representation of a formula (that is, standard 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 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.
* ModulePart      % Representation of a part keyword of this module.
* 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.
* Term            % Representation of a term 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.
* Type              % Representation of the type of this term with respect to this
                    % variable typing.
* VarTyping.        % Representation of the variable typing obtained by combining
                    % the variable typing in the fifth argument with the types of
                    % all variables occurring in the term.

DELAY      TermInModule(x,y,z,u,v,_,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z) &
                                                           GROUND(u) & GROUND(v).


PREDICATE  FormulaInModule :

  Program           % Representation of a program.
* String            % Name of a module in this program.
* ModulePart        % Representation of a part keyword of this module.
* 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 (that is, 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.
* 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.

DELAY      FormulaInModule(x,y,z,u,v,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z)
                                                         & GROUND(u) & GROUND(v).


PREDICATE  StringToProgramType :

  Program           % Representation of a program.
* String            % Name of a module in this program.
* String            % A string.
* List(Type).       % List (in a definite order) of representations of types in
                    % the flat language of this module wrt this program whose
                    % string representation is the third argument.

DELAY      StringToProgramType(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).


PREDICATE  StringToProgramTerm :
  Program         % Representation of a 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(Formula).  % List (in a definite order) of representations of formulas
                  % (that is, 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.
* String          % Name of a module in this program.
* Type            % Representation of a type.
* String.         % The string representation of this type. (Subtypes of the
                  % type not in the flat language of the module do not appear.)

DELAY      ProgramTypeToString(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).


PREDICATE  ProgramTermToString :

  Program         % Representation of a program.
* String          % Name of a module in this program.
* Term            % Representation of a term.
* String.         % The string representation of this term. (Subterms 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.
* Formula         % Representation of a formula (that is, standard body or
                  % standard resultant).
* String.         % The string representation of this formula. (Terms or atoms
                  % 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.
* Name.           % Representation of the corresponding flat name of 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.
* Name.           % Representation of the corresponding flat name of this
                  % constructor.

DELAY      ProgramConstructorName(x,y,z,u,v) UNTIL GROUND(x) &
                             ((GROUND(y) & GROUND(z) & GROUND(u)) \/ GROUND(v)).

PREDICATE  ProgramConstantName :

  Program         % Representation of a program.
* String          % Name of a module in this program.
* String          % Name of a constant.
* Name.           % Representation of the corresponding flat name of this
                  % constant.

DELAY      ProgramConstantName(x,y,z,u) UNTIL GROUND(x) &
                                         ((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.
* Integer         % Arity of this function.
* Name.           % Representation of the corresponding flat name of this
                  % function.

DELAY      ProgramFunctionName(x,y,z,u,v) UNTIL GROUND(x) &
                  ((GROUND(y) & GROUND(z) & GROUND(u)) \/ GROUND(v)).


PREDICATE  ProgramPropositionName :

  Program         % Representation of a program.
* String          % Name of a module in this program.
* String          % Name of a proposition.
* Name.           % Representation of the corresponding flat name of 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.
* Integer         % Arity of this predicate.
* Name.           % Representation of the corresponding flat name of this
                  % predicate.

DELAY      ProgramPredicateName(x,y,z,u,v) UNTIL GROUND(x) &
                  ((GROUND(y) & GROUND(z) & GROUND(u)) \/ GROUND(v)).


PREDICATE  MainModuleInProgram :

  Program         % Representation of a program.
* String.         % Name of the main module in this program.

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

```
PREDICATE  ModuleInProgram :

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

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


PREDICATE  DeclaredInClosedModule :

  Program         % Representation of a 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.

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


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,y,_) UNTIL GROUND(x) & GROUND(y).
```

```
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.
* List(Formula)   % List (in a definite order) of representations of 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.
* Condition       % Representation of a DELAY declaration condition, Cond2.
* Condition.      % Representation of the condition Cond1 & Cond2.
```

```
PREDICATE  OrCondition :

  Condition       % Representation of a DELAY declaration condition, Cond1.
* Condition       % Representation of a DELAY declaration condition, Cond2.
* Condition.      % Representation of the 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  StringCondition :

  String          % The string representation of a condition.
* Condition.      % Representation of this condition.

DELAY      StringCondition(x,y) UNTIL GROUND(x) \/ GROUND(y).


PREDICATE  BaseInModule :

  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 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.
* 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 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.
* 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 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 which cannot
                   % be LOCAL if this module is closed.
* Name             % Representation of the flat name of a function 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.
* Type             % Representation of the range type of this function.
* String.          % Name of module in which this function is declared.

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

```
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.

DELAY      PropositionInModule(x,_,_,_,_) UNTIL GROUND(x).


PREDICATE  PredicateInModule :

  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 predicate accessible to
                  % this part of this module.
* PredicateInd    % Representation of the indicator for this predicate.
* List(Type)      % List of the representations of the types for this predicate.
* String.         % Name of module in which this predicate is declared.

DELAY      PredicateInModule(x,_,_,_,_,_,_) UNTIL GROUND(x).


PREDICATE  DelayInModule :

  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.
* Formula         % Representation of the Atom part of a DELAY declaration
                  % appearing in this part of this module.
* Condition.      % Representation of the Cond part of this DELAY declaration.

DELAY      DelayInModule(x,_,_,_,_) UNTIL GROUND(x).


PREDICATE  ImportInModule :

  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.
* String.         % Name of a module appearing in an IMPORT declaration in this
                  % part of this module.

DELAY      ImportInModule(x,_,_,_) UNTIL GROUND(x).


PREDICATE  LiftInModule :

  Program         % Representation of a program.
* String          % Name of an open module in this program.
* String.         % Name of a module appearing in a LIFT declaration 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.

DELAY      NewProgram(x,_) UNTIL GROUND(x).


PREDICATE  InsertProgramBase :

  Program         % Representation of a program.
* String          % Name of an open 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.

DELAY      InsertProgramBase(x,y,z,u,_) UNTIL GROUND(x) & GROUND(y) &
                                                  GROUND(z) & GROUND(u).
```

```
PREDICATE  DeleteProgramBase :

  Program         % Representation of a program.
* String          % Name of an open module in this program.
* 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.

DELAY      DeleteProgramBase(x,y,z,_,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).


PREDICATE  InsertProgramConstructor :

  Program         % Representation of a program.
* String          % Name of an open module in this program.
* 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.
* String          % Name of an open module in this program.
* ModulePart      % Representation of a part keyword of this module.
* Name            % Representation of the flat name of a constructor 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 does not contain the
                  % declaration of this constructor in this part of this module.

DELAY      DeleteProgramConstructor(x,y,z,_,_,_) UNTIL GROUND(x) & GROUND(y) &
                                                                  GROUND(z).
```

```
PREDICATE  InsertProgramConstant :

  Program         % Representation of a program.
* String          % Name of an open module in this program.
* 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.
* Type            % 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.

DELAY      InsertProgramConstant(x,y,z,u,v,_) UNTIL GROUND(x) & GROUND(y) &
                                            GROUND(z) & GROUND(u) & GROUND(v).


PREDICATE  DeleteProgramConstant :

  Program         % Representation of a program.
* String          % Name of an open module in this program.
* ModulePart      % Representation of a part keyword of this module.
* Name            % Representation of the flat name of a constant declared in
                  % this part of this module.
* Type            % 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.

DELAY      DeleteProgramConstant(x,y,z,_,_,_) UNTIL GROUND(x) & GROUND(y) &
                                                                GROUND(z).


PREDICATE  InsertProgramFunction :

  Program         % Representation of a program.
* String          % Name of an open 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     % Representation of the indicator for this function.
* List(Type)      % List of the representations of the domain types of this
                  % function.
* Type            % Representation of the range type of this function.
* Program.        % Representation of a program which differs from the program
                  % in the first argument only in that it also contains the
                  % declaration of this function in this part of this module.
```

```
DELAY      InsertProgramFunction(x,y,z,u,v,w,r,_) UNTIL GROUND(x) & GROUND(y) &
                      GROUND(z) & GROUND(u) & GROUND(v) & GROUND(w) & GROUND(r).


