Towards Fast and Declarative Meta-programming

Antony F. Bowers and Corin A. Gurr

January 1995

Abstract

Meta-programming, the ability to manipulate programs as data, is fundamental to the success of declarative languages. Regardless of the choice of logic, without properly representing object variables as ground terms, declarative meta-programs are severely limited. However, use of the ground representation seems to involve unacceptable programming effort and computational overhead. We examine the underlying reasons for these problems, and investigate techniques for addressing them so that the dream of declarative meta-programming can be realised. Gödel is a programming language based on first order logic which is intended to have better declarative semantics than Prolog. Taking Gödel interpreters as typical meta-programs, we show how awareness of efficiency issues in the interpretation algorithm suggests a particular programming style. The careful design and construction of the Gödel system modules has made writing such interpreters straightforward, and also made them amenable to partial evaluation. By using mutable data structures to represent programs and substitutions, it is possible to optimise these interpreters further until they approach the speed of interpreters using the traditional non-ground representation. The details of these optimisations may remain entirely hidden from the programmer. The efficiency gains apply equally to interpreters performing tasks that require the full power of the ground representation, and so have no declarative non-ground equivalent for comparison.

1 Introduction

Declarative meta-programming is vital for the future of computing. If the use of computers is to continue to expand, but the task of programming them is not to occupy the entire working population, we must eventually learn to teach them to create and maintain their own software. Declarative programming languages surely offer the best hope that machines will one day be able to assist us in mastering the complexity of their own programming, because it is declarative programs that are most readily understood and manipulated by other programs. These meta-programs must also take their turn as data, and so should be declarative.

The full potential of a truly declarative language for meta-programming is sometimes overlooked. Hill and LLoyd [HL94] summarise declarative programming as being "much more concerned with writing down *what* should be computed and much less concerned with *how* it should be computed". This approach has two major benefits of particular relevance to meta-programming. The first is that declarative meta-programs, particularly large programs, can be much easier to understand than comparable procedural meta-programs. The second advantage becomes apparent when writing meta-programs which manipulate object programs or theories which are themselves declarative. Such meta-programs often need pay little or no concern to the procedural semantics or non-logical quirks of the object programs which they manipulate. As a consequence these meta-programs can employ far simpler algorithms than would be needed to manipulate comparable non-declarative object programs.

It is well-known that declarative meta-programming requires carefully representing the language of the object program in the language of the meta-program, and that this can be done in two ways,

the non-ground and ground representations described by Hill and Lloyd [HL89]. The non-ground representation is easy to implement efficiently and easy to program, but its power to manipulate object programs is extremely limited because object variables cannot be inspected. Regardless of the particular choice of logic, without the representation of object variables by ground terms, declarative meta-programming is not powerful enough to implement sophisticated meta-programs such as program transformers. However, at first sight the management of object variables in the ground representation involves unacceptable programming effort and computational overhead, so in current Prolog practice object variables are usually represented by meta-variables and handled by means of non-logical operations. Consequently the declarative property of meta-programs is lost.

Gödel is a programming language based on first-order logic and intended to have better declarative semantics than Prolog. Gödel places particular emphasis on declarative meta-programming, and provides considerable support for programmers wanting to manipulate ground representations of Gödel object programs. Gödel has a type system based on *polymorphic many-sorted logic*, and a module system. 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. Gödel predicates may have control declarations, called `DELAY` declarations. These are syntactic variants of the `when` declarations of NU-Prolog (Thom and Zobel [TZ88]). It is not our intention to describe the Gödel language here. For a complete technical description and specification the reader is referred to Hill and Lloyd [HL94].

A Gödel system has been implemented at Bristol. This implementation (known as "Bristol Gödel") is based on a compiler that translates the Gödel source into SICStus Prolog. The experiments described in this paper were conducted on this system.

Meta-programs using the ground representation incur a considerable overhead due to the necessity of explicitly managing the ground variables. In this paper we investigate possible techniques to reduce this overhead in Gödel. These techniques include partial evaluation and the use of arrays with destructive assignment to represent substitutions. We take simple interpreters as examples of typical meta-programs, as the performance of interpreters is generally critically dependent on the efficiency with which variables are handled.

The central problem we are struggling with is to find a way to relate the complex and laborious high-level manipulations performed by an interpreter executing SLD-resolution on the ground representation of an object program, to the relatively simple and efficient operations performed by the underlying language implementation, which is essentially doing the same thing. If the interpreter can somehow be made to copy the low-level mechanism without sacrificing the declarative semantics, it can be made efficient. Remember that the declarative semantics of a meta-interpreter is the procedural semantics of the object program. It is therefore not surprising that our optimisations lead to interpreters that resemble compiled Prolog.

The standard Prolog implementation, using the WAM (Warren [War83]), is unsuitable as a basis for interpreting programs in the ground representation as it has too many optimisations which are specific to resolution without the occur check in a depth-first backtracking system. The only interpretation mechanism previously formulated for the ground representation was the explicit *unify-compose-apply* style of resolution presented in the interpreters of first Bowen and Kowalski [BK82] and then Hill and Lloyd [HL89]. The course of this paper follows the historical course taken in the development of the current implementation of the ground representation where successive development steps have led to an implementation which is similar, but not identical, to the WAM.

Of course, an essential feature of meta-programming in Gödel is that all the cunning is hidden from the programmer inside the library modules provided by the system. In fact, the style of meta-interpreter that emerges from these considerations, once it becomes familiar, can be seen to be very much simpler than traditional interpreters for the ground representation, and is analogous to the Vanilla interpreter for the non-ground representation.

In the next section we look briefly at the design of the ground representation in Gödel, and the important issues arising from the simulation of unification in the ground representation. In section 3 we consider the design of some simple Gödel meta-interpreters and compare their performance. Finally, in section 4 we investigate the performance improvements that can be obtained by partial evaluation and by adding special-purpose low-level support to the implementation. Although performed in a crude prototype implementation, these experiments show a dramatic performance improvement which clearly promises that the techniques we outline have the potential to make meta-programming in the ground representation fully practical.

2 The Ground Representation in Gödel

2.1 Syntax and Programs

Anyone attempting to write a meta-program using a ground representation in a logic programming language will immediately meet the first difficulty: the labour involved. A lot of routine decisions have to be made concerning the details of the representation, and a lot of routine programming work is required to implement basic operations on the representation such as unification.

