Representing Gödel Object Programs in Gödel*

A.F. Bowers

November 1992 CSTR-92-31

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

email: Antony.Bowers@bristol.ac.uk

*Work supported by ESPRIT Basic Research Action COMPULOG (3012)

Abstract

Meta-programs are an important class of programs. When meta-programming is done in logic the object program must be represented in the meta-program. This can be done in two ways: the *non-ground* and *ground* representations. The non-ground representation severely limits the meta-program. The ground representation is more powerful, but rarely used because it is thought to be complex, laborious and to have a high computational overhead. Prolog programs therefore use the non-ground representation and supplement it with non-logical features. Gödel is a logic programming language that aims to have similar expressive power to Prolog but with greatly improved declarative semantics. Gödel has types and a module system, and supports declarative meta-programming in the ground representation. It is argued that providing the representation as an abstract data type answers the accusation of inconvenience. The representation of Gödel programs as ground terms is investigated and shown to be practical. The prospect of reducing the computational overhead by partial evaluation exists.

All that I have to say is to tell you that the lanthorn is the moon; I, the Man i' th' Moon; this thorn-bush, my thorn-bush; and this dog, my dog.

–*William Shakespeare*, "A Midsummer Night's Dream" Act 5 Scene I.

1 Introduction

Computer programming is difficult. Anyone who reads the computing press will know that an alarming number of software development projects fail, simply because the programs cannot be made to work with the available resources. The precise reasons for these failures may be various, but one is surely chief among them: unmastered complexity.

By programming in logic we can go a long way towards mastering complexity. A programmer using a logic programming language is likely to be able to handle a more intricate problem before being overwhelmed than a programmer using a procedural language. This is because the logic language is compact, high-level and expressive. Since first-order logic developed out of attempts to mechanise certain aspects of human reasoning, it is close to the way we naturally express ideas about the world. It has a simple, clean semantics that is easy to understand. Add types and modules to the logic, and the advantage is greater still.

Another way to handle complexity is to seek help from automation. Programs written in a language with simple declarative semantics can more easily be synthesised, transformed, analysed and debugged by other programs, and the soundness of these processes can be verified. For these reasons, we have a chance of contributing to the future of computing by taking seriously the central thesis of logic programming: that a program is a theory, and computation is deduction.

Meta-programs are an important class of programs. A meta-program is any program that treats one or more other programs, called object programs, as data. Interpreters, compilers, program transformers and debuggers are all examples of meta-programs. If our idea that programs are first-order theories is a natural and useful one, clearly we should be able to express meta-programs in this form. The natural way to view meta-programs is as first-order theories over domains that are representations of their object theories. There are two distinct ways in which this representation of object programs can be accomplished: the *non-ground* and *ground* representations [5].

In the non-ground representation, object-level variables are represented by variables at the meta-level. This imposes severe restrictions on the power of the meta-program; for example, it cannot inspect the variables of the object program, and it cannot add statements to the object program. The latter restriction arises because it is impossible to distinguish between representations of statements containing free object-level variables and incomplete statement representations containing meta-level variables.

In the ground representation, object-level variables are represented by ground terms at the meta-level. This gives a much more powerful formalism for meta-programming in which object-level variables can be manipulated in exactly the same way as any other syntactic element of the object program. If the entire object program is represented as a single term in the meta-program, it is easy to add, remove or change statements in the object program, and object-level goals can be proved by means of a meta-interpreter [7].

If meta-programming is to be done in logic, there is no obvious alternative to making the ground representation effective. There is surely little prospect of discovering a logic in which meta-programs using a non-ground representation can inspect object-level variables and yet retain semantics as simple, intuitive and tractable as classical first-order logic.

Despite its power, elegance and logical purity, the ground representation is rarely used in current logic programming practice. This is because it appears to have the following drawbacks:

- the representation of object programs as terms is too complex;
- meta-programming in the ground representation is laborious, because large procedures are required to do simple things such as unifying object-level terms;
- object-level variables and their bindings must be handled explicitly, an overhead that makes the meta-program unacceptably slow.

As a consequence of these perceived difficulties, unsatisfactory solutions to the problem of meta-programming are commonly adopted. The majority of meta-programs written in Prolog use the non-ground representation, and circumvent the inadequacy of that representation by resorting to non-logical facilities that have been added to the language for that purpose. Programs that use the meta-logical predicates of Prolog such as `var`, `nonvar`, `assert` and `retract` cannot be understood directly as theories, and therefore lose the advantages of a simple declarative semantics. They are much harder to debug and transform, for example, and also harder to execute in parallel because the non-logical features have a procedural component that implies sequential execution.