PREDICATE  DeleteProgramFunction :

  Program          % Representation of a program.
* String           % Name of an open module in this program.
* ModulePart       % Representation of a part keyword of this module.
* Name             % Representation of the flat name of a function declared in
                   % 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.
* Type             % Representation of the range type of this function.
* 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 function in this part of this module.

DELAY      DeleteProgramFunction(x,y,z,_,_,_,_,_) UNTIL GROUND(x) & GROUND(y) &
                                                                      GROUND(z).


PREDICATE  InsertProgramProposition :

  Program          % Representation of a program.
* String           % Name of an open module in this program.
* 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.

DELAY      InsertProgramProposition(x,y,z,u,_) UNTIL GROUND(x) & GROUND(y) &
                                                          GROUND(z) & GROUND(u).


PREDICATE  DeleteProgramProposition :

  Program          % Representation of a program.
* String           % Name of an open module in this program.
* 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.

DELAY      DeleteProgramProposition(x,y,z,_,_) UNTIL GROUND(x) & GROUND(y) &
                                                                    GROUND(z).


PREDICATE  InsertProgramPredicate :

  Program         % Representation of a program.
* String          % Name of an open module in this program.
* 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.
* PredicateInd    % Representation of the indicator for this predicate.
* List(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.

DELAY      InsertProgramPredicate(x,y,z,u,v,w,_) UNTIL GROUND(x) & GROUND(y) &
                                  GROUND(z) & GROUND(u) & GROUND(v) & GROUND(w).


PREDICATE  DeleteProgramPredicate :

  Program         % Representation of a program.
* String          % Name of an open 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.
* PredicateInd    % Representation of the indicator for this predicate.
* List(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 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 module in this program.
* ModulePart      % Representation of a part keyword of this module.
* Formula         % Representation of a DELAY Atom part.
* Condition       % Representation of a DELAY Cond part.
* Program.        % Representation of a program which differs from the program
                  % in the first argument only in that it also contains in this
                  % part of this module the DELAY declaration consisting of the
                  % Atom part in the fourth argument and the Cond part in the
                  % fifth argument.

DELAY      InsertDelay(x,y,z,u,v,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z) &
                                                       GROUND(u) & GROUND(v).


PREDICATE  DeleteDelay :

  Program         % Representation of a program.
* String          % Name of an open 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.
* Program.        % Representation of a program which differs from the 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 module in this program.
* Formula         % Representation of a statement in the flat language of this
                  % module wrt this program.
* Program.        % Representation of a program which differs from the program
                  % in the first argument only in that it also contains this
                  % statement in this module.

DELAY      InsertStatement(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).


PREDICATE  DeleteStatement :

  Program         % Representation of a program.
* String          % Name of an open 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.

DELAY      DeleteStatement(x,y,_,_) UNTIL GROUND(x) & GROUND(y).


PREDICATE  InsertProgramImport :

  Program         % Representation of a program.
* String          % Name of an open module in this program.
* 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 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.
* String          % Name of an open module in this program.
* 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 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 also contains in the local
                   % part of this module the LIFT declaration importing the
                   % module in the third argument.
                   %
                   % 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.

DELAY      InsertProgramLift(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).


PREDICATE  DeleteProgramLift :

  Program          % Representation of a program.
* String           % Name of an open 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 :

  Program         % Representation of a program.
* Formula         % Representation of the body of a goal in the flat 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 argument and the flat form of this
                  % program or, if the computation flounders, 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 answer are greater than or equal to the third
                  % argument and strictly less than the fourth argument.
* Formula.        % Representation of the body of the last goal in the
                  % 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 :

  Program          % Representation of a 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.

DELAY      SucceedAll(x,y,_) UNTIL GROUND(x) & GROUND(y).


PREDICATE  ComputeAll :

  Program          % Representation of a program.
* Formula          % Representation of the body of a goal in the flat 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.
* List(Formula).  % List of representations of the bodies of the last goals in
                   % each of the corresponding derivations (which are empty if
                   % the derivation succeeded and are 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.
* Formula.         % Representation of the body of a goal in the flat language of
                   % this program such that this goal and the flat form of this
                   % program have a finitely failed search tree.
```

```
DELAY      Fail(x,y) UNTIL GROUND(x) & GROUND(y).
```

13.13 Scripts

```
EXPORT     Scripts.


% Module containing predicates provided by the ground representation of (object)
% Goedel scripts.


IMPORT     Programs.


BASE       Script.         % Type of a term representing a script.


PREDICATE  ProgramToScript :

  Program          % Representation of a program
* Script.          % Representation of the associated script for this program.

DELAY      ProgramToScript(x,_) UNTIL GROUND(x).


PREDICATE  TypeInScript :

  Script           % Representation of a script.
* Type.            % Representation of a type in the language of this script.

DELAY      TypeInScript(x,y) UNTIL GROUND(x) & GROUND(y).


PREDICATE  TermInScript :

  Script           % Representation of a script.
* VarTyping        % Representation of a variable typing in the language of
                   % this script.
* Term             % Representation of a term in the language of this script.
* Type             % Representation of the type of this term with respect 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 variables occurring in the term.

DELAY      TermInScript(x,y,z,_,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).
```

```
PREDICATE  FormulaInScript :

  Script          % Representation of a script.
* VarTyping       % Representation of a variable typing in the language
                  % of this script.
* Formula         % Representation of a formula (that is, standard body or
                  % standard resultant) in the language of this script.
* 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      FormulaInScript(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).


PREDICATE  BaseInScript :

  Script          % Representation of a script.
* Name.           % Representation of the flat name of a base in the open
                  % language of this script.

DELAY      BaseInScript(x,_) UNTIL GROUND(x).


PREDICATE  ConstructorInScript :

  Script          % Representation of a script.
* Name            % Representation of the flat name of a constructor in the open
                  % language of this script.
* Integer.        % Arity of this constructor.

DELAY      ConstructorInScript(x,_,_) UNTIL GROUND(x).


PREDICATE  ConstantInScript :

  Script          % Representation of a script.
* Name            % Representation of the flat name of a constant in the open
                  % language of this script.
* Type.           % Representation of the type of this constant.

DELAY      ConstantInScript(x,_,_) UNTIL GROUND(x).


PREDICATE  FunctionInScript :

  Script          % Representation of a script.
* Name            % Representation of the flat name of a function in the open
```

```
                  % language of this script.
* FunctionInd     % Representation of the indicator for this function.
* List(Type)      % List of the representations of the domain types of this
                  % function.
* Type.           % Representation of the range type of this function.

DELAY      FunctionInScript(x,_,_,_,_) UNTIL GROUND(x).

PREDICATE  PropositionInScript :

  Script          % Representation of a script.
* Name.           % Representation of the flat name of a proposition in the open
                  % language of this script.

DELAY      PropositionInScript(x,_) UNTIL GROUND(x).

PREDICATE  PredicateInScript :

  Script          % Representation of a script.
* Name            % Representation of the flat name of a predicate in the open
                  % language of this script.
* PredicateInd    % Representation of the indicator for this predicate.
* List(Type).     % List of the representations of the types for this predicate.

DELAY      PredicateInScript(x,_,_,_) UNTIL GROUND(x).

PREDICATE  StatementInScript :

  Script          % Representation of a script.
* Formula.        % Representation of a statement in the open part of this
                  % script.

DELAY      StatementInScript(x,_) UNTIL GROUND(x).

PREDICATE  StatementMatchAtom :

  Script          % Representation of a script.
* Formula         % Representation of an atom in the language of this script,
                  % whose proposition or predicate symbol is in the open
                  % language of this script.
* Formula.        % Representation of a statement in the open part of this
                  % script 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  DefinitionInScript :

  Script          % Representation of a script.
* Name            % Representation of the flat name of a proposition or
                  % predicate in the open language of this script.
* List(Formula).  % List (in a definite order) of representations of statements
                  % in the definition of this proposition or predicate.

DELAY      DefinitionInScript(x,_,_) UNTIL GROUND(x).


PREDICATE  ControlInScript :

  Script          % Representation of a script.
* Name            % Representation of the flat name of a predicate in the
                  % open language of this script.
* List(Formula)   % List (in a definite order) of representations of 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      ControlInScript(x,_,_,_) UNTIL GROUND(x).