To solve this problem, Gödel provides a fixed representation for Gödel programs, in the form of a collection of abstract data types so that programmers need not be concerned with the details of the representation. Two of Gödel's system modules, `Syntax` and `Programs`, export an extensive library of basic operations on these abstract data types. There are disadvantages in a fixed representation, as argued by van Harmelen [vH92], but the facilities provided are intended to be sufficient to cover most meta-programming tasks involving manipulations of Gödel object programs, and to assist the programmer in achieving efficiency in a large proportion of these tasks. Some of the design issues arising from this latter aim are the topic of this paper. The alternative, allowing the programmer to design the representation, would surely also return a good part of the labour to the programmer.

The `Syntax` module is concerned with the manipulation of syntactic expressions in the language of the Gödel object program. It exports the types `Name`, `Term` and `Formula`, which are the types of terms representing object symbols, terms and formulas respectively.

The `Programs` module is concerned with the representation of complete Gödel object programs as terms in the meta-program. It exports the type `Program` for this purpose. We do not deal with the detailed structure of these program terms here, the interested reader is referred to Bowers [Bow92]. It is sufficient to note that in the Bristol implementation the module structure, language declarations, control declarations and statements of the object program are stored in balanced binary trees, permitting them to be accessed in logarithmic time.

We now take a brief look at some of the constants and functions used to form the ground representation in the `Syntax` module, as this will help to make the following discussion concrete. In `Syntax`, we are concerned with the representation of the syntactic structures of the object program. Let us begin with the symbols of the object programs's alphabet. Every symbol in a Gödel program has two names. Firstly, it has a *declared name*, which is the name given by the programmer when the symbol is declared. This is the name used wherever the symbol appears in the program, and for simplicity a symbol is usually identified with its declared name. Secondly, it has a *flat name*, which is an internal name used to identify symbols uniquely. Gödel allows generous overloading, and consequently the declared name of a symbol is not sufficient to identify it unambiguously. A flat name has four components: the name of the module in which the symbol is declared, its declared name, its category (base, constructor, constant, function, proposition or predicate) and its arity. Language conditions ensure that this quadruple uniquely identifies the symbol. It is the flat name that is represented by in the `Syntax` module.

The abstract data type `Name` is provided to represent symbol names. The `Name` type is the basic

unit from which the representation of syntax is constructed. 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:

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

The name `Name` is overloaded here: it can stand for both a type symbol and a function symbol. This is a convenient programming style, since it avoids the proliferation of symbols with names that are artificially differentiated simply because they are in different categories. Overloading like this is common in natural language, and the readability of the program is not compromised because it is always obvious from the context whether a type symbol or a function symbol is intended.

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

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

For example, let there be a constant with declared name `January` in a module called `Calendar`. The 4-tuple <`Calendar`, `January`, *constant*, 0> is the flat name of this constant. At the meta-level, we can represent the declared name by the string `"January"` and the module name by the string `"Calendar"`.[1] The representation of the symbol `January` is then the ground term `Name("Calendar", "January", Constant, 0)`.

Terms in the object language are represented by terms of type `Term` in the meta-language:

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

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

The `Term` function is used to represent a compound term, and includes a list of representations of its arguments, and the `CTerm` function is used to represent a constant. Object atoms by are similarly represented by terms of type `Formula`:

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

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

[1] Strings are constants of type `String` exported by the module `Strings`. There are infinitely many, one for every possible sequence of characters.

Of course, since this is a *ground* representation, object level variables are represented by ground terms at the meta-level. The particular variable represented is conveniently identified by a `String` constant.

```
FUNCTION Var :
   String                          % Variable name.
-> 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 `~'` and binary functions `&'`, `\/'`, `->'`, `<-'` and `<->'`.

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

```
Append([], x, x) <-
```

in the module `Lists`. It is:

```
Atom(Name("Append", "Lists", Predicate, 3),
   [CTerm(Name("Nil", "Lists", Constant, 0)),
      Var("x"), Var("x")])
<-' Empty
```

Notice that this representation is very much larger than the formula it represents, both syntactically and in terms of it's internal form within a typical WAM-style implementation. If term with n arguments requires $n+1$ machine words plus the size of its arguments for its object level representation, its meta-level representation would require $2n+9$ machine words plus the size of the representation of its arguments.

Notice also that this representation is fully general in terms of permitting the representation of partially known structures. That is, meta-variables can appear in place of the `Name` terms that represent function or predicate symbols, or in place of some terms representing arguments, or in place of some or all of the argument list. For example, it is straightforward to use the facilities of the `Syntax` module to create a partial representation such as `Term(y, [Var("w")|z])`, where `y` and `z` are meta-variables. This represents some compound object term which has the variable `w` as its first argument, but about which nothing else is known. Although control declarations could prevent the creation of partial representations, this restriction has been avoided to give the programmer maximum flexibility. Partial representations are occasionally useful, for example in program synthesis, where they can act as constraints upon parts of a program under construction as described by Christiansen [Chr94]. Programmers working with the traditional Prolog style of meta-programming cannot normally represent terms with variable function symbols or arities. Some Prolog systems may permit terms with variable function symbols, but of course such a feature is outside first-order logic.

For compactness and readability, in what follows we shall usually omit the bulky `Name` terms that Gödel uses to represent symbols, and in their place write the declared name of the represented symbol with a prime. For example, `F'` is to be read as

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

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

2.2 Unification and Substitutions

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

An example of common Prolog meta-programming practice is given by programs that perform some variant of the default unification process, perhaps by adding the occur check. They usually work by using the built in primitives `functor/3` and `arg/3` to compare the functors and arguments of the input terms recursively. This type of program is often used for teaching; several examples can be found in Sterling and Shapiro [SS86]. These programs still rely on, and obtain their efficiency from, the underlying system's treatment of the meta-variables that stand for object variables. It goes without saying that these techniques are not declarative, and one consequence of this is that the programs change their input arguments (by binding the meta-variables they contain) in a way that seems awkward once a declarative style of meta-programming has become familiar.

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

Contrast this with the ground approach. Here we want to perform a meta-level simulation of unification on the representations `Term(F', [CTerm(A'),Var("x")])` and `Term(F', [Var("v"),(Var("v")])`. Beginning by comparing the terms from left to right, we soon need to record the binding of variable `v` to constant `A`. In non-ground, this was done by simply instantiating the meta-variable standing in place of the representation of `v` to the constant `A'` representing `A`, and of course all the other occurrences of this same meta-variable representing `v` were also instantly affected. If we try, naively, to proceed using an analogous algorithm in the ground case, we need to replace all occurrences of `Var("v")` by `CTerm(A')` in the representations of both terms. However, since the representation of `v` is a ground term, rather than a meta-variable, we cannot do this destructively as in non-ground. Instead, the process of searching both representations for occurrences of `Var("v")` and replacing each one with `CTerm(A')` will construct a complete new copy of the representation of each term. Now notice that this has been done simply to record a single variable binding. Doing this every time a variable is bound clearly involves an unacceptable overhead in space and time.

Recording bindings by constructing a representation of a substitution looks a more promising approach. Gödel's `Syntax` module exports an abstract type `TermSubst` for representations of substitutions, and predicates `UnifyTerms` and `UnifyAtoms` that take a pair of terms or atoms respectively and return the representation of their mgu. `Syntax` also provides predicates for other standard operations on substitutions, such as application, composition, and adding and deleting individual bindings. The intended interpretations of these predicates conform to the definitions in Lloyd [Llo87].

The direct and obvious way to represent substitutions is as lists of bindings.