Several attempts have been made to address this deficiency of Prolog. The MetaProlog language [1] provides more declarative forms of `assert` and `retract` by keeping track of named theories, and has a mechanism for distinguishing between object-variables and meta-variables when theories are updated. However MetaProlog is still mainly based on the non-ground representation. The language 'LOG (pronounced *quotelog*) [3] directly supports the ground representation of a Prolog-like language, and has built-in predicates for creating such representations. The representations are based on lists, and a large part of the labour of manipulating them is left to the programmer.

Gödel is a logic programming language intended as a successor to Prolog. It aims to have power and expressiveness similar to that of Prolog, but with greatly improved declarative semantics. Gödel also has a type system and a module system; the value of these is well-known in software engineering. Indeed, these features greatly benefit Gödel's meta-programming facilities.

Although it is possible to write programs using the non-ground representation in Gödel, the ground representation is far more important. Using the ground representation, Gödel gives completely declarative counterparts to Prolog's `var`, `nonvar`, `assert`, `retract`, `=..` etc. By providing the ground representation as an abstract data type, Gödel frees the programmer from concern with the details of the representation, and allows access to it at a relatively high level by means of an extensive library of predicates, each with a precise semantic definition. This renders meta-programming in Gödel almost as straightforward as it is in Prolog, with the advantage of declarative semantics, and answers the objec-

tions based on complexity and labour that have previously discouraged use of the ground representation.

The thought of representing entire object programs, even Prolog object programs, as single terms can seem rather daunting. In representing a Gödel program, we have the extra burden of object programs written in a much richer language. The representation has to include the module structure of the object program, its type declarations, statements and control annotations. It is, of course, these very features that enable effective use of the ground representation in Gödel.

In the remainder of this paper a specific representation for Gödel programs in Gödel is described, a representation that is sufficient to meet the specifications of the meta-predicates in Gödel's system modules. This representation is in a pure polymorphic many-sorted logic, and does not use any features outside the Gödel language. Apart from being declarative, this logical representation theoretically allows construction of a tower of meta-levels for free; that is, since the representation is a Gödel term, it can itself be represented. In practice the size of the representation increases dramatically with each meta-level and becomes unmanageable for even small programs after the third or fourth level. The paper demonstrates that as long as careful attention is paid to certain design considerations, the representation in typed logic of a structure as complex as a Gödel program can be accomplished in a fairly straightforward way, and still allow standard access and update operations to be performed simply and reasonably efficiently. The Gödel programmer is of course unaware of the details of the representation, as it is presented as an abstract data type. This leaves the language implementors free to alter the structure at will.

The paper is organised as follows. In the next section a brief overview of the Gödel language is presented. The reader is given the flavour of meta-programming in Gödel and then a representation for Gödel object programs is described in detail. Finally, we draw some conclusions and suggest directions for future research.

2 The Gödel Language

Space does not permit more than a cursory overview of Gödel here. For a complete technical description and specification the reader is referred to [6].

Symbols in Gödel either begin with an upper case-letter, or are composed entirely of non-alphanumeric characters. Variables begin with a lower-case letter. Gödel statements and goals can have arbitrary first-order formulas in the body, and may also contain a *commit* pruning operator, and the *IF-THEN-ELSE* conditionals of [8].

Gödel has a type system based on *polymorphic many-sorted logic*. The statements of a Gödel program are written in a language defined by a set of language declarations. Every symbol in the language has a declaration that places it in one of the six categories: `BASE`, `CONSTRUCTOR`, `CONSTANT`, `FUNCTION`, `PROPOSITION` and `PREDICATE`. The types (or sorts) of this language are built from the symbols declared as bases and constructors, together with *parameters*, in the same way that terms are built from constants, functions and variables. Every constructor symbol is assigned an arity by its declaration. Parameters provide polymorphism; they are type variables and range over all the type symbols in the language. Thus given the declarations

```
BASE Hat.
CONSTRUCTOR List/1.
```

we have a type `Hat`, a polymorphic type `List(a)`, and instances of this type `List(Hat)`, `List(List(a))` and so on.