PREDICATE  DelayInScript :

  Script          % Representation of a script.
* Formula         % Representation of the Atom part of a DELAY declaration
                  % appearing in the open part of this script.
* Condition.      % Representation of the Cond part of this DELAY declaration.

DELAY      DelayInScript(x,_,_) UNTIL GROUND(x).


PREDICATE  InsertScriptProposition :

  Script          % Representation of a script.
* Name            % Representation of the flat name of a proposition whose
                  % module name component is not the name of a closed module
                  % and which is not declared in this script.
* Script.         % Representation of a script which differs from the script
                  % in the first argument only in that it also contains the
                  % declaration of this proposition in its open part.

DELAY      InsertScriptProposition(x,y,_) UNTIL GROUND(x) & GROUND(y).
```

```
PREDICATE  DeleteScriptProposition :

  Script          % Representation of a script.
* Name            % Representation of the flat name of a proposition whose
                  % module name component is not the name of a closed module
                  % and which is declared in the open part of this script.
* Script.         % Representation of a script which differs from the script
                  % in the first argument only in that it does not contain the
                  % declaration of this proposition in its open part.

DELAY      DeleteScriptProposition(x,_,_) UNTIL GROUND(x).


PREDICATE  InsertScriptPredicate :

  Script          % Representation of a script.
* Name            % Representation of the flat name of a predicate whose
                  % module name component is not the name of a closed module
                  % and which is not declared in this script.
* PredicateInd    % Representation of the indicator for this predicate.
* List(Type)      % List of the representations of the types of this predicate.
* Script.         % Representation of a script which differs from the script
                  % in the first argument only in that it also contains the
                  % declaration of this predicate in its open part.

DELAY      InsertScriptPredicate(x,y,z,u,_) UNTIL GROUND(x) & GROUND(y) &
                                                  GROUND(z) & GROUND(u).


PREDICATE  DeleteScriptPredicate :

  Script          % Representation of a script.
* Name            % Representation of the flat name of a predicate whose
                  % module name component is not the name of a closed module
                  % and which is declared in the open part of this script.
* PredicateInd    % Representation of the indicator for this predicate.
* List(Type)      % List of the representations of the types of this predicate.
* Script.         % Representation of a script which differs from the script
                  % in the first argument only in that it does not contain the
                  % declaration of this predicate in its open part.

DELAY      DeleteScriptPredicate(x,_,_,_,_) UNTIL GROUND(x).


PREDICATE  InsertDelay :
  Script          % Representation of a script.
* Formula         % Representation of a DELAY Atom part.
* Condition       % Representation of a DELAY Cond part.
* Script.         % Representation of a script which differs from the script
                  % in the first argument only in that it also contains in its
                  % open part the DELAY declaration consisting of the Atom part
                  % in the second argument and the Cond part in the third
                  % argument.

DELAY      InsertDelay(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE  DeleteDelay :

  Script          % Representation of a script.
* Formula         % Representation of the Atom part of a DELAY declaration in
                  % the open part of this script.
* Condition       % Representation of the Cond part of this DELAY declaration.
* Script.         % Representation of a script which differs from the script
                  % in the first argument only in that it does not contain this
                  % DELAY declaration in its open part.

DELAY      DeleteDelay(x,_,_,_) UNTIL GROUND(x).

PREDICATE  InsertStatement :

  Script          % Representation of a script.
* Formula         % Representation of a statement in the open language of this
                  % script.
* Script.         % Representation of a script which differs from the script
                  % in the first argument only in that it also contains this
                  % statement in the open part of the script.

DELAY      InsertStatement(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE  DeleteStatement :

  Script          % Representation of a script.
* Formula         % Representation of a statement in the open language of this
                  % script in this script.
* Script.         % Representation of a script which differs from the script
                  % in the first argument only in that it does not contain this
                  % statement in the open part of the script.

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

13.14 Theories

```
EXPORT     Theories.


% Module containing predicates provided by the ground representation of (object)
% theories.


IMPORT     Syntax.


BASE       Theory.          % Type of a term representing a theory.


PREDICATE  TypeInTheory :

  Theory          % Representation of a theory.
* Type.           % Representation of a type in the flat language of this
                  % theory.

DELAY      TypeInTheory(x,y) UNTIL GROUND(x) & GROUND(y).


PREDICATE  TermInTheory :

  Theory          % Representation of a theory.
* VarTyping       % Representation of a variable typing in the flat language of
                  % this theory.
* Term            % Representation of a term in the flat language of this
                  % theory.
* Type            % Representation of the type of this term with respect 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 variables occurring in the term.

DELAY      TermInTheory(x,y,z,_,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).


PREDICATE  FormulaInTheory :

  Theory          % Representation of a theory.
* VarTyping       % Representation of a variable typing in the flat language
```

```
                  % of this theory.
* Formula         % Representation of a formula in the flat language of this
                  % theory.
* 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      FormulaInTheory(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).


PREDICATE  StringToTheoryType :

  Theory          % Representation of a theory.
* String          % A string.
* List(Type).     % List (in a definite order) of representations of types in
                  % the language of this theory whose string representation is
                  % the second argument.

DELAY      StringToTheoryType(x,y,_) UNTIL GROUND(x) & GROUND(y).


PREDICATE  StringToTheoryTerm :

  Theory          % Representation of a theory.
* String          % A string.
* List(Term).     % List (in a definite order) of representations of terms in
                  % the language of this theory whose string representation is
                  % the second argument.

DELAY      StringToTheoryTerm(x,y,_) UNTIL GROUND(x) & GROUND(y).


PREDICATE  StringToTheoryFormula :

  Theory          % Representation of a theory.
* String          % A string.
* List(Formula).  % List (in a definite order) of representations of formulas
                  % in the language of this theory whose string representation
                  % is the second argument.

DELAY      StringToTheoryFormula(x,y,_) UNTIL GROUND(x) & GROUND(y).


PREDICATE  TheoryTypeToString :

  Theory          % Representation of a theory.
* Type            % Representation of a type.
* String.         % The string representation of this type. (Subtypes of the
                  % type not in the language of this theory do not appear.)

DELAY      TheoryTypeToString(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE  TheoryTermToString :

  Theory          % Representation of a theory.
* Term            % Representation of a term.
* String.         % The string representation of this term. (Subterms of the
                  % term not in the language of this theory do not appear.)

DELAY      TheoryTermToString(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE  TheoryFormulaToString :

  Theory          % Representation of a theory.
* Formula         % Representation of a formula.
* String.         % The string representation of this formula. (Terms or atoms
                  % of the formula not in the language of this theory do not
                  % appear.)

DELAY      TheoryFormulaToString(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE  TheoryBaseName :

  Theory          % Representation of a theory.
* String          % Name of this theory.
* String          % Name of a base.
* Name.           % Representation of the corresponding flat name of this base.

DELAY      TheoryBaseName(x,y,z,u) UNTIL GROUND(x) &
                              ((GROUND(y) & GROUND(z)) \/ GROUND(u)).

PREDICATE  TheoryConstructorName :

  Theory          % Representation of a theory.
* String          % Name of this theory.
* String          % Name of a constructor.
* Integer         % Arity of this constructor.
* Name.           % Representation of the corresponding flat name of this
                  % constructor.

DELAY      TheoryConstructorName(x,y,z,u,v) UNTIL GROUND(x) &
                        ((GROUND(y) & GROUND(z) & GROUND(u)) \/ GROUND(v)).
```

```
PREDICATE  TheoryConstantName :

  Theory           % Representation of a theory.
* String           % Name of this theory.
* String           % Name of a constant.
* Name.            % Representation of the corresponding flat name of this
                   % constant.

DELAY      TheoryConstantName(x,y,z,u) UNTIL GROUND(x) &
                                   ((GROUND(y) & GROUND(z)) \/ GROUND(u)).

PREDICATE  TheoryFunctionName :

  Theory           % Representation of a theory.
* String           % Name of this theory.
* String           % Name of a function.
* Integer          % Arity of this function.
* Name.            % Representation of the corresponding flat name of this
                   % function.

DELAY      TheoryFunctionName(x,y,z,u,v) UNTIL GROUND(x) &
                             ((GROUND(y) & GROUND(z) & GROUND(u)) \/ GROUND(v)).

PREDICATE  TheoryPropositionName :

  Theory           % Representation of a theory.
* String           % Name of this theory.
* String           % Name of a proposition.
* Name.            % Representation of the corresponding flat name of this
                   % proposition.

DELAY      TheoryPropositionName(x,y,z,u) UNTIL GROUND(x) &
                                   ((GROUND(y) & GROUND(z)) \/ GROUND(u)).

PREDICATE  TheoryPredicateName :

  Theory           % Representation of a theory.
* String           % Name of this theory.
* String           % Name of a predicate.
* Integer          % Arity of this predicate.
* Name.            % Representation of the corresponding flat name of this
                   % predicate.

DELAY      TheoryPredicateName(x,y,z,u,v) UNTIL GROUND(x) &
                             ((GROUND(y) & GROUND(z) & GROUND(u)) \/ GROUND(v)).
```

```
PREDICATE  AxiomInTheory :

  Theory          % Representation of a theory.
* Formula.        % Representation of an axiom in this theory.

DELAY      AxiomInTheory(x,_) UNTIL GROUND(x).


PREDICATE  BaseInTheory :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a base accessible to this
                  % theory.
* String.         % Name of theory or module in which this base is declared.

DELAY      BaseInTheory(x,_,_) UNTIL GROUND(x).


PREDICATE  ConstructorInTheory :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a constructor accessible
                  % to this theory.
* Integer         % Arity of this constructor.
* String.         % Name of theory or module in which this constructor is
                  % declared.

DELAY      ConstructorInTheory(x,_,_,_) UNTIL GROUND(x).


PREDICATE  ConstantInTheory :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a constant accessible to
                  % this theory.
* Type            % Representation of the type of this constant.
* String.         % Name of theory or module in which this constant is declared.

DELAY      ConstantInTheory(x,_,_,_) UNTIL GROUND(x).


PREDICATE  FunctionInTheory :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a function accessible to
                  % this theory.
* Integer         % Arity of this function.
* FunctionInd     % Indicator for this function.
* List(Type)      % List of the representations of the domain types of this
                  % function.
* Type            % Representation of the range type of this function.
* String.         % Name of theory or module in which this function is
                  % declared.

DELAY      FunctionInTheory(x,_,_,_,_,_,_) UNTIL GROUND(x).


PREDICATE  PropositionInTheory :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a proposition accessible
                  % to this theory.
* String.         % Name of theory or module in which this proposition is
                  % declared.

DELAY      PropositionInTheory(x,_,_) UNTIL GROUND(x).


PREDICATE  PredicateInTheory :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a predicate accessible to
                  % this theory.
* Integer         % Arity of this predicate.
* PredicateInd    % Indicator for this predicate.
* List(Type)      % List of the representations of the types for this predicate.
* String.         % Name of theory or module in which this predicate is
                  % declared.

DELAY      PredicateInTheory(x,_,_,_,_,_) UNTIL GROUND(x).


PREDICATE  ImportInTheory :

  Theory          % Representation of a theory.
* String.         % Name of a module appearing in an IMPORT declaration in this
                  % theory.

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