```
CONSTRUCTOR Binding/1.
```

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

So the term `Subst([Var(v1)/t1,...,Var(vn)/tn])` represents the object level substitution $\{v_1/t_1, \ldots, v_n/t_n\}$, where `Var(v1)`,...,`Var(tn)` represent the variables $v_1, \ldots, v_n$ and `t1`,...,`tn` are the representations of the terms $t_1, \ldots, t_n$ respectively.

During the unification process, it is necessary both to look up bindings in the substitution whenever a variable is encountered, and to add new bindings to the substitution as they are made. Looking up a binding means conducting a linear search of the list. Adding the binding of variable v_m to term t_m to the substitution θ already constructed means forming the composition $\theta\{v_i/t_i\}$. If θ is $\{v_1/t_1, \ldots, v_n/t_n\}$, then all occurrences of v_m in $t_1, \ldots, t_n$ must be replaced by t_m. Performing this manipulation on the representation `[Var(v1)/t1,...,Var(vn)/tn]` will necessarily involve constructing a copy of the list on current Prolog systems. We will return to this problem in the context of interpreters in the next section.

Two more possibilities suggest themselves for efficient simulation of unification in the ground representation: structure reuse and reflection. Structure reuse using liveness analysis techniques such as described by Mulkers [Mul93] might, in theory, permit fast unification by destructive assignment to variable representations in one or both of the structures to be unified, but there is a long way to go before practical analyses are accurate enough to make this work well.

Implementing unification by reflection, that is by actually performing the unification at the object level, is more promising. However, it suffers from the fact that in purely declarative meta-programming, the reflective implementation must be completely hidden. One possibility involves constructing the object level structures denoted by the ground representation, unifying them using the efficient unification routine in the underlying system, and then mapping the result back into the ground representation. However, the translation step is quite expensive, and since the operation is internally non-logical, it's logic cannot be made available to general optimisation processes such as partial evaluation and static analysis. These are important disadvantages. It's also possible to imagine an alternative reflective implementation, in which object level structures are maintained in parallel with their representations at the meta level. The difficulty here is that the object level unification essentially destroys these parallel structures, so copies of the structures to be unified must again be made before unifying them. Performing compile time analysis to determine when this is necessary leads back to the structure reuse idea.

3 Simple Interpreters

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

Even simple SLD-interpreters for the ground representation add value over the non-ground Vanilla interpreter, because the result is returned in the form of the ground representation of a computed answer substitution, rather than by instantiating meta-variables, and this is potentially more useful. In addition, it is straightforward to extend our SLD-interpreters, for example to return a proof term, or to return all the input clauses used in a derivation as part of a knowledge assimilation procedure like those described by Guessoum and Lloyd [GL90], [GL91]. The latter task requires the ground representation if it is to be done declaratively. We have not included enhanced

```
PREDICATE Demo : Program * Formula * Formula.

Demo(program, goal, goal_instance) <-
  InstanceOf(goal, goal_instance) &
  IDemo(goal_instance, program).

PREDICATE IDemo : Formula * Program.

IDemo(empty, _) <-
  EmptyFormula(empty).

IDemo(conjunction, program) <-
  And(left, right, conjunction) &
  IDemo(left, program) &
  IDemo(right, program).

IDemo(atom, program) <-
  IClause(clause, program) &
  IsImpliedBy(atom, body, clause) &
  IDemo(body, program).
```

Figure 1: Instance Demo

interpreters in the discussion, because they are so easily imagined by the reader, and because the additional complexity might obscure the argument.

While we use only simple interpreters to illustrate the efficiency issues for declarative programming in this paper, this in no way implies that the optimisation techniques we propose are applicable only to such programs. The techniques described in this paper have been used in the construction of a range of sophisticated Gödel meta-programs, including coroutining interpreters, theorem provers, abstract analysis tools, declarative debuggers and program specialisers such as partial evaluators.

3.1 Instance Demo

In passing, it is worth discussing the interpreter proposed by Hill and Gallagher [HG94], called *Instance Demo*. Instance Demo is an interpreter for programs in the ground representation, yet it uses meta-variables for unification and relies on the underlying system to manage variable bindings and backtracking in exactly the same way as the Vanilla interpreter does in the non-ground representation. Figure 1 shows part of this interpreter, written using Gödel's meta-programming facilities. The predicate `InstanceOf` is intended to be true when its second argument represents an instance of the formula represented by its first argument. However, when called with the second argument unknown, its effect is to return the formula in the first argument with all the representations of variables replaced by meta-variables. The predicate `IClause` uses `InstanceOf` to find instances of program clauses.

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

At first sight this looks a very promising way to obtain efficient interpreters for the ground representation. However, because Instance Demo is so similar to non-ground based interpreters, it suffers similar limitations. It effectively captures the declarative semantics of the object program