When a constant, function or predicate symbol is declared, it is assigned a domain and/or range type as appropriate from this set of types. Functions and predicates may also be specified as prefix, infix or postfix operators by including an *indicator* in their declaration. A typical operator declaration might be

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

The indicator `yFx(510)` specifies the precedence and associativity of the operator.

Gödel predicates may have control declarations, called `DELAY` declarations. These are syntactic variants of the `when` declarations of NU-Prolog [10]. `DELAY` declarations have the form

`DELAY` *Atom* `UNTIL` *Condition*

where *Atom* is an atom (the *head*) and *Condition* is constructed from `NONVAR(`*var*`)`, `GROUND(`*var*`)`, `TRUE`, and connectives `&` and `\/`. Variable *var* must occur in *Atom*. Delay declarations influence the computation rule as follows: if a call unifies with *Atom*, it is delayed until it becomes an instance of *Atom* and then until *Condition* is satisfied. There may be several delay declarations for the same predicate, provided no pair of heads have a common instance.

Most Gödel modules consist of two parts, a local part and an export part. The *local part* of a module is indicated by a `LOCAL` or `MODULE` module declaration. The *export part* is indicated by an `EXPORT` or `CLOSED` module declaration. These keywords are followed by the name of the module. Modules are linked by `IMPORT` declarations that name imported modules.

The export part of a module can contain `IMPORT` declarations, language declarations and control declarations. The local part can contain `IMPORT` declarations, language declarations, control declarations and statements. If a module consists only of a local part, it has a `MODULE` declaration rather than a `LOCAL` declaration. The keyword `CLOSED` appearing in the export part of a module implies that it is a system module and specially protected. Modules that are not closed are said to be *open*.

A Gödel module exports all the symbols declared in its export part, and all the symbols it imports into its export part. The declaration

```
IMPORT Hats.
```

appearing in a part of a module imports all the symbols exported by module `Hats` into that part of the module. Thus a module can import module `Hats` either *directly* (by naming `Hats` in an import declaration) or *indirectly* (by importing another module that imports `Hats` into its export part). The set of symbols declared in and imported into both parts of

a module forms the *module language*, and is the language in which the statements of the module are written.

We say that a module `M` *refers to* a module `N` if either part of `M` contains the declaration `IMPORT N`. We define the relation *depends upon* to be the transitive closure of the relation `refers to`. A *Gödel program* is then a set of modules, consisting of one *main module* that is not imported by any other module, together with all the modules upon which this main module depends.

The *definition of a type* in a Gödel program is the set of constants and functions that have that type as their range type. The *definition of a predicate* (or *proposition*) in a Gödel program is the set of statements in the program that have that predicate (or proposition) in the head. The definition of a predicate or proposition must reside entirely within the module that declares it. Similarly, the definition of a type must reside within the module that declares its top-level constructor, or that declares the type itself if it is a base type.

Gödel is flexible in allowing overloading. The only condition on the naming of symbols is that distinct symbols cannot be declared in the same module with the same category, name and arity. The data structures described in this paper make liberal use of overloading: there are several instances where a function and its type have the same declared name.

The name that appears in the declaration for a symbol is called the *declared name* of the symbol. The possibility of overloading means that this name is not necessarily unique. In order to form the ground representation, we need the *flat name* of the symbol, which is the quadruple consisting of the name of the module in which the symbol is declared, its declared name, its category and its arity. The condition imposed on naming ensures that this quadruple uniquely identifies the symbol. The *flat form* of a program is obtained by replacing each occurrence of the declared name of a symbol in the program by its flat name. The *flat language* of a program is the language defined by the set of language declarations in the flat form of the program.

Gödel provides a rich set of system modules, such as `Integers`, `Lists`, `Rationals`, `Floats`, `Strings`, `Sets` and `IO` to support the commonly used data types and common operations on those types. Importing the `Lists` module makes the conventional notation for lists available. Similarly importing `Strings` allows the familiar notation `"abcd"` to be used for sequences of characters.