```
PREDICATE  NewTheory :

  String          % A string.
* Theory.         % Representation of an empty theory with this string as the
                  % name of the theory.

DELAY      NewTheory(x,_) UNTIL GROUND(x).


PREDICATE  InsertTheoryBase :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a base not declared in
                  % this theory.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it also contains the
                  % declaration of this base.

DELAY      InsertTheoryBase(x,y,_) UNTIL GROUND(x) & GROUND(y).


PREDICATE  DeleteTheoryBase :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a base declared in this
                  % theory.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it does not contain the
                  % declaration of this base.

DELAY      DeleteTheoryBase(x,_,_) UNTIL GROUND(x).


PREDICATE  InsertTheoryConstructor :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a constructor not
                  % declared in this theory.
* Integer         % Arity of this constructor.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it also contains the
                  % declaration of this constructor.

DELAY      InsertTheoryConstructor(x,y,z,_) UNTIL GROUND(x) & GROUND(y) &
                                                             GROUND(z).
```

```
PREDICATE  DeleteTheoryConstructor :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a constructor declared
                  % in this theory.
* Integer         % Arity of this constructor.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it does not contain the
                  % declaration of this constructor.

DELAY      DeleteTheoryConstructor(x,_,_,_) UNTIL GROUND(x).


PREDICATE  InsertTheoryConstant :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a constant not declared
                  % in this theory.
* Type            % Representation of the type of this constant.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it also contains the
                  % declaration of this constant.

DELAY      InsertTheoryConstant(x,y,z,_) UNTIL GROUND(x) & GROUND(y) &
                                                            GROUND(z).


PREDICATE  DeleteTheoryConstant :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a constant declared in
                  % this theory.
* Type            % Representation of the type of this constant.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it does not contain the
                  % declaration of this constant.

DELAY      DeleteTheoryConstant(x,_,_,_) UNTIL GROUND(x).


PREDICATE  InsertTheoryFunction :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a function not declared
                  % in this theory.
* FunctionInd     % Indicator for this function.
* List(Type)      % List of the representations of the domain types of this
                  % function.
* Type            % Representation of the range type of this function.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it also contains the
                  % declaration of this function.

DELAY      InsertTheoryFunction(x,y,z,u,v,_) UNTIL GROUND(x) & GROUND(y) &
                                              GROUND(z) & GROUND(u) & GROUND(v).


PREDICATE  DeleteTheoryFunction :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a function declared in
                  % this theory.
* FunctionInd     % Indicator for this function.
* List(Type)      % List of the representations of the domain types of this
                  % function.
* Type            % Representation of the range type of this function.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it does not contain the
                  % declaration of this function.

DELAY      DeleteTheoryFunction(x,_,_,_,_,_) UNTIL GROUND(x).


PREDICATE  InsertTheoryProposition :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a proposition not
                  % declared in this theory.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it also contains the
                  % declaration of this proposition.

DELAY      InsertTheoryProposition(x,y,_) UNTIL GROUND(x) & GROUND(y).


PREDICATE  DeleteTheoryProposition :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a proposition declared
                  % in this theory.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it does not contain the
                  % declaration of this proposition.

DELAY      DeleteTheoryProposition(x,_,_) UNTIL GROUND(x).


PREDICATE  InsertTheoryPredicate :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a predicate not declared
                  % in this theory.
* PredicateInd    % Indicator for this predicate.
* List(Type)      % List of the representations of the types of this predicate.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it also contains the
                  % declaration of this predicate.

DELAY      InsertTheoryPredicate(x,y,z,u,_) UNTIL GROUND(x) & GROUND(y) &
                                                       GROUND(z) & GROUND(u).


PREDICATE  DeleteTheoryPredicate :

  Theory          % Representation of a theory.
* Name            % Representation of the flat name of a predicate declared in
                  % this theory.
* PredicateInd    % Indicator for this predicate.
* List(Type)      % List of the representations of the types of this predicate.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it does not contain the
                  % declaration of this predicate.

DELAY      DeleteTheoryPredicate(x,_,_,_,_) UNTIL GROUND(x).


PREDICATE  InsertTheoryImport :

  Theory          % Representation of a theory.
* String          % Name of a system module not imported into the theory.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it also contains the
                  % IMPORT declaration importing the module in the second
                  % argument.

DELAY      InsertTheoryImport(x,y,_) UNTIL GROUND(x) & GROUND(y).
```

```
PREDICATE  DeleteTheoryImport :

  Theory          % Representation of a theory.
* String          % Name of a system module imported into the theory.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it does not contain the
                  % IMPORT declaration importing the module in the second
                  % argument.

DELAY      DeleteTheoryImport(x,_,_) UNTIL GROUND(x).


PREDICATE  InsertAxiom :

  Theory          % Representation of a theory.
* Formula         % Representation of a formula in the flat language of this
                  % theory.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it also contains this
                  % formula as an axiom.

DELAY      InsertAxiom(x,y,_) UNTIL GROUND(x) & GROUND(y).