```
...
Select(goal, left, sel, right) &
StatementMatchAtom(prog, _, sel, stat) &
RenameFormulas(avoid, [stat], [stat1]) &
IsImpliedBy(head, body, stat1) &
UnifyFormulas(head, sel, mgu) &
ComposeTermSubsts(so_far, mgu, subst) &
And(left, body, goal1) &
And(goal1, right, goal2) &
ApplySubstToFormula(goal2, mgu, new_goal) &
...
```

Figure 2: Part of the main loop of an interpreter using composition

but not the procedural semantics: it cannot be used to vary the search strategy much from that of the underlying system, and it cannot give information about variable bindings. The answer it provides to the object program and goal is some instance of the original goal, but that instance may be incompletely specified (it may contain meta-variables), which makes it much less useful than an answer in the true ground representation.

3.2 SLD Interpreter with Composition

If we must build interpreters that deal directly with the undiluted ground representation of the object program, how should such interpreters work? Although there are relatively few examples in the literature, most of those that exist, for example in Bowen and Kowalski [BK82] or Hill and Lloyd [HL89], are based on explicitly mimicking the standard mechanism of SLD-resolution (as it appears for example in Lloyd [Llo87]).

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

Figure 2 shows part of an interpreter using composition, constructed from some of Gödel's meta-predicates. This interpreter has to do an enormous amount of work in each resolution step. As we have seen, the unification operation involves linear search within the representation of the partially constructed mgu, and making copies of it, but the mgu is usually quite small. The explicit `Compose` operation, however, must copy the entire substitution computed up to this point, and this contains bindings for many variables that have been introduced, used and subsequently dropped from the computation. This substitution can become very large indeed, and copying its representation is a major overhead. Renaming the variables of the input clause necessitates making a new copy of the clause, and similarly, the application of the mgu to the new goal necessitates copying the goal. All these copying operations are expensive in time and space, and as might be expected, the performance of this interpreter is a disaster.

We will now look at some ways to reduce this extra work.

3.3 Improvements: Unify Demo

The first priority is to avoid composing substitutions at all. We achieve this by altering the representation of substitutions, so that they are represented in an effectively *uncomposed* form. When a new binding is added to a substitution in this representation, it is simply added to the set of existing bindings and the rest of the substitution is unchanged. In this way, reference chains are built up, similar to the reference chains that form on the stack in a standard WAM implementation. To determine the result of applying a substitution in this form to some term, it is necessary to repeatedly follow all reference chains until a result is obtained that contains only variables for which there are no bindings in the substitution. For example, the the classical substitution $\{w/z, x/z, y/z\}$ might be represented by `[Var("w")/Var("x"), Var("x")/Var("y"), Var("y")/Var("z")]`.

This less literal representation of substitutions differs from the classical representation, in that adding an arbitrary binding could lead to a loop in a reference chain. It is also not clear how to compose two substitutions in this representation; one way would be to translate them into classical substitutions first, by collapsing all the reference chains, and then use the classical composition algorithm. Naturally, this would be an expensive operation, but the point of this optimisation is to avoid composition entirely. If we limit the available operations on substitutions to creating an empty one, unification (with the occur check), and application, this representation is well behaved. This is easily done because the representation is an abstract type.

The representation of substitutions in uncomposed form is alone sufficient to avoid composition during unification, that is within a call of `UnifyTerms`, but during interpretation we must also avoid the explicit composition of the new mgu with the accumulated answer substitution so far computed. Passing the accumulated substitution to `UnifyTerms` as an input argument solves this problem; as `UnifyTerms` generates new bindings during the unification process, it can simply add them to the list of bindings already created by previous unifications.

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

We introduce a new set of variable representations, in which the variables are uniquely identified by integer indexes. These are in addition to the `Var("x")` form, which is still required to represent all other variables, and these are all taken to have index 0. The new representations use the function

```
FUNCTION V : Integer -> Term
```

and the meta-terms `V(1)`,`V(2)`,`V(3)`... are taken to represent the variables `v_1`,`v_2`,`v_3`... The root `v` is simply an arbitrary choice to construct syntactically correct Gödel variables. While it would not be practical to expect the authors of object programs to confine their choice of variable names to this limited set, we can ensure that all new variables introduced by a Gödel interpreter (chiefly when renaming clauses) are of this form.

Now we have the possibility of using the integer index to obtain faster look up in the substitution, by using a declarative array implementation to represent the binding list. In practice we use a carefully designed and optimised binary tree, giving logarithmic access and update times.

We can also use the variable indexes to simplify the interpreter considerably, by reducing the information needed for safely renaming the input clause. Instead of carrying around formulas containing the variables to be avoided, (for example, in the heads of resultants), we merely have to remember the smallest variable index that has not previously been used. These considerations lead to the type of interpreter shown in Figure 3. The structure of this interpreter is analogous to that of the Vanilla interpreter, and it shares the selection rule and search strategy of the underlying

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

Demo(_, empty_goal, sp, sp, subst, subst) <-
  EmptyFormula(empty_goal).

Demo(program, goal_atom, sp, new_sp, subst, new_subst) <-
  Atom(goal_atom) &
  StatementMatchAtom(program, goal_atom, statement) &
  StandardiseFormula(statement, sp, sp1, statement1) &
  IsImpliedBy(head, new_goal, statement1) &
  UnifyAtoms(goal_atom, head, subst, subst1).
  Demo(program, new_goal, sp1, new_sp, subst1, new_subst).

Demo(program, goal, sp, new_sp, subst, new_subst) <-
  And(left, right, goal) &
  Demo(program, left, sp, sp1, subst, subst1) &
  Demo(program, right, sp1, new_sp, subst1, new_subst).
```

Figure 3: Unify Demo

system. It still performs explicitly the standard steps of input statement selection, standardisation apart, and unification but it is much simpler and more efficient than the composition interpreter. However, there is one more major optimisation that we can perform, and that is described in the next section.

3.4 SLD Demo and `Resolve`

Gödel supplies a system predicate, `Resolve`, which provides an optimised implementation of resolution. When resolving an atom with respect to some statement in an object program we must first rename the variables in the statement, replacing them with new variables which do not occur elsewhere in the current computation. The implementation of `Resolve` may take advantage of this knowledge to optimise the resolution process. The fact that these new variables are unique to the renamed statement allows `Resolve` to simplify many of the necessary unification operations and reduce the amount of occur-checking.