We conclude this overview with a tiny example program. The module `Inclusion` defines a predicate `IncludedIn` which can be used to check that all the elements of a list also appear in another list. This module forms a program together with the module `Lists` and the module `Integers` (which is imported into the export part of `Lists` and so indirectly into `Inclusion`).

```
MODULE        Inclusion.

IMPORT        Lists.

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

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

3 Meta-programming in Gödel

Three system modules are provided by Gödel for meta-programming in the ground representation: `Syntax`, `Programs` and `Theories`. The `Theories` module, which is concerned with representing full first-order theories (rather than logic programs) is outside the scope of this paper.

The `Syntax` module is concerned with representing object expressions in Gödel syntax, and is imported into the export parts of both `Programs` and `Theories`. The export part of `Syntax` declares the abstract data types

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

Terms of type `Name` represent the names of symbols; terms of type `Type`, `Term`, and `Formula` represent types, terms, and formulas, respectively; terms of type `TypeSubst` and `TermSubst` represent type and term substitutions; and terms of type `VarTyping` represent variable typings (a variable typing is set of assignments of variables to types).

`Syntax` exports many predicates for manipulating these types. For example

```
PREDICATE And : Formula * Formula * Formula.
```

`And(u,v,w)` is intended to be true when `w` represents the conjunction of the formulas represented by `u` and `v`.

The module `Programs` supports the representation of Gödel programs as terms, and exports the abstract data type `Program`. The export part of `Programs` also declares a type `ModulePart` and constants

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

to represent the part keywords of module declarations.

A typical predicate exported by `Programs` is the predicate `FormulaInModule` that is used to check whether a term of type `Formula` represents a formula that is valid in the language of a specific module. The declaration of `FormulaInModule` is

```
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.
* Formula          % Representation of a formula in the
                   % flat language of this module.
* 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.
```

Among many others, the `Programs` module also exports a pair of predicates called `StringToProgramFormula` and `ProgramFormulaToString`. The former invokes the Gödel parser in order to convert the intuitive representation of an object formula as a `String` into its ground representation. The latter works in the opposite direction, generating a string from the `Formula` abstract data type. These predicates are similar to the built-in primitive `<=>` of 'LOG [3].

The predicate `Succeed` invokes a full-scale interpreter for Gödel object programs. Its declaration is

```
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.
```

There are several other similar predicates in `Programs` that provide different flavours of interpreter.

To facilitate the creation of representations of object programs, Gödel provides a utility called the *program compiler*. This takes the source of the object program and generates its ground representation in a file. The system module `ProgramsIO` exports a predicate

```
PREDICATE GetProgram : InputStream * Program.
```

that reads the representation from the file. This IO operation is not declarative, but as long as Gödel programmers keep all IO in the top-level module the logical purity of the majority of the program is preserved.

4 Representing Gödel Syntax in Gödel

In this section we discuss the representation of syntactic expressions in polymorphic many-sorted logic: symbol names, types, terms and formulas. These objects are represented independently of the context of any particular object program, by structures declared in the `Syntax` module.

The abstract data type `Name` is provided to represent symbol names. The `Name` type is the basic unit from which syntax is constructed; the `Syntax` module can fulfil its specification without ever examining the internal structure of an `Name` term, except in one case only: since `Name` is an abstract data type, `Syntax` must provide a means for the programmer to create instances of the type.

Clearly we would like Gödel meta-programs to be independent of any particular object program, so we must avoid having to declare explicitly the meta-level constants that represent object-level names. We need a data type that has enough instances already available, and the possibility of an easily defined mapping between these instances and the object-level names. This clearly suggests using the `String` type.

We will see later that for representing Gödel programs it is convenient to have the four components of Gödel flat names explicit in the `Name` term so that they can be easily

extracted and separated. Gödel flat names are therefore represented by the following function, declared in the local part of `Syntax`:

```
FUNCTION Name :
   String                        % Name of module where symbol is declared.
 * String                        % Declared name of symbol.
 * Category                      % Symbol's category.
 * Integer                       % Symbol's arity.
-> Name.
```

The base type `Category` gives the category of the symbol and is defined by six constants:

```
CONSTANT
   Base, Constructor,
   Constant, Function, Proposition, Predicate : Category.
```

Since the `Syntax` module is not dedicated to the representation of Gödel programs, `Name` must also support the representation of names other than flat names. For this the function

```
FUNCTION SimpleName :
   String
-> Name.
```

is used; the argument can be any string.

Object types are represented by ground terms built from the following two functions:

```
FUNCTION BType :
   Name                          % Name of base type symbol.
-> Type.