PREDICATE  DeleteAxiom :

  Theory          % Representation of a theory.
* Formula         % Representation of a formula in the flat language of this
                  % theory in this theory.
* Theory.         % Representation of a theory which differs from the theory
                  % in the first argument only in that it does not contain this
                  % formula as an axiom.

DELAY      DeleteAxiom(x,_,_) UNTIL GROUND(x).


PREDICATE  Prove :

  Theory          % Representation of a theory.
* Formula.        % Representation of a theorem of this theory.

DELAY      Prove(x,y) UNTIL GROUND(x) & GROUND(y).
```

13.15 IO

```
EXPORT     IO.


% Module providing basic input/output facilities.


IMPORT     Strings.


BASE       InputStream, OutputStream.
%
% Stream types.


BASE       ResultOfFind.
%
% Type of result indicating the success or failure of an attempt to open a
% stream.


CONSTANT   StdIn : InputStream.
%
% Built-in stream corresponding to UNIX file descriptor 0.


CONSTANT   StdOut, StdErr : OutputStream.
%
% Built-in stream corresponding to UNIX file descriptors 1 and 2.


CONSTANT   NotFound : ResultOfFind.
%
% Indicates stream could not be opened.

FUNCTION   In : InputStream -> ResultOfFind.
%
% Indicates input stream was opened successfully.

FUNCTION   Out : OutputStream -> ResultOfFind.
%
% Indicates output stream was opened successfully.
```

```
PREDICATE  FindInput :

  String          % Name of file.
* ResultOfFind.   % In(stream), where stream is the new input stream
                  % pointing to the beginning of the file, if the attempt
                  % to open the file was successful;
                  % otherwise, NotFound, if file could not be opened.

DELAY      FindInput(x,_) UNTIL GROUND(x).


PREDICATE  FindOutput :

  String          % Name of file.
* ResultOfFind.   % Out(stream), where stream is the new output stream
                  % pointing to the beginning of the file, if the attempt
                  % to open the file was successful;
                  % otherwise, NotFound, if file could not be opened.

% If the file already exists it is truncated, otherwise an empty file with the
% given name is created.

DELAY      FindOutput(x,_) UNTIL GROUND(x).


PREDICATE  FindUpdate :

  String          % Name of file.
* ResultOfFind.   % Out(stream), where stream is the new output stream
                  % pointing to the end of the file, if the attempt
                  % to open the file was successful;
                  % otherwise, NotFound, if file could not be opened.

% The file pointer is set to the end of the file if it already exists.
% Otherwise, an empty file with the given name is created.

DELAY      FindUpdate(x,_) UNTIL GROUND(x).

% The previous three predicates are guaranteed to succeed if the file argument
% is instantiated and the result argument is uninstantiated. If the result
% argument is NotFound, subsequent read, write and close operations on the
% stream will fail.


PREDICATE  EndInput :
  InputStream.    % An open input stream to be closed.

DELAY      EndInput(x) UNTIL GROUND(x).

PREDICATE  EndOutput :

   OutputStream.  % An open output stream to be closed.

DELAY      EndOutput(x) UNTIL GROUND(x).

% Closing one of the fixed streams StdIn, StdOut, or StdErr will succeed, but
% the stream is immediately reopened.

PREDICATE  Get :

  InputStream     % An open input stream.
* Integer.        % The ASCII code of the next character read from the stream
                  % or -1 if the end of the file has been reached.

DELAY      Get(x,_) UNTIL GROUND(x).

PREDICATE  ReadChar :

  InputStream     % An open input stream.
* String.         % A string of length 1 containing the next character read from
                  % the stream or the empty string if the end of the file has
                  % been reached.

DELAY      ReadChar(x,_) UNTIL GROUND(x).

PREDICATE  Put :

  OutputStream    % An open output stream.
* Integer.        % An integer between 0 and 127. The character which has this
                  % ASCII code is written to the stream.

DELAY      Put(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE  WriteString :

  OutputStream    % An open output stream.
* String.         % A string of characters, which are written to the stream.

DELAY      WriteString(x,y) UNTIL GROUND(x) & GROUND(y).
```

```
PREDICATE  NewLine :

  OutputStream.   % An open output stream, to which a newline character is
                  % written.

DELAY      NewLine(x) UNTIL GROUND(x).


PREDICATE  Flush :

  OutputStream.   % An open output stream.

% Calling Flush forces any characters that have been written to the stream,
% but are buffered internally, to be physically written to the output device.

DELAY      Flush(x) UNTIL GROUND(x).
```

13.16 NumbersIO

```
EXPORT      NumbersIO.


% Module providing input/output for integers, rationals, and floating-point
% numbers.


IMPORT      IO, Numbers.


BASE        FileInfo.
%
% Type for constants indicating end of file.


CONSTANT    EOF, NotEOF : FileInfo.
%
% Constants indicating end of file and not end of file, respectively.

PREDICATE   ReadInteger :

  InputStream    % An open input stream.
* Integer        % An integer read from the stream unless the end of the file
                 % has already been reached.
* FileInfo.      % EOF if already end of file; otherwise, NotEOF.

DELAY       ReadInteger(x,_,_) UNTIL GROUND(x).

% ReadInteger skips over layout characters to find the next token which should
% be an integer. If the next token is not an integer, ReadInteger fails. If end
% of file has already been reached (or if there are only layout characters
% before the end of file), ReadInteger succeeds with 0 in the second argument
% and EOF in the third. If end of file has not been reached and ReadInteger
% succeeds, then NotEOF is returned in the third argument.


PREDICATE   ReadRational :

  InputStream    % An open input stream.
* Rational       % A rational read from the stream unless the end of the file
                 % has already been reached.
* FileInfo.      % EOF if already end of file; otherwise, NotEOF.
```

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

% ReadRational skips over layout characters to find the next token which should
% be a rational. If the next token is not a rational, ReadRational fails. If end
% of file has already been reached (or if there are only layout characters
% before the end of file), ReadRational succeeds with 0 in the second argument
% and EOF in the third. If end of file has not been reached and ReadRational
% succeeds, then NotEOF is returned in the third argument.


PREDICATE   ReadFloat :

  InputStream    % An open input stream.
* Float          % A floating-point number read from the stream unless the end
                 % of the file has already been reached.
* FileInfo.      % EOF if already end of file; otherwise, NotEOF.

DELAY       ReadFloat(x,_,_) UNTIL GROUND(x).

% ReadFloat skips over layout characters to find the next token which should
% be a floating-point number. If the next token is not a floating-point number,
% ReadFloat fails. If end of file has already been reached (or if there are
% only layout characters before the end of file), ReadFloat succeeds with 0 in
% the second argument and EOF in the third. If end of file has not been reached
% and ReadFloat succeeds, then NotEOF is returned in the third argument.


PREDICATE   WriteInteger :

  OutputStream   % An open output stream.
* Integer.       % An integer which is written to the stream.

DELAY       WriteInteger(x,y) UNTIL GROUND(x) & GROUND(y).


PREDICATE   WriteRational :

  OutputStream   % An open output stream.
* Rational.      % A rational which is written to the stream.

DELAY       WriteRational(x,y) UNTIL GROUND(x) & GROUND(y).
```

```
PREDICATE    WriteFloat :

  OutputStream   % An open output stream.
* Float.         % A floating-point number which is written to the stream.

DELAY        WriteFloat(x,y) UNTIL GROUND(x) & GROUND(y).
```

13.17 FlocksIO

```
EXPORT       FlocksIO.


% Module providing input/output for flocks.


IMPORT       IO, Flocks.


PREDICATE    GetFlock :

  InputStream    % An open input stream corresponding to a file with a .flk
                 % extension.
* Flock.         % The representation of the flock in the file corresponding to
                 % the input stream in the first argument is read into this
                 % argument.

DELAY        GetFlock(x,_) UNTIL GROUND(x).


PREDICATE    PutFlock :

  OutputStream   % An open output stream corresponding to a file with a .flk
                 % extension.
* Flock.         % The representation of the flock in this argument is
                 % written to the output stream in the first argument.

DELAY        PutFlock(x,y) UNTIL GROUND(x) & GROUND(y).
```