Figure 4 shows SLD Demo, a very simple Gödel meta-interpreter for definite programs which uses the Gödel predicate `Resolve` to resolve an atom in the current goal with respect to a statement selected from the object program.

The atom `Resolve(goal_atom,statement,v_in,v1,subst,subst1,new_goal)` performs the resolution of the atom `goal_atom` with the statement `statement`. The integers `v_in` and `v1` are used to rename the statement with `v_in` being the integer value used in renaming before the resolution step is performed and `v1` being the corresponding value after the resolution step has been performed. The representations of term substitutions `subst` and `subst1` represent respectively the answer substitution before and after the resolution step. The last argument, `new_goal`, is the representation of the body of the renamed statement.

Figure 5 shows the performance of Unify Demo and SLD Demo interpreting (as object program) the well known Naive Reverse program on a list of 50 elements. The ground interpreters are compared with the object program executed directly on the Gödel system, and with the non-ground Vanilla interpreter interpreting the same program. Approximate performance data is adequate for

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

Demo(_, empty_goal, v, v, subst, subst) <-
  EmptyFormula(empty_goal).

Demo(program, goal_atom, v_in, v_out, subst, new_subst) <-
  Atom(goal_atom) &
  StatementMatchAtom(program, goal_atom, statement) &
  Resolve(goal_atom, statement, v_in, v1, subst, subst1, new_goal) &
  Demo(program, new_goal, v1, v_out, subst1, new_subst).

Demo(program, goal, v_in, v_out, subst, new_subst) <-
  And(left, right, goal) &
  Demo(program, left, v_in, v1, subst, subst1) &
  Demo(program, right, v1, v_out, subst1, new_subst).
```

Figure 4: SLD Demo

Program	*Relative time for Naive Reverse*
Object	1
Vanilla	10
Unify Demo	280000
SLD Demo	31000

Figure 5: Performance of Interpreters

the purposes of this discussion, so we need only consider the results from a single benchmark.

The table indicates that SLD Demo is faster than Unify Demo by a factor of 10, demonstrating the effectiveness of combining renaming and unification in the `Resolve` predicate and optimising them. However, the interpreter is still 3 orders of magnitude slower than its non-ground equivalent, and obviously an overhead of this magnitude renders such interpreters almost useless for most practical applications. Note, however, that the interpreter and the system predicates it depends upon are written almost entirely in Gödel (a very small fraction is written directly in Prolog), and the Gödel code is translated to Prolog by the Gödel compiler. The Prolog system has no built-in support for the ground representation. In the remainder of this paper we consider two strategies for improving the performance of ground interpreters: compiling away some redundant computation by specialising the interpreter to its object program by partial evaluation; and providing in-built support for the ground representation at a low-level in the implementation.

4 Speeding up SLD Demo

4.1 Partial Evaluation

Partial evaluation is a program specialisation technique. A partial evaluator is a tool which takes a program and some partial input data to it and produces a specialised version of that program. A recent overview of partial evaluation and its application to several classes of programming language is given by Jones *et al* [JGS93]. The partial evaluation of logic programs by unfolding was put on a firm theoretical footing by Lloyd and Shepherdson [LS91] and we refer to that paper for a formal definition of partial evaluation for logic programming.

SAGE (Self-Applicable Gödel partial Evaluator), written by Gurr [Gur94], is a partial evaluator based mainly on finite unfolding. SAGE implements techniques developed with the aim of producing a declarative, effectively self-applicable partial evaluator for a full logic programming language (as opposed to some restricted subset of the language). A self-applicable partial evaluator is one which is capable of specialising itself and an *effectively* self-applicable partial evaluator is one which produces significantly improved, efficient residual code upon self-application.

SAGE performs the partial evaluation of a program wrt some goal in four main phases:

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

The primary motivation for the static analysis performed by *SAGE* is as a termination analysis. We produce an abstraction of the partial tree which is used to compute the subsequent partial evaluation and by analysis of this tree we partition the predicates into two sets. In the first set, which we refer to as the *safe* predicates, we place those predicates for which all atoms with this predicate may be unfolded without the risk of leading to an infinite unfolding. The complement of this set, the *unsafe* predicates, contains those predicates for which unrestricted unfolding of atoms with this predicate could not be guaranteed to terminate. This prior identification of the unsafe predicates permits SAGE to apply a more cautious strategy when unfolding them during the partial evaluation phase.

The main post-partial evaluation optimisation performed by *SAGE* is the removal of redundant terms. In general we expect to have constructed the partial evaluation of a set of atoms that define

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

4.2 Implementation of Resolve

The implementation of `Resolve` must handle the following operations:

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

Each of these four operations is potentially very expensive when we are dealing with the explicit representation of substitutions and so the implementation of `Resolve` is optimised to make them highly efficient.

After the statement is renamed, any variables in it are guaranteed not to appear in either the current goal or the current substitution. This means that any bindings for variables in the head of the statement may be applied to the body of the statement and then discarded. Consequently only that part of the mgu of the atom and the renamed head of the statement that records the bindings of variables in the atom will need to be composed with the current substitution in order to produce the new substitution.

Having performed the unification of an atom with the head of a statement we must (in theory) combine the mgu of this unification with the current substitution. In practice it is more efficient for any bindings made to variables in the atom to be composed with the current substitution immediately. In order to achieve these compositions Bristol Gödel has a set of highly specialised predicates, each of which performs one specific unification operation.

Implementing `Resolve` based upon a set of highly specialised unification predicates has the added advantage that when partially evaluating a call to `Resolve` wrt a known statement we may unfold the call until the residual code consists of a (possibly empty) conjunction of atoms with these predicates. When we specialise a meta-program such as the interpreter in Figure 4 to a known object program, the statements in the object program will be known. Therefore we may specialise `Resolve` with respect to each statement in the object program. Such a specialisation will remove the vast majority of the overhead associated with explicit unification in the ground representation.

We refer to the specialised unification predicates which `Resolve` is based upon as the *WAM-like predicates* as they are analogous to emulators for instructions in the WAM-engine. As such, these operations may be implemented at a very low level, leading to a computation time for the specialised form of a meta-program, such as that in Figure 4, comparable to that of the object program itself. In the next section we give an example of such a specialisation.

4.3 Applying SAGE to SLD Demo

Figure 6 illustrates the result of specialising the `Demo` interpreter of Figure 4 with respect to the standard `Append` program. In this specialised version of the interpreter we have made three optimisations:

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

Object program:

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

Specialised interpreter:

```
Demo_1(Empty',v,v,subst,subst).
Demo_1(left &' right,v_in,v_out,subst_in,subst_out) <-
  Demo_1(left,v_in,new_v,subst_in,new_subst) &
  Demo_1(right,new_v,v_out,new_subst,subst_out).
Demo_1(Atom(Append',[arg1,arg2,arg3]),v_in,v_out,subst_in,subst_out) <-
  GetConstant(arg1,Nil',subst_in,s1) &
  GetValue(arg2,arg3,s1,new_subst)
  Demo_1(Empty',v_in,v_out,new_subst,subst_out).
Demo_1(Atom(Append',[arg1,arg2,arg3]),v_in,v_out,subst_in,subst_out) <-
  GetFunction(arg1,Term(Cons',[sub11,sub12]),mode,subst_in,s1) &
  UnifyVariable(mode,sub11,v_in,v1) &
  UnifyVariable(mode,sub12,v1,v2) &
  GetFunction(arg3,Term(Cons',[sub21,sub22]),mode1,s1,s2) &
  UnifyValue(mode1,sub11,sub21,s2,new_subst) &
  UnifyVariable(mode1,sub22,v2,new_v) &
  Demo_1(Atom(Append',[sub12,arg2,sub22]),new_v,v_out,
         new_subst,subst_out).
```

Figure 6: Specialisation of `Demo` wrt `Append`

Specialising SLD Demo with respect to a given object program yields very considerable performance improvements. The specialised programs typically run 40 or more times faster than the original code. There are several sources of this improvement. Unfolding the predicates in `Syntax` that manipulate the abstract data type is an important one; this reduces the number of procedure calls and enables direct pattern matching on the constants and functions making up the abstract type. It is also significant that many of these predicates have control declarations which are expensive to evaluate at run-time. These control declarations are eliminated when the predicates they guard are unfolded. There is a considerable redundant computation involved in repeatedly

extracting the input statements from the `Program` term, where they are stored in a binary tree. For a recursive program such as Naive Reverse, the same statement is used many times over; specialisation completely removes this overhead by promoting all the statements of the object program into the clauses of the specialised interpreter, and eliminating the program term entirely. Finally, a specialised unification procedure is generated for the head of each clause in the object program, expressed in terms of the WAM-like predicates that remain from the unfolding of `Resolve`. Thus it can be seen that the partial evaluation of interpreters is analogous to the compilation of logic programs into WAM instructions. In the next section we give a brief description of these WAM-like predicates.

4.4 The WAM-like Predicates

In Bristol Gödel there are seven WAM-like predicates. We illustrate each of these in Figure 7 by specialising `Resolve` to the statement `P(x,x,A,F(y,F(x,A))) <- Q(y)`. The body of this specialised statement contains one atom for each of the WAM-like predicates described below.

Statement: `P(x, x, A, F(y, F(x, A))) <- Q(y).`

Specialised call to `Resolve`:

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

Figure 7: Illustration of WAM-like Predicates

The first three WAM-like predicates unify arguments of the head of the statement with the matching arguments of the atom as follows:

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

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

The third of the WAM-like predicates, `GetFunction`, is more complex than the previous two and so we first give an overview of its operation. If an argument in the head of the statement is a term with

a function at the top level, then there are two cases in which a call to `GetFunction` will succeed. In the first case the atom's matching argument is a variable. We must bind this variable to a renamed version of the term in the head of the statement (the necessary renaming is performed by subsequent calls and not at this point). In the second case the atom's matching argument is a term with a matching function at the top level. In this case the subsequent calls will unify the arguments of the atom's term with the corresponding arguments in the statement's term.

In precise terms, `GetFunction` behaves as follows:

`GetFunction(term,function,mode,subst,subst1)` attempts to unify the atom's term `term` with a term `function`, which has a function at the top level. If `term` is bound in the current substitution to a variable, then this variable is bound to `function` and `mode` is set to `Write`. The term `function` will be instantiated by subsequent calls to a renamed version of the term in the statement to which `term` is now bound. If, when `GetFunction` is called, `term` is bound in the current substitution to a term with a function at the top level which matches that of `function`, then `function` is instantiated to `term` and `mode` is set to `Read`.

The following predicates perform the unification operations necessary for processing the arguments of function terms in the head of the statement. This involves either renaming variables when in `Write` mode or unifying these arguments with the arguments of the matching function term in the atom when in `Read` mode. When the following calls are made in `Write` mode the terms `term` will be uninstantiated variables. When these calls are made in `Read` mode the terms `term` will have been set to terms in the atom being resolved.

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

`UnifyFunction(mode,term,function,mode1,subst,subst1)` in `Write` mode will instantiate `term` to the term `function` and `mode1` is set to `Write`. In `Read` mode this call behaves as for `GetFunction`, attempting to unify the atom's term `term` with a term `function` with a function at the top level. In this and the two subsequent atoms, `subst` is the current substitution and `subst1` is this substitution after the relevant unification step.

`UnifyValue(mode,term1,term,subst,subst1)` in `Write` mode will instantiate `term` to `term1`. A check is made at this point to ensure that this does not introduce a loop into the substitution (an occur-check, in effect). In `Read` mode this call will unify (with occur-checking) the atom's two terms `term1` and `term`.

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

The above seven predicates are analogous to emulators for the WAM instructions GetValue (in the case of `UnifyTerms`), GetConstant, GetFunction, UnifyValue, UnifyVariable and UnifyConstant, after which they are named. Note that a subtle difference in the manner in which the WAM implements the unification of nested function terms and the manner in which `Resolve` implements it means that the WAM (as originally defined by Warren [War83]) does not have an equivalent to the `UnifyFunction` instruction.

4.5 Further Optimisations

Recall that substitutions are represented as uncomposed sets of bindings, and that, since looking up bindings in the substitution is a frequent occurrence during unification, the speed with which the

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

Figure 8: The pattern in which substitutions are passed

Program	*Relative time for Naive Reverse*
Unify Demo	280000
Unify Demo, specialised	500000
SLD Demo	31000
SLD Demo, no control	1980
SLD Demo, specialised	612
SLD Demo, specialised, arrays	216

Figure 9: Effect of optimising interpreters

binding for a particular variable can be located is an important factor in the efficiency of unification. For this reason, we added integer indexes to the representation of variables, and stored the bindings in a binary tree. Suppose we try to use a genuine array to store the bindings, to obtain constant rather than logarithmic access times? The difficulty is, of course, that arrays can only be updated destructively, and this destroys the soundness of the implementation if any references supposed to be to the original array remain after the update. However the update time is also constant, and fast compared with the typically logarithmic update time for a binary tree (because a copy must be made of the branch of the tree from the root to the updated leaf).

Consider the sequence of WAM-like predicates in Figure 7 that typically results from specialising `Resolve`, and the way these operate on substitutions. Figure 8 shows the sequence, with all arguments other than those of type `TermSubst` removed for clarity. As it is relatively simple, take the call of `GetConstant(arg3,CTerm(A'),s1,s2)`. If `arg3` is instantiated to the representation of a variable, this variable is dereferenced by following its binding chain in the input substitution `s1`. If the result of dereferencing `arg3` is another variable, the binding of this variable to constant `A` is recorded by updating substitution `s1` to give the output substitution `s2`. Given that the sequence of WAM-like predicates is executed from left to right, as it must be (DELAY declarations can be added to make sure this is so), if there are no other references to `subst_in` at the time `Resolve` is called, it will be safe to perform this update destructively, and similarly for updates made by all the other instructions in the sequence. We assume that all destructive updates are trailed where necessary, and therefore correctly undone on backtracking. Performing this analysis by hand, we can determine that it is safe to experiment with the use of arrays and destructive assignment for the representation of substitutions in the SLD Demo interpreter and its specialised form.

Figure 9 shows the results of some experiments with specialising Demo interpreters and using arrays to represent substitutions. We use the same Naive Reverse benchmark as before. The SICStus Prolog built-in predicate `setarg/3` was used to provide the destructive array update.

It is interesting to note that specialising Unify Demo actually makes it worse by a significant amount. Because SAGE will not unfold the system predicate `StandardiseFormula` without knowing