FUNCTION Type :
   Name                          % Name of constructor symbol.
 * List(Type)                   % List of argument types.
-> Type.
```

The `Type` function represents a compound type with its arguments in a list, and the `BType` function represents a base type. This strategy follows a principle of good logic programming style (see [9]) that the top-level function symbol of a term should express maximum information about the term.

Type parameters are represented by ground terms using the function `Par`:

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

The index argument is provided so that type parameters can be renamed by simply incrementing the index. This needs to be done, for example, when finding the type of a term (see the algorithm in [6]).

In a similar way to types, object terms are represented by functions of type `Term`:

```
FUNCTION CTerm :
   Name                         % Name of constant symbol.
-> Term.

FUNCTION Term :
   Name                         % Name of function symbol.
 * List(Term)                  % List of argument terms.
-> Term.
```

and object atoms by functions of type `Formula`:

```
FUNCTION PAtom :
   Name                         % Name of proposition symbol.
-> Formula.

FUNCTION Atom :
   Name                         % Name of predicate symbol.
 * List(Term)                  % List argument terms.
-> Formula.
```

Of course, since this is a *ground* representation, object-level variables are represented by ground terms at the meta-level. Like those used to represent parameters, these terms also have an index argument, in this case to facilitate standardisation apart.

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

Representations of formulas, which are all terms of type `Formula`, are built from terms representing atoms, the constant `Empty` (representing the empty formula), and functions representing connectives. Among these are the unary function `~'`, binary functions `&'`, `\/'`, `->'`, `<-'` and `<->'`. Quantifiers are represented by

```
FUNCTION All, Some :
   List(Term)                    % List of quantified variables.
 * Formula                       % Quantified formula.
-> Formula.
```

and commits by

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

Lastly, there are four functions to represent four different flavours of conditional, of which the most sophisticated is

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

5 Representing a Gödel Program in Gödel

We now come to the main topic of this paper, which is the structure of terms of type `Program`. Each `Program` term represents an entire Gödel object program, including its module structure, language declarations, control declarations and statements. It is actually the flat form of the object program that is represented; in this form every symbol has a unique name and there is no overloading.

Clearly, a dictionary structure of some kind is required to keep all these components together and readily accessible in one term. Unfortunately, there is no convenient logical data structure that will allow us to access the data in constant time. It is well-known among compiler constructors that users (or in our case, perhaps automatic program synthesisers) sometimes declare large numbers of symbols in a lexical order, causing catastrophic degradation in the performance of simple binary tree dictionaries because the ordered insertion creates linear structures; a balanced tree dictionary is therefore appropriate. We also observe that lookup operations are performed very much more frequently than update operations on the `Program` structure, so the additional cost of rebalancing after insertions and deletions is not significant. We have chosen (arbitrarily) to use AVL-tree dictionaries, although other schemes (such as 2-3 trees) are available which, like AVL-trees, guarantee logarithmic time for lookup, insertion and deletion. Algorithms for insertion and deletion can be found in [12]; they are straightforward to code as Horn-clauses, see for example [2, 11].

In accordance with the principle of encapsulation, the predicates that manipulate dictionaries are provided by a subsidiary module `AVLTrees` that is imported into the local part of the `Programs` module (and is therefore invisible to the users of `Programs`). Each dictionary associates a key and a value. `AVLTrees` exports the constructor `AVLTree/1` so that the dictionary has the parametric type `AVLTree(a)`, where parameter `a` is instantiated to the type of the value the AVL-tree associates with the key. The key is always of type `String`.

At the top level, every term of type `Program` has the 4-place function `Program`, whose declaration is

```
FUNCTION Program :
   String                        % Name of the main module.
 * AVLTree(ModuleDefinition)     % Module structure.
 * Language                      % Flat language of the program.
 * AVLTree(ModuleCode)           % Statements and delays.
-> Program.
```

The four arguments represent the three main components of a Gödel program: the module structure (represented by the first two arguments), the flat language, and the program statements and delay declarations expressed in this flat language. We consider each of these in turn.

5.1 Representing the Module Structure

The module structure of a Gödel program is a tree rooted at the main module. The set of modules making up the program form the nodes of the tree, and the edges are formed by import declarations. The edges can be divided into two classes, depending on whether the import declaration is in the local part or the export part of the importing module.

It would, of course, be possible to use this tree structure directly to represent itself. However, it is more convenient to use a dictionary that simply associates the name of each module with the names of the modules that it imports. This structure has less duplication, is easier to update, is more appropriate to the questions usually asked of it, and has the useful property that the representation of each node is independent of its child nodes.

The module structure is represented by a dictionary of type