13.18 ProgramsIO

```
EXPORT       ProgramsIO.


% Module providing input/output for the ground representation of (object)
% Goedel programs.


IMPORT       IO, Programs.


PREDICATE    GetProgram :

  InputStream    % An open input stream corresponding to a file with a .prm
                 % extension.
* Program.       % The ground representation of the program in the file
                 % corresponding to the input stream in the first argument is
                 % read into this argument.

DELAY        GetProgram(x,_) UNTIL GROUND(x).


PREDICATE    PutProgram :

  OutputStream   % An open output stream corresponding to a file with a .prm
                 % extension.
* Program.       % The ground representation of the program in this argument is
                 % written to the output stream in the first argument.

DELAY        PutProgram(x,y) UNTIL GROUND(x) & GROUND(y).
```

13.19 ScriptsIO

```
EXPORT       ScriptsIO.


% Module providing input/output for the ground representation of (object)
% Goedel scripts.


IMPORT       IO, Scripts.


PREDICATE    GetScript :

  InputStream    % An open input stream corresponding to a file with a .scr
                 % extension.
* Script.        % The ground representation of the script in the file
                 % corresponding to the input stream in the first argument is
                 % read into this argument.

DELAY        GetScript(x,_) UNTIL GROUND(x).


PREDICATE    PutScript :

  OutputStream   % An open output stream corresponding to a file with a .scr
                 % extension.
* Script.        % The ground representation of the script in this argument is
                 % written to the output stream in the first argument.

DELAY        PutScript(x,y) UNTIL GROUND(x) & GROUND(y).
```

13.20 TheoriesIO

```
EXPORT       TheoriesIO.


% Module providing input/output for the ground representation of (object)
% theories.


IMPORT       IO, Theories.


PREDICATE    GetTheory :

  InputStream    % An open input stream corresponding to a file with a .thy
                 % extension.
* Theory.        % The ground representation of the theory in the file
                 % corresponding to the input stream in the first argument is
                 % read into this argument.

DELAY        GetTheory(x,_) UNTIL GROUND(x).


PREDICATE    PutTheory :

  OutputStream   % An open output stream corresponding to a file with a .thy
                 % extension.
* Theory.        % The ground representation of the theory in this argument is
                 % written to the output stream in the first argument.

DELAY        PutTheory(x,y) UNTIL GROUND(x) & GROUND(y).
```

13.21 System Utilities

This section contains a description of the system utilities that are provided by Gödel.

- `flock-compile "File" "Flock"` – takes a file `File` containing a flock and produces a file `Flock.flk` containing the internal representation of this flock.
- `flock-decompile "Flock" "File"` – takes a file `Flock.flk` containing the internal representation of a flock and produces a file `File` containing this flock.
- `program-compile Main` – takes a program with main module `Main` and produces the ground representation of the program in a file `Main.prm`.
- `program-decompile "File"` – takes a file `File` which contains the ground representation of a program and produces files containing the modules of this program.
- `theory-compile Name` – takes a file containing the theory `Name` and produces the ground representation of the theory in the file `Name.thy`.
- `theory-decompile "File"` – takes a file `File` which contains the ground representation of a theory and produces a file containing this theory.
- `script-view Name` – allows viewing of the script whose ground representation is in the file `Name.scr`.

Appendix A

Polymorphic Many-Sorted Logic

Many-sorted first order logic generalises ordinary (unsorted) first order logic ([7], [16]) in that it has sort declarations for the variables, constants, functions, and predicates. We can regard (unsorted) logic as a special case in which there is only one sort. In the general case, there are a number of (possibly infinitely many) sorts. Each constant, for example, is then specified as having a particular sort. There is also a natural definition of what it means for an expression to be a formula in the many-sorted language. For the declarative semantics, we have to generalise the usual notions of interpretation, logical consequence, and so on. The key idea here is that a many-sorted interpretation has a domain for every sort (instead of just one domain for an unsorted logic). Then each constant, for example, is assigned by the interpretation an element of the domain corresponding to its sort. The remainder of the declarative concepts can be developed in a natural way. Polymorphism is introduced by adding parameters, which are sort variables, and constructors. Generally speaking, the development of the theory of (unsorted) logic can be carried through with only minor changes for the more general case of polymorphic many-sorted logic. For further discussion on many-sorted logic beyond that given in this appendix, see [7]. There is also some discussion of many-sorted logic with applications to deductive database systems in [15]. In the following, to conform to the terminology normally used in programming languages, we refer to a sort as a *type*. For clarity, we treat many-sorted logic in detail first, then consider polymorphic many-sorted logic. We also give definitions of the basic logic programming concepts of program, completion, and correct answer in the setting of polymorphic many-sorted logic.

A.1 Many-sortedness

The alphabet of a many-sorted language contains types, variables, constants, functions, propositions, predicates, connectives, and quantifiers. In general, there is at least one type. Also there are zero or more constants and functions, and at least one proposition or predicate. Types are denoted by Greek letters such as τ and σ. Variables and constants have types such as τ. For each type τ, there are denumerably many variables v_τ^1, v_τ^2, ... of type τ. Occasionally it will be convenient to omit the superscript on a variable. Functions of arity n have types of the form $\tau_1 \times \ldots \times \tau_n \to \tau$, and predicates of arity n have types of the form $\tau_1 \times \ldots \times \tau_n$. If f has type $\tau_1 \times \ldots \times \tau_n \to \tau$, we say f has *range type* τ. For each type τ, there is a universal quantifier $\forall_\tau$ and an existential quantifier $\exists_\tau$.

Definition A *term* is defined inductively as follows:

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

2. A constant of type τ is a term of type τ.

3. If f is an n-ary function of type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$ and t_i is a term of type τ_i $(i = 1, \ldots, n)$, then $f(t_1, \ldots, t_n)$ is a term of type τ.

Next we define many-sorted formulas. These formulas will be *rectified*, in the sense that no variable can be bound by more than one quantifier and no variable can be both bound and free.

Definition A *many-sorted formula* is defined inductively as follows:

1. If p is a proposition, then p is a many-sorted atomic formula.

2. If p is an n-ary predicate of type $\tau_1 \times \ldots \times \tau_n$ and t_i is a term of type τ_i $(i = 1, \ldots, n)$, then $p(t_1, \ldots, t_n)$ is a many-sorted atomic formula.

3. If F and G are many-sorted formulas (whose common variables are free in both formulas), then so are $\sim F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$, and $F \leftrightarrow G$.

4. If F is a many-sorted formula and v_τ is a variable (free in F) of type τ, then $\forall_\tau v_\tau\ F$ and $\exists_\tau v_\tau\ F$ are many-sorted formulas.

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

We let $\forall(F)$ denote the universal closure of the formula F and $\exists(F)$ denote the existential closure. The universal closure of F is obtained by prefixing F with universal quantifiers, each one corresponding to a free variable in F. The existential closure is defined analogously.

Now we turn to the declarative semantics of many-sorted languages.

Definition A *pre-interpretation* of a many-sorted language consists of the following:

1. For each type τ, a non-empty set D_τ, called the *domain of type* τ of the pre-interpretation.

2. For each constant of type τ, the assignment of an element in D_τ.

3. For each n-ary function of type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$, the assignment of a mapping from $D_{\tau_1} \times \ldots \times D_{\tau_n}$ to D_τ.

Definition An *interpretation* I of a many-sorted language consists of a pre-interpretation J with domains $\{D_\tau\}$ together with the following:

1. For each proposition, the assignment of a value, true or false.

2. For each n-ary predicate of type $\tau_1 \times \ldots \times \tau_n$, the assignment of a mapping from $D_{\tau_1} \times \ldots \times D_{\tau_n}$ to {true, false} (or, equivalently, a relation on $D_{\tau_1} \times \ldots \times D_{\tau_n}$).

Definition Let J be a pre-interpretation of a many-sorted language. A *variable assignment* (*wrt* J) is an assignment to each variable of type τ of an element in the domain D_τ of J, for each type τ.