```
Demo_1(Atom(Append',v_23),v_17,v_2,v_20,v_4) <-
  StandardiseFormula(
    Atom(Append',[Term(Cons',[Var("x"),Var("xs")]),Var("ys"),
                Term(Cons',[Var("x"),Var("zs")])])<-'
    Atom(Append',[Var("xs"),Var("ys"),Var("zs")]),
    v_17,v_18,v_21<-'v_9) &
  UnifyAtoms(Atom(Append',v_23),
    v_21,v_20,v_22) &
  Demo_1(v_9,v_18,v_2,v_22,v_4).
```

Figure 10: Part of Unify Demo specialised to Append

the initial free index, the residual code contains clauses such as that shown in Figure 10, generated from the second clause of Append. Building the representation of the clause on the heap before calling `StandardiseFormula` turns out to be more expensive than retrieving it from the `Program` term was in the unspecialised version. This example is included to demonstrate the sensitivity of partial evaluation to the way the source program is written.

In Figure 9 the line marked "no control" indicates the speed up obtained by compiling the unspecialised SLD Demo with a special Gödel compiler that ignores all control declarations and does not enforce safe negation, so the resulting program simply executes from left to right. This line shows that the run-time control in the Gödel system by itself costs a factor of 10 at least, and although this is one of the overheads removed by SAGE it is not an overhead created by the use of the ground representation, but rather a limitation of the present implementation. It is possible to remove almost all of the control overhead by performing a static analysis of the program at compile time, and an analyser to do this is under development. With respect to the speed obtained without checking control information, the partial evaluation gains a factor of 3 and the use of arrays another factor of 3, bringing the overall performance within a factor of 20 of the speed of the non-ground Vanilla interpreter.

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

If the use of arrays and destructive assignment is to be a practical tool, some way must be found to ensure that programs remain declarative. For the SLD interpreter and similar programs in which there is just one substitution that is updated sequentially there is no difficulty, but we need to be able to make sure all programs are declarative, and do so automatically. One possibility would be to use an efficient but declarative array implementation, such as the "hairy structures" of Barklund

and Millroth [BM87], which provide mutable data structures with a constant time overhead, and constant time access to the most recent version of the data structure. Another possibility would be to use a sophisticated abstract analysis aimed at deriving liveness information for possible structure reuse, or compile-time garbage collection, such as that of Mulkers [Mul93]. However, such analyses are complex to implement, slow to execute, and a little too general for our present needs.

A scheme that is close to the informal argument given above for the safety of SLD Demo would be ideal. In each clause of SLD Demo, and of the specialised version, there are no more than two occurrences of each variable of type `TermSubst`. This suggests something like the two-occurrence restriction of Janus (Saraswat *et al* [SKL90]), but making use of the Gödel type system to limit the restriction to variables of type `TermSubst` so that the impracticalities of Janus are avoided. Should a programmer write a clause with more than two occurrences of a `TermSubst` variable, the Gödel compiler would generate code to make a duplicate of the entire array at run-time, thus avoiding aliasing problems. While it may be a little heavy-handed, this approach is adequate for interpreters like SLD Demo and provides a starting point for developing a more discriminatory analysis.

4.6 Future work

By specialising it with SAGE, and using special-purpose low-level support in the form of arrays, we have brought SLD Demo from being 3000 times slower than a Vanilla interpreter with the same object program down to 20 times slower. Is there any prospect of narrowing the remaining gap? Most certainly there is. Even with arrays, our implementation of the WAM-like predicates is rather crude, and could be much improved by efficient low-level implementation. The SICStus `setarg/3` predicate is more general than necessary, and the amount of trailing it performs could be reduced considerably in a special-purpose implementation.

Another possibility for for improving the efficiency of all manipulations of the ground representation is to try to decrease the size of the representation. It was noted in section 2.1 that the ground representation of a given term is very much larger than the term itself. As demonstrated convincingly in O'Keefe [O'K90], the size of the terms chosen to represent a given data set has an important effect on efficiency. Our choice of representation resulted from the requirement to implement the meta-programming facilities specified by Gödel, and to do so without customised low-level support from the implementation. With low-level support, it should be possible to reduce the size of the terms used for the ground representation to little larger than the object terms they represent, and without losing the generality over non-ground noted in section 2.1. Unfortunately, we have not yet been able to experiment with this idea in the existing Gödel system because it cannot be done without a major rewrite and access to the low-level implementation. It is hoped that a future version of Gödel will incorporate a compact representation as a fundamental design decision.

Partial evaluation has a drawback as a general optimisation technique: it is sometimes fragile. For example, choosing the wrong building block, such as `UnifyTerms` rather than `Resolve` as in Unify Demo, not only leads to a slower interpreter, but also one that is not amenable to partial evaluation by SAGE. On the other hand, an advantage of Gödel's module system is that some of the difficulty of effective program specialisation can be ameliorated by providing the programmer with a kit of procedures that specialise nicely. It is likely however that the speed up from specialising meta-programs will become less important as the language implementation improves.

5 Conclusions

If declarative programming is an important idea, then the problem of practical declarative meta-programming must be solved. The non-ground representation is not powerful enough for many important applications, so the central issue is how to make programs that use the ground repre-

sentation run fast enough to be practical. Our initial experience with Gödel interpreters supported the common observation that programs using the ground representation are unacceptably slow. However, we believe that it is possible to improve the performance of these interpreters until they approach the speed of equivalent programs using the non-ground representation.

