Specialising the Ground Representation in the Logic Programming Language Gödel.

C.A.Gurr*

University of Edinburgh

Abstract

Meta-programs form a class of logic programs of major importance. In the past it has proved very difficult to provide a declarative semantics for meta-programs in languages such as Prolog. These problems have been identified as largely being caused by the fact that Prolog fails to handle the necessary representation requirements adequately. The ground representation is receiving increasing recognition as being necessary to adequately represent meta-programs. However, the expense it incurs has largely precluded its use to date.

The logic programming language Gödel is a declarative successor to Prolog. Gödel provides considerable support for meta-programming, in the form of a ground representation. Using this representation, Gödel meta-programs have the advantage of having a declarative semantics and can be optimised by program specialisation, to execute in a time comparable to equivalent Prolog meta-programs which use a non-ground representation.

Keywords: Partial evaluation, meta-programming, ground representation.

1 Introduction.

The Prolog language, and variants of it, are fraught with problems caused by their non-logical features, and this means that it is not possible to provide a declarative semantics for most, practical, Prolog programs. This applies most strongly to meta-programs in Prolog, where Prolog's declarative problems are compounded by the fact that the non-ground

*email: corin@cogsci.ed.ac.uk

representation is used to represent terms, formulas and programs, despite the fact that, with such a representation, no declarative semantics can be provided for the Prolog features such as `var`, `nonvar`, `assert` and `retract`. The use of a ground representation ([2, 8, 7]) is receiving increasing recognition as being essential for declarative meta-programming, although, until now, the expense that is incurred by the use of such a representation has largely precluded its use.

The logic programming language Gödel [9] has been developed with the intention that it be "a declarative successor to Prolog". Gödel directly addresses the semantic problems of Prolog, providing declarative replacements for the non-logical features of Prolog (such as unsafe negation, Prolog's `setof`, unification without occur-checking, and inadequate facilities for meta-programming).

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

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

Gödel's ground representation is presented to the user via an abstract data type, thus avoiding the need for the user to have knowledge of its implementation, and therefore not confusing the user with a profusion of constant and function symbols. In addition to this, the development of large meta-programming applications such as interpreters, theorem provers, partial evaluators and debuggers, in Gödel, have influenced the development of Gödel's ground representation, so that a natural and clearly readable style of meta-programming with the ground representation is now emerging. This is exemplified by the comparison between the 'naive' Gödel meta-interpreter in figure 4, where unification and resolution are handled explicitly in the code, and the more natural meta-interpreter of figure 3, where resolution is handled implicitly by the Gödel system predicate `Resolve`, discussed in more detail in section 4.1. Henceforth we shall refer to meta-programs which use a ground representation as 'ground' meta-programs and meta-programs which use a non-ground representation as 'non-ground' meta-programs.

Using a ground representation means that unification, particularly the binding of variables (i.e. substitutions), must be handled explicitly by the meta-program. Programmers are unable to rely upon the underlying system to perform unification for them. This can cause considerable execution overheads in meta-programs. However, through program

specialisation the speed of Gödel meta-programs can be optimised so as to remove these overheads, producing specialised versions of unification that may be comparable in execution time to the implicit unification of the underlying system. Certain other specialisations, described below, may also be performed on ground meta-programs. Performing the above specialisations can therefore produce ground Gödel meta-programs that have the potential of executing in a time comparable to equivalent non-ground Prolog meta-programs.

The program specialisation technique that we use is partial evaluation[1], a specialisation technique that has been shown to have great potential, particularly in Functional and Logic Programming. It was first explicitly introduced into Computer Science by Futamura [5] and into Logic Programming by Komorowski [10]. Partial evaluation was put on a firm theoretical basis in [14]. While partial evaluation is capable of removing the majority of the overheads associated with the ground representation, to date attention has focused mainly on the elimination of overheads in non-ground Prolog meta-programs, in Prolog interpreters [6, 12, 16, 17, 18], for example, and, more generally, in [11, 13, 19, 20].

The desire to specialise Gödel meta-programs has prompted the development of a declarative partial evaluator, *SAGE*[2], written in Gödel, that is capable of partially evaluating any program in the Gödel language. Using *SAGE* we have been able to specialise Gödel meta-programs, including *SAGE* itself, to produce residual programs that execute in a significantly reduced time.

The layout of this paper is as follows. In the following section we describe Gödel's meta-programming facilities in more detail. In the third and fourth sections we describe how the ground representation and ground unification, respectively, may be specialised. Finally, we present some results and conclusions, and discuss directions of future research.