Definition Let J be a pre-interpretation with domains $\{D_\tau\}$ of a many-sorted language L and let V be a variable assignment. The *term assignment* (*wrt* J *and* V) of the terms in L is defined as follows:

1. Each variable is given its assignment according to V.
2. Each constant is given its assignment according to J.
3. If $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ and f' is the assignment of the n-ary function f of range type τ, then $f'(t'_1, \ldots, t'_n) \in D_\tau$ is the term assignment of $f(t_1, \ldots, t_n)$.

Definition Let I be an interpretation with domains $\{D_\tau\}$ of a many-sorted language L and let V be a variable assignment. Then a formula in L can be given a *truth value*, true or false, (*wrt* I *and* V) as follows:

1. If the formula is a proposition p, then the truth value of the formula is the same as the value assigned to p by I.
2. If the formula is an atom $p(t_1, \ldots, t_n)$, then the truth value is obtained by calculating the value of $p'(t'_1, \ldots, t'_n)$, where p' is the mapping assigned to p by I and $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ wrt I and V.
3. If the formula has the form $\sim F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$, or $F \leftrightarrow G$, then the truth value of the formula is given by the following table:

F	G	$\sim F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

4. If the formula has the form $\exists_\tau v_\tau\ F$, then the truth value of the formula is true if there exists $d \in D_\tau$ such that F has truth value true wrt I and $V(v_\tau/d)$, where $V(v_\tau/d)$ is V except that v_τ is assigned d; otherwise, its truth value is false.
5. If the formula has the form $\forall_\tau v_\tau\ F$, then the truth value of the formula is true if, for all $d \in D_\tau$, we have that F has truth value true wrt I and $V(v_\tau/d)$; otherwise, its truth value is false.

The truth value of a closed formula does not depend on the variable assignment. Consequently, we can speak unambiguously of the truth value of a closed formula wrt to an interpretation. If the truth value of a closed formula wrt to an interpretation is true (resp., false), we say the formula is true (resp,. false) wrt to the interpretation.

Definition Let I be an interpretation of a many-sorted language L and let F be a closed formula of L. Then I is a *model* for F if F is true wrt I.

The axioms of a many-sorted theory are a designated subset of closed formulas in the language of the theory. For example, the many-sorted theories in which we are most interested have the statements of a program as their axioms.

Definition Let T be a many-sorted theory and let L be the language of T. A *model* for T is an interpretation for L which is a model for each axiom of T.
If T has a model, we say T is *consistent*.

The concept of a model of a closed formula can easily be extended to a model of a set of closed formulas.

Definition Let S be a set of closed formulas of a many-sorted language L and let I be an interpretation of L. We say I is a *model* for S if I is a model for each formula of S.

Now we can give the definition of the important concept of logical consequence.

Definition Let S be a set of closed formulas and F be a closed formula of a many-sorted language L. We say F is a *logical consequence* of S if, for every interpretation I of L, I is a model for S implies that I is a model for F.

There is a transformation of many-sorted formulas into (unsorted) formulas, which shows that the apparent extra generality provided by many-sorted logics is illusory [7]. This transformation allows one to reduce the proof of a theorem in many-sorted logic to a corresponding theorem in (unsorted) logic. The existence of this theorem makes many-sorted logic a mathematically trivial extension of (unsorted) logic and explains why so few logic books even mention many-sorted logic. However, many-sorted logic is more expressive than (unsorted) logic and it avoids having to use the type predicates that are introduced in the mapping to (unsorted) logic.

A.2 Polymorphism

Next we introduce (parametric) polymorphism into the logic. For this, we extend the alphabet by adding parameters, bases, and constructors. Parameters are type variables; bases correspond to what were called types in the many-sorted case; and constructors have an arity and are used to construct new types. In contrast to the many-sorted case, there is now a single polymorphic universal quantifier $\forall$ and a single polymorphic existential quantifier $\exists$. Also variables do not have fixed types as in the many-sorted case, but have their types inferred from the context in which they occur. We assume that there are denumerably many variables v^1, v^2, ...

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 $\tau_1, \ldots, \tau_n$ 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 each type is defined according to the preceding definition). A symbol is *polymorphic* if its type contains a parameter; otherwise, it is *monomorphic*. A polymorphic symbol can be intuitively understood as representing a collection of (monomorphic) symbols, one for each ground instance of its type. (See below.)

We now define the concept of a term t of type τ 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 $\tau_1 \times \ldots \times \tau_n \to \tau$ and let t_i be a term of type σ_i, for $i = 1, \ldots, n$. Suppose that the parameters in $\tau_1 \times \ldots \times \tau_n \to \tau$ 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 standardized 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 we say that $f(t_1, \ldots, t_n)$ 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 $f(t_1, \ldots, t_n)$ all have the same type in $f(t_1, \ldots, t_n)$. Also the type of $f(t_1, \ldots, t_n)$ and the type in $f(t_1, \ldots, t_n)$ of each of its subterms is unique up to variants.

Next we define the concept of an atom A 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 standardized 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 we say that $p(t_1, \ldots, t_n)$ 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 $p(t_1, \ldots, t_n)$ all have the same type in $p(t_1, \ldots, t_n)$. Also the type in $p(t_1, \ldots, t_n)$ of each of its subterms is unique up to variants.

Now we can give the definition of a polymorphic many-sorted formula F 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 standardized 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., $F \vee G$, $F \rightarrow G$, $F \leftrightarrow G$) 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., $F \vee G$, $F \rightarrow G$, $F \leftrightarrow G$), 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., $F \vee G$, $F \rightarrow G$, $F \leftrightarrow G$).

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 is 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.

The universal closure and existential closure of a polymorphic formula can be defined by obvious extensions of the many-sorted definitions.

A polymorphic formula can be intuitively understood as representing a collection of (monomorphic) formulas. We now make this idea more precise.

Definition Let L be a polymorphic many-sorted language. The *underlying many-sorted language* of L is the (monomorphic) many-sorted language L^* with the following alphabet:

1. The types of L^* are the ground types of L. (Any structure in a ground type is ignored.)

2. For each ground type δ in L, there are denumerably many variables v_δ^1, v_δ^2, ... in L^*.

3. For each constant c of type τ in L and ground instance δ of τ, there is a constant c_δ of type δ in L^*.

4. For each function f of type $\tau_1 \times \ldots \times \tau_n \to \tau$ in L and ground instance $\delta_1 \times \ldots \times \delta_n \to \delta$ of $\tau_1 \times \ldots \times \tau_n \to \tau$, there is a function $f_{\delta_1 \times \ldots \times \delta_n \to \delta}$ of type $\delta_1 \times \ldots \times \delta_n \to \delta$ in L^*.

5. The propositions of L^* are the propositions of L.

6. For each predicate p of type $\tau_1 \times \ldots \times \tau_n$ in L and ground instance $\delta_1 \times \ldots \times \delta_n$ of $\tau_1 \times \ldots \times \tau_n$, there is a predicate $p_{\delta_1 \times \ldots \times \delta_n}$ of type $\delta_1 \times \ldots \times \delta_n$ in L^*.

Definition Let F be a formula in a polymorphic many-sorted language. Each occurrence of a variable, constant, function, or predicate in F has a *relative type* in F defined as follows.

1. The relative type of an occurrence of a variable v in F is the type of v in F.

2. The relative type of an occurrence of a constant c in F is the type of the subterm c in F.

3. Consider an occurrence of the function f in the subterm $f(t_1, \ldots, t_n)$ of F. Suppose the subterm $f(t_1, \ldots, t_n)$ has type τ in F and the subterm t_i has type τ_i in F $(i = 1, \ldots, n)$. Then the relative type of this occurrence of f in F is $\tau_1 \times \ldots \times \tau_n \to \tau$.

4. Consider an occurrence of the predicate p in the atom $p(t_1, \ldots, t_n)$ of F. Suppose the subterm t_i has type τ_i in F $(i = 1, \ldots, n)$. Then the relative type of this occurrence of p in F is $\tau_1 \times \ldots \times \tau_n$.

Note that the relative type of a constant in a formula is an instance of its type in the language. A similar comment applies to functions and predicates.

Definition Let F be a formula in a polymorphic many-sorted language. A *grounding type substitution* is a type substitution which binds all the parameters appearing in relative types of symbols in F to ground types.