We have described a progression from the naive *unify-compose-apply* style of interpreter that explicitly mimics a standard formulation of SLD-resolution to a style that is much more easily optimised. Indeed, a significant part of the process of attaining practicality involves the creation of an idiom in which the use of these initially unfamiliar programming techniques will become natural.

A good part of the overhead that Gödel interpreters suffer from is due to extraneous language factors and limitations of the present implementation, rather than the specific use of the ground representation. In particular, evaluating control declarations, and the loss of indexing caused by the use of abstract data types increase the computation time significantly. While these overheads can be removed by partial evaluation, they could equally well be removed by improved Gödel compilation technology. Another overhead that is not entirely inevitable in the ground representation is due to the increased size of represented terms over the terms themselves. This overhead may be avoided by appropriate low-level support for the representation.

The only truly *inherent* overhead suffered by interpreters of the ground representation, compared with interpreters of the non-ground representation, lies in the management of variables. Without careful design, the additional computation from this source can cripple the meta-program. In a sense, writing an efficient ground interpreter from scratch is like building an efficient WAM implementation from scratch, because the programmer is entirely responsible for the management of variables. So interpreters and other meta-programs need to be designed with careful attention to the unification process and the representation of substitutions. However, we do not want to pass this design burden on to the programmer, but would rather relieve it; so Gödel provides, via the system modules `Syntax` and `Programs`, carefully chosen primitives such as `Resolve`, to be the building blocks from which efficient meta-programs can be constructed.

Once the interpreter has been constructed, most of the redundant computation it performs can be removed by specialising it with respect to a particular object program by partial evaluation. This process can change calls to `Resolve` into specialised unification procedures for all the statements of the object program, and so is analogous to compilation. Further speed up can be obtained by optimising the implementation of substitutions, using arrays to store the binding lists. There is a strong suggestion that this is the natural way to obtain efficient management of ground variables; substitutions can be viewed as operating in parallel to the stack in the underlying system. Alternatively, they provide an extra level of indirection between a structure and the values to which its variables are bound. This leads to a programming style in which application and composition of substitutions is discouraged, because these are expensive operations. Instead terms are viewed in the context of specified substitutions.

Taken literally, the performance figures we obtained still seem discouraging. Our simple SLD interpreter is at best an order of magnitude slower than the equivalent Vanilla interpreter. However, we think that the prospect of reducing this gap significantly is an excellent one, because the measurements were made using a crude experimental implementation, and there are important optimisations that remain to be tried, such as reducing the storage requirements of the representation. It must also be emphasised that something is gained in exchange for the extra computation: the ground representation is *declarative*, with all the advantages that declarative programming brings, such as clarity, increased opportunities for parallelism, and ease of processing by other meta-programs. One way to look at the remaining overhead is as an indication that Prolog is not powerful enough to build an efficient implementation of Gödel, and this is only to be expected since Gödel extends the functionality of Prolog considerably.

We claim that the combination of the techniques outlined here, if effectively implemented, can

lead to declarative meta-programs that are easy to write, and come very close in execution speed to programs written using the traditional non-ground (and usually non-logical) Prolog approach. Such programs will have all the added functionality of the ground representation, being fully declarative and able to inspect variables. We *can* have efficient, declarative and powerful meta-programs.

References

[BK82] 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.

[BM87] Jonas Barklund and Håkan Millroth. Integrating complex data structures in Prolog. In *1987 Symposium on Logic Programming*, pages 415–425, 1987.

[Bow92] A.F. Bowers. Representing Gödel object programs in Gödel. Technical Report CSTR-92-31, Department of Computer Science, University of Bristol, 1992.

[Chr94] H. Christiansen. Efficient and complete demo predicates for definite clause languages. Technical Report 51, Computer Science Department, Roskilde University, 1994.

[GL90] A. Guessoum and J.W. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, 1990.

[GL91] A. Guessoum and J.W. Lloyd. Updating knowledge bases II. *New Generation Computing*, 10(1):73–101, 1991.

[Gur94] C.A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[HG94] P.M. Hill and J. Gallagher. Meta-programming in logic programming. Technical Report 94.22, School of Computer Studies, University of Leeds, 1994.

[HL89] 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.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[JGS93] N. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Software Generation.* Prentice Hall, 1993.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[LS91] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3&4):217–242, 1991.

[Mul93] Anne Mulkers. *Live Data Structures in Logic Programs.* Lecture Notes in Computer Science 675. Springer-Verlag, 1993.

[O'K90] Richard A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[SKL90] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In Saumya Debray and Manuel Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference*, pages 421–446, 1990.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[TZ88] 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.

[vH92] F.V. van Harmelen. Definable naming relations in meta-level systems. In A. Petorossi, editor, *Meta-Programming in Logic, Proceedings of the 3rd International Workshop, META-92.* Springer-Verlag, 1992.

[War83] D.H.D. Warren. An abstract PROLOG instruction set. Technical Note 309, SRI International, 1983.