```
AVLTree(ModuleDefinition)
```

where the keys are module names. There is one function of type `ModuleDefinition`:

```
FUNCTION ModDef :
   OModuleKind                  % What kind of module this is.
 * List(String)                % Import declarations in export part.
 * List(String)                % Import declarations in local part.
-> ModuleDefinition.
```

and the type `OModuleKind` is defined by

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

These constants indicate whether the named module has an ordinary export and local part, is closed, or has a `MODULE` keyword in its local part and no export part, respectively.

For example, if on looking up the name `"Hats"` in the module structure we obtain the term

```
ModDef(NormalKind, [], ["Lists"])
```

then `Hats` is an ordinary module that has the declaration `IMPORT Lists` in its local part.

5.2 Representing the Program Language

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

1. It must contain a representation of the declaration of every symbol in the program.

2. It should be able to act as the symbol table for the parser, and handle overloading. When the Gödel parser meets a declared name in the program source, it needs to look up that name in the symbol table and recover a list of all the symbols that have that declared name.

3. It should allow access to the declaration of any symbol given its flat name. Speed is important since the declaration of every symbol in an expression must be examined during the process of validating the expression with respect to a language, and such validations are performed frequently in routine meta-programming.

4. It is often necessary to validate an expression with respect to one of the various views of the program language, such as the export language of a specific module. It should therefore be possible to retrieve any of these views from the program language, and do so at reasonable cost.

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

A naive approach to representing the language structure might be as a simple dictionary linking flat names with their declarations. For example, the string

```
"Hats.Topper.Constant"
```

might represent the flat name of the constant `Topper` in the module `Hats`, and could be used as a key to index the declaration of this constant. However, this scheme does not meet requirements 2 and 4 very well.

To solve this problem we make use of the internal structure imposed on the `Name` type in `Syntax`. We can then make use of all the components of the name to minimise the number of comparisons that are made for each access. The idea is to use nested dictionaries: the outer containing an entry for each module, which entry is in turn a dictionary linking each declared name in the module with all the symbols declared in the module with that declared name, and their declarations.

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

The flat language of the object program and the other flat languages it defines are represented by terms of type `Language`, formed from the function[1]

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

[1]This function is a remnant of a previous design incarnation; it is not essential for the representation, but gives the term representing the program language a simple (base) type.

The key for the dictionary `AVLTree(ModuleDescriptor)` is the name of a module, and the base type `ModuleDescriptor` is defined by

```
FUNCTION Module :
   Accessibility                % Which symbols are accessible.
 * CategoryTable                % Dictionary of symbols.
-> ModuleDescriptor.
```

The `Accessibility` argument is present to satisfy requirement 4. It is a flag that can take two values:

```
BASE Accessibility.

CONSTANT Exported, Hidden : Accessibility.
```

The accessibility flag determines the role that this module plays in the language as a whole by indicating which of the symbols declared in the module are actually present in the language. If the accessibility of the module is `Hidden`, all the symbols it declares are present; if its accessibility is `Exported` only those symbols declared in the export part of the module are present in the language.

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

Different views of the program language are easily constructed in this representation. For example, to find the flat language of a module `M`, first use the representation of the module structure to find all the modules `M` imports, directly or indirectly. Extract the `ModuleDescriptor` terms for these modules from the program language, set their accessibility to `Exported` and place them in a new dictionary. Finally, add to this dictionary the `ModuleDescriptor` for `M` with accessibility `Hidden`.

Since there are typically only a dozen or so modules in a medium-sized Gödel program, this is not a large computation. In an implementation using something like the structure-sharing techniques of the WAM, this procedure is also quite space efficient, because the representations of different views of the same language share the bulk of their structure.

In an early version of the ground representation[2] the module languages were pre-calculated, so the entry for each module in the language representation included the declarations of all symbols accessible in that module instead of just those symbols declared in the module. Although computationally efficient, this made the language representation very large because each declaration had to appear many times. This is a valid approach, but the space saving seems to be more important than the small additional computation. The size of the representation is however a critical factor at higher meta-levels, where the representation is itself represented; unless special techniques are used these terms quickly become very large indeed.

[2]Due to Alistair Burt

The other argument of the `ModuleDefinition` structure is a term describing all the symbols declared in this module. This term has type `CategoryTable`.