2 The Ground Representation in Gödel.

The main facilities provided by the Gödel language are types, modules, control (in the form of control declarations, constraint solving, and a pruning operator), meta-programming and input/output. This means that Gödel, being a rich and expressive language, has a complex syntax. As Gödel's ground representation is intended to be sufficient to represent Gödel programs, as well as arbitrary theories, it must allow for the construction of terms of sufficient complexity to describe arbitrary formulas and Gödel's types, modules, control, meta-programming and input/output facilities. The current implementation of the ground

[1] Also referred to, in this context, as partial deduction.

[2] Self-Applicable Gödel partial Evaluator.

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

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

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

Figure 1: Gödel code for `VarsInTerm`.

representation [3] requires some 75 constants and function symbols to construct the terms necessary to adequately represent the entire Gödel language. If all of these symbols were visible in Gödel meta-programs, it would be necessary for the user to be familiar with the entire representation and competent in the manipulation of all these symbols, before he/she would be competent in the writing of meta-programs. To avoid confronting the user with such complexity unnecessarily, in Gödel, the representations of object level expressions and programs are treated as abstract data types. This also has the added advantage that meta-programs are independent of any specific implementation of the ground representation.

Example Figure 1 gives the Gödel code for finding the variables in an object level term. The predicates `Variable`, `ConstantTerm` and `FunctionTerm` are provided by Gödel. The first argument to such predicates are, respectively, the representations of object level variables, constants, and terms with a function at the top level.

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

There are two major contributory factors to the expense of the ground representation in Gödel. The first is a direct result of supporting the ground representation as an abstract data type. The second, and potentially more serious, factor is that, when using the ground representation, the process of unification must be performed explicitly. However, the expense incurred by both of these factors has been overcome by partially evaluating meta-programs with respect to particular object programs, using the partial evaluator *SAGE*, that is itself written in Gödel. We discuss the above two factors, and their solutions, in more detail in the following two sections.

3 Specialising the Representation of Gödel.

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

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

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

Figure 2: Specialised code for `VarsInTerm1(term,vars,vars1)`.

achieve the efficiency of one that has.

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

We may specialise the Gödel code in figure 1, even without further knowledge of the values of any arguments. The first atom in the body of each statement in the definition of `VarsInTerm1` may be unfolded. The result of this will be to make visible the relevant function symbols in Gödel's ground representation. Figure 2 illustrates the specialised code for `VarsInTerm1`. As the relevant function symbols representing variables, constant and function terms now appear in the first argument of the heads of the statements defining `VarsInTerm1`, the Gödel system may perform first argument indexing to differentiate between the three statements. Consequently, when a call is made to `VarsInTerm1`, with the first argument instantiated, no choicepoints are created, and no backtracking is necessary at any point in the computation. When such specialisations are performed upon an entire meta-program, the resulting gains in efficiency are considerable.

The *SAGE* partial evaluator is capable of performing an automatic specialisation of the code in figure 1. The residual code will leave the definitions of the predicates `VarsInTerm` and `VarsInTerm2` unchanged, and replace the definition of `VarsInTerm1` with the code in figure 2.

```
Solve(program, goal, v, v, subst, subst) <-
  EmptyFormula(goal).
Solve(program, goal, v_in, v_out, subst_in, subst_out) <-
  And(left, right, goal) &
  Solve(program, left, v_in, new_v, subst_in, new_subst) &
  Solve(program, right, new_v, v_out, new_subst, subst_out).
Solve(program, goal, v_in, v_out, subst_in, subst_out) <-
  Atom(goal) &
  StatementMatchAtom(program, module, goal, statement) &
  Resolve(goal, statement, v_in, new_v, subst_in, new_subst, new_goal) &
  Solve(program, new_goal, new_v, v_out, new_subst, subst_out).
```

Figure 3: A Simple Gödel Meta-Interpreter

4 Specialising Resolution in the Ground Representation.

The greatest expense incurred by the use of the ground representation occurs in the manipulation of substitutions. When any variable binding is made, this must be explicitly recorded. Thus any unification, and similarly the composition and application of substitutions, must be performed explicitly. This produces significant overheads in the manipulation of the representations of terms and formulas. In this section we discuss how this expense may be greatly reduced, potentially leading to a specialised form of unification that is comparable to the WAM code [1, 21] for the object program. The need to specialise an explicit unification algorithm for efficiency has also been investigated in [4, 11]. Specialising meta-interpreters for propositional logic to produce WAM-like code has been investigated in [15].