Definition Let F be a formula in a polymorphic many-sorted language L and Ψ be a grounding type substitution for F. The many-sorted formula F_Ψ in the language L^* is obtained by replacing all occurrences of symbols in F as follows:

1. For an occurrence of a variable v of relative type τ appearing in F, replace v by the variable $v_{\tau\Psi}$.

2. For an occurrence of a constant c of relative type τ appearing in F, replace c by the constant $c_{\tau\Psi}$.

3. For an occurrence of a function f of relative type $\tau_1 \times \ldots \times \tau_n \to \tau$ appearing in F, replace f by the function $f_{\tau_1\Psi \times \ldots \times \tau_n\Psi \to \tau\Psi}$.

4. For an occurrence of a predicate p of relative type $\tau_1 \times \ldots \times \tau_n$ appearing in F, replace p by the predicate $p_{\tau_1\Psi \times \ldots \times \tau_n\Psi}$.

Now we can give precise meaning to the intuitive concept of a polymorphic many-sorted formula representing a set of (monomorphic) many-sorted formulas.

Definition Let F be a formula in a polymorphic many-sorted language. The set

$$\{F_\Psi : \Psi \text{ is a grounding type substitution for } F\}$$

is called the *set of many-sorted formulas underlying* F.

Now we turn to the declarative semantics of polymorphic many-sorted languages.

Definition A *pre-interpretation* of a polymorphic many-sorted language consists of the following:

1. For each ground type δ, a non-empty set D_δ, called the *domain of type* δ of the pre-interpretation.
2. For each constant of type τ and for each ground instance δ of τ, the assignment of an element in D_δ.
3. For each n-ary function of type $\tau_1 \times \ldots \times \tau_n \to \tau$ and for each ground instance $\delta_1 \times \ldots \times \delta_n \to \delta$ of $\tau_1 \times \ldots \times \tau_n \to \tau$, the assignment of a mapping from $D_{\delta_1} \times \ldots \times D_{\delta_n}$ to D_δ.

Definition An *interpretation* I of a polymorphic many-sorted language consists of a pre-interpretation J with domains $\{D_\delta\}$ together with the following:

1. For each proposition, the assignment of a value, true or false.
2. For each n-ary predicate of type $\tau_1 \times \ldots \times \tau_n$ and for each ground instance $\delta_1 \times \ldots \times \delta_n$ of $\tau_1 \times \ldots \times \tau_n$, the assignment of a mapping from $D_{\delta_1} \times \ldots \times D_{\delta_n}$ to {true, false} (or, equivalently, a relation on $D_{\delta_1} \times \ldots \times D_{\delta_n}$).

An interpretation for a polymorphic many-sorted language L can also be regarded as an interpretation for the many-sorted language L^* underlying L in the obvious way.

Definition Let I be an interpretation of a polymorphic many-sorted language L and F be a closed formula of L. The truth value of F wrt I is true if, for every grounding type substitution Ψ for F, the truth value of F_Ψ is true wrt I. The truth value of F wrt I is false if, for every grounding type substitution Ψ for F, the truth value of F_Ψ is false wrt I.

If the truth value of a closed formula wrt to an interpretation is true (resp., false), we say the formula is true (resp,. false) wrt to the interpretation.

We conclude with the definitions of model, consistency, and logical consequence, which are analogous to the many-sorted case.

Definition Let I be an interpretation of a polymorphic many-sorted language L and F be a closed formula of L. Then I is a *model* for F if F is true wrt I.

Definition Let T be a polymorphic many-sorted theory and let L be the language of T. A *model* for T is an interpretation for L which is a model for each axiom of T.

If T has a model, we say T is *consistent*.

Definition Let S be a set of closed formulas of a polymorphic many-sorted language L and let I be an interpretation of L. We say I is a *model* for S if I is a model for each formula of S.

Definition Let S be a set of closed formulas and F be a closed formula of a polymorphic many-sorted language L. We say F is a *logical consequence* of S if, for every interpretation I of L, I is a model for S implies that I is a model for F.

A.3 Logic Programming Concepts

In this section, we give the definitions of the basic declarative logic programming concepts of program, completion, and correct answer in the context of polymorphic many-sorted logic. Throughout this section, we suppose the definitions are given in the context of some fixed, but arbitrary, polymorphic many-sorted language.

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.

Next we give the definition of the completion of a program.

Definition The *definition* of a 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 predicate p in a program is

$$\begin{gathered} A_1 \leftarrow W_1 \\ \ldots \\ A_k \leftarrow W_k \end{gathered}$$

Then the *completed definition* of p is the formula

$$\forall x_1 \ldots \forall x_n (p(x_1, \ldots, x_n) \leftrightarrow \; E_1 \vee \ldots \vee E_k)$$

where E_i is $\exists y_1 \ldots \exists y_d((x_1 = t_1) \wedge \ldots \wedge (x_n = t_n) \wedge W_i)$, 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.

Example Let the definition of p be
$p(x) \leftarrow q(x, y)$

$p(b) \leftarrow$
Then the completed definition for p is

$$\forall z(p(z) \leftrightarrow (\exists x \exists y((z = x) \wedge q(x, y)) \vee (z = b))).$$

Definition Let P be a program and p a predicate occurring in P. Suppose there is no program statement in P with predicate p in its head. Then the *completed definition* of p is the formula

$$\forall x_1 \ldots \forall x_n \sim p(x_1, \ldots, x_n).$$

The *equality theory* for a program consists of all axioms of the following form:

1. $c \neq d$, for all pairs c, d of distinct constants whose types have a common instance.
2. $\forall(f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m))$, for all pairs f, g 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. $\forall((x_1 \neq y_1) \ \vee \ \ldots \ \vee \ (x_n \neq y_n) \ \rightarrow \ f(x_1, \ldots, x_n) \neq f(y_1, \ldots, y_n))$, for each function f.
6. $\forall x(x = x)$.
7. $\forall((x_1 = y_1) \ \wedge \ \ldots \ \wedge \ (x_n = y_n) \ \rightarrow \ f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$, for each function f.
8. $\forall((x_1 = y_1) \ \wedge \ \ldots \ \wedge \ (x_n = y_n) \ \rightarrow \ (p(x_1, \ldots, x_n) \ \rightarrow p(y_1, \ldots, y_n)))$, 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 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 to a program.

Bibliography

[1] *Language Independent Arithmetic, ISO/IEC CD 10967-1:1992.* August 1992. Second Committee Draft (Version 4.0); JTC1/SC22/WG11 N318, ANSI X3T2 92-064.

[2] *Standard for Binary Floating-Point Arithmetic.* 1985. ANSI/IEEE Std 754-1985.

[3] K.A. Bowen and R.A. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S.-A. Tarnlund, editors, *Logic Programming*, pages 153–172, Academic Press, 1982.

[4] A.F. Bowers. *Representing Gödel Object Programs in Gödel.* Technical Report CSTR-92-31, Department of Computer Science, University of Bristol, 1992.

[5] K. L. Clark, F. G. McCabe, and S. Gregory. IC-Prolog language features. In K.L. Clark and S.-A. Tarnlund, editors, *Logic Programming*, pages 253–266, Academic Press, 1982.

[6] A. Dovier, E.G. Omodeo, E. Pontelli, and G. Rossi. {Log}: a logic programming language with finite sets. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 111–124, Paris, 1991.

[7] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[8] C.A. Gurr. *SAGE: A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* Technical Report CSTR-93-??, Department of Computer Science, University of Bristol, 1993.

[9] C.A. Gurr. Specialising the ground representation in the logic programming language Gödel. In *Proceedings of the Third International Workshop on Logic Program Synthesis and Transformation*, Springer-Verlag, July 1993.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] J.-L. Lassez and K. McAloon. A canonical form for generalized linear constraints. *J. Symbolic Computation*, 13:1–24, 1992.

[15] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[16] E. Mendelson. *Introduction to Mathematical Logic.* 3rd Edition, Van Nostrand, 1987.

[17] 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.

[18] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[19] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[20] J. A. Thom and J. Zobel. *NU-Prolog Reference Manual, Version 1.3.* Technical Report, Machine Intelligence Project, Department of Computer Science, University of Melbourne, 1988.

System Index

General Index