```
FUNCTION Categories :
   AVLTree(List(SymbolDescriptor))      % Symbols declared as Bases
                                        % or Constructors.
 * AVLTree(List(SymbolDescriptor))      % Symbols declared as Constants,
                                        % Functions, Propositions or
                                        % Predicates.
-> CategoryTable.
```

As a slight optimisation, this dictionary is split into two parts. Since it is part of the flat name, we can use the category of a symbol as an index to reduce the overhead of locating its entry in the language representation. However, we have also to meet requirement 2 and it is crucial to the usability of the system that the parser is as fast as possible. When the parser meets a symbol in its input, it can't tell the category of the symbol in advance, but it can tell from the context whether the symbol is part of a type or part of a formula. We can therefore divide the symbols into two classes without compromising the efficiency of the parser, whereas if we separated all six categories, the parser would have to search four dictionaries for every symbol it encountered while parsing a statement or goal.

The two AVL-trees associate the declared name of a symbol with the list of declarations with this declared name in this module and class. Linear search is completely adequate here; overloading is uncommon so there is usually only one entry in the list. The representation used for declarations encodes the category, and where appropriate the arity, of the symbol so that the correct declaration is uniquely identified by the flat name.

Every symbol is described by a structure of type `SymbolDescriptor` with the following function:

```
FUNCTION Symbol :
   Accessibility
 * Declaration
-> SymbolDescriptor.
```

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

Finally, there are constants of type `Declaration` to represent the declarations of base and proposition symbols, and functions with range type `Declaration` to represent the declarations of constructor, constant, function and predicate symbols together with the attributes, such as type, that these declarations provide.

```
CONSTANT BaseDecl, PropositionDecl : Declaration.
```

```
FUNCTION ConstructorDecl :
   Integer                      % The arity of the constructor.
-> Declaration.

FUNCTION ConstantDecl :
   Type                         % The type of the constant.
-> Declaration.

FUNCTION FunctionDecl :
   Integer                      % The arity of the function.
 * FunctionInd                  % Its indicator.
 * List(Type)                   % Its domain type.
 * Type                         % Its range type.
-> Declaration.

FUNCTION PredicateDecl :
   Integer                      % The arity of the predicate.
 * PredicateInd                 % Its indicator.
 * List(Type)                   % Its domain type.
-> Declaration.
```

The types `FunctionInd` and `PredicateInd` and the constants and functions that define them are declared in the export part of `Programs` and so are public. These represent the indicator component of functions and predicates declared in the object program in a straightforward way.

5.3 Representing the Program Code

The final element of the representation is the fourth argument of the `Program` function, the dictionary containing representations of the object program's statements and control declarations. By looking up the flat name of a predicate symbol in this dictionary all the statements in the definition of the predicate, and all the delay declarations for the predicate can be found. Note that there is no predicate exported by the `Programs` module that is capable of returning the statements (or delay declarations made in the local part) of a closed module; these are completely protected from user access. In practice we do not need to represent them, and only the delay declarations that appear in the export parts of the closed modules are actually present in the entries for these modules in the code dictionary.

It is convenient to organise the code dictionary analogously to the language dictionary, with two nested AVL-trees exploiting the flat name structure. The outer structure, keyed on the module name, has type `AVLTree(ModuleCode)`; the `ModuleCode` type is defined by

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

The `Integer` argument here is not part of the representation in logic, but is used in the implementation of `Succeed` and the predicates similar to it that simulate Gödel computations for representations of object goals with respect to representations of object programs. Rather than interpreting the representation directly, it is more efficient to reflect the representation of program and goal down to the object level, perform the computation directly to obtain a computed answer, and reflect this answer back up to its meta-level representation. The programmer is of course completely unaware of this implementation technique. To facilitate this technique, and reduce the overhead of reflection, the system retains the code compiled by the reflection process and only recompiles a module of an object program when necessary. Recompilation is required when the module represented in the object program to be executed is different from a module with the same name that was previously reflected to produce the compiled code. This integer argument acts as a version number and allows the system to keep track of such changes.

Dividing the representation of the object program code into separate modules allows the reflection operation to recompile module by module only those modules that have changed since the last time they were reflected, rather than having to recompile the whole object program. A better scheme would be to actively reflect only predicates whose definition has changed, or even to operate statement by statement. This leads naturally towards a representation for dynamic object theories like that used in MetaProlog [1].

The code portion of each module is represented by a term of type

```
AVLTree(List(PredicateDefinition))
```

which is keyed by the declared names of predicates declared in the module. Overloading of course means that there can be more than one such predicate per declared name, but these can be distinguished by their arities. Hence there is a list of `PredicateDefinition` terms for each declared name, each with the function

```
FUNCTION PredDef :
   Integer                      % The arity of the predicate.
 * List(Formula)                % List of statements in the definition.
 * List(Delay)                  % Delay declarations in the export part
 * List(Delay)                  % Delay declarations in the local part.
-> PredicateDefinition.
```

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

The base type `Delay` is not exported by `Programs`. Instead delay declarations appear to the programmer in two parts: the head atom, represented by a term of type `Formula`, and the condition, represented by a term of type `Condition`. The `Delay` function joins these components in an `Delay` term.

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

Delay conditions are represented by terms built from functions of type `Condition` in an obvious way.

```
CONSTANT TrueCond : Condition.

FUNCTION Nonvar, Ground :
   Term
-> Condition.

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

6 Conclusion

We have described a scheme for repesenting Gödel object programs in Gödel. This is not the fearsome task it might appear to be at first sight. By presenting the representation to the programmer as an abstract data type, providing an exhaustive library of predicates to perform standard operations on the data type, and also providing a program compiler to generate the representation from the object program source, the labour and complexity of writing meta-programs in the ground representation is considerably reduced. When writing a meta-program in Gödel one can start on the problem straight away, without having to set up the representation first.

A prototype implementation of the Gödel modules `Syntax` and `Programs` has been produced, based on the above representation. It has been and is being used successfully by students and researchers at Bristol and elsewhere to build programs such as interpreters, theorem provers, partial evaluators, declarative debuggers, and knowledge assimilators.

The reader's attention is especially drawn to [4], in which some experiments in partial evaluation of Gödel meta-programs are reported. These suggest that, if the meta-program is carefully written, a large part of the computational overhead of using the ground representation can be compiled away. Partial evaluation is also effective in reducing the overhead that results from using an abstract data type when the underlying engine is highly optimised for pattern matching. In addition, we have the prospect of increased parallelism obtained by eliminating the sequential properties of Prolog's extra-logical features. Finally, a small computational overhead seems a more than reasonable price to pay for delarative semantics.

More research needs to be done on the topic of compilation techniques for meta-programs, in order to reduce the overhead further. It will also be interesting to consider how the underlying system can provide support for the ground representation, for example by using implicit reflection to implement meta operations such as unification of object terms.

This research described here is, of course, only the first step on the road to effective use of the ground representation, but we contend that we *can* have the full power of the ground representation with at least the convenience and (given sophisticated compilation

techniques and parallelism) efficiency of the non-logical route; and we can have all this without straying from first-order logic.

Acknowledgements

I am particularly indebted to John Lloyd and Pat Hill for designing the Gödel language and encouraging this research; they also commented on drafts of this paper. Special thanks are due to Jiwei Wang for developing a fast parser (with excellent diagnostics) and compiler for Gödel, and to Corin Gurr who demonstrated the value of partial evaluation in compiling Gödel meta-programs. Thanks also to Dominic Binks, Nick Moffat and André de Waal for the benefit of their practical experience in making use of this work. This research was supported by ESPRIT Basic Research Action 3012 (Compulog).

References

[1] K.A. Bowen and T. Weinberg. A meta-level extension of Prolog. In *Proceedings of 1985 Symposium on Logic Programming,* Boston, pages 669–675, 1985.

[2] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1986.

[3] I. Cervesato and G.F. Rossi. Logic meta-programming facilities in 'LOG. In Alberto Pettorossi, editor, *Proceedings of the Third International Workshop on Metaprogramming in Logic*, June 1992.

[4] C.A. Gurr. *Specialising the Ground Representation in the Logic Programming Language Gödel*. Technical Report CSTR-92-30, University of Bristol, November 1992.

[5] 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.

[6] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. Technical Report CSTR-92-27, University of Bristol, October 1992.

[7] 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.

[8] 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.

[9] Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[10] 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.

[11] M. van Emden. AVL Tree insertion: a benchmark program biased towards Prolog. *Logic Programming Newsletter*, 2, Autumn 1981.

[12] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.