In meta-programming the main manipulations of substitutions occur during resolution or unfolding, where we must unify an atom in some goal with a statement in the object program. Figure 3 gives the main part of a very simple Gödel meta-interpreter for definite programs. It is in the third statement of this program that we see the Gödel predicate `Resolve` being used to resolve an atom in the current goal with respect to a statement selected from the object program. The remaining predicates in Figures 3 and 4 are provided by Gödel, and the following comments are adapted from the definition of Gödel [9]:

```
EmptyFormula(
      formula).      % Representation of the empty formula.

And(
      left ,         % Representation of a formula W.
      right ,        % Representation of a formula V.
      and).          % Representation of the formula  W & V.

IsImpliedBy(
      left ,         % Representation of a formula W.
      right ,        % Representation of a formula V.
      isimpliedby).  % Representation of the formula  W <- V.

StatementMatchAtom(
      program ,      % Representation of a program.
      module ,       % Name of a module in this program.
      atom ,         % Representation of an atom in the language of
                     % this program.
      statement).    % Representation of a statement in this module
                     % whose proposition or predicate in the head is
                     % the same as the proposition or predicate in this
                     % atom.

ApplySubstToFormula(
      formula ,      % Representation of a formula.
      subst ,        % Representation of a term substitution.
      formula1).     % Representation of the formula obtained by
                     % applying this substitution to this formula.

RenameFormulas(
      formulas ,     % List of representations of formulas.
      formulas1 ,    % List of representations of formulas.
      formulas2).    % List of representations of the formulas obtained
                     % by renaming the free variables of the formulas in
                     % the second argument by a specific, unique term
                     % substitution such that they become distinct from
                     % the free variables in the formulas in the first
                     % argument.
```

```
ComposeTermSubsts(
      subst1 ,       % Representation of a term substitution.
      subst2 ,       % Representation of a term substitution.
      subst3).       % Representation of the substitution obtained by
                     % composing these two substitutions (in the order
                     % that they appear as arguments).
```

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

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

Each of these four operations is potentially very expensive when we are dealing with the explicit representation of substitutions, therefore it is vital that `Resolve` be implemented as efficiently as possible.

By contrast to the use of `Resolve`, as in the interpreter of Figure 3, consider the somewhat naive (although still declarative) interpreter of Figure 4. The third statement in the interpreter performs the same task as that of the third statement in the interpreter of Figure 3. However this naive interpreter is arguably more obtuse than that of Figure 3, as the manipulation of formulas and substitutions is here being performed explicitly. There would appear to be two very strong arguments for avoiding this style of meta-programming. The first is that it is more arduous for a programmer, requiring as it does explicit and sophisticated manipulation of formulas and substitutions. The second is not immediately apparent, but it is that the implementation of the interpreter of Figure 4 would be noticeably less efficient than that of Figure 3. Furthermore, the interpreter of Figure 3 may be specialised with respect to an object program in order to remove the majority of the expense of the ground representation, as we shall describe below. With the inherent inefficiencies of the interpreter of Figure 4 however, with its repeated explicit manipulation of the representations of the atom, statement and current substitution, it is far from clear that any specialisation could specialise the resolution process to the same extent.

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

Figure 4: A Naive Gödel Meta-Interpreter

4.1 Specialising Resolve

When we specialise a meta-program such as the interpreter in Figure 3 to a known object program, the statements in the object program will be known. Therefore we may specialise `Resolve` with respect to each statement in the object program. Specialising a call to `Resolve` with respect to a known statement will remove the vast majority of the expense of the ground representation. To see how this is achieved we must look more carefully at the implementation of `Resolve`.

The atom `Resolve(atom,st,v,v1,s,s1,body)` is called to perform the resolution of the atom `atom` with the statement `st`. The integers `v` and `v1` are used to rename the statement with `v` being the integer value used in renaming before the resolution step is performed and `v1` being the corresponding value after the resolution step has been performed. The representations of term substitutions `s` and `s1` represent respectively the answer substitution before and after the resolution step. The last argument, `body`, is the representation of the body of the renamed statement.

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

Figure 5: A Gödel Statement

4.1.1 Variable Renaming

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

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

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

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

For example, in the statement in Figure 5 the variables `x`, `y` and `z` are variables of the first type, variable `u` is of the second type and variable `x1` is of the third type. Thus while variable `x1` will certainly require renaming and variable `u` *may* require renaming, the remaining variables need not be renamed. To see how renaming is achieved we must look more closely at how variables are represented in Gödel.

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

the increment of the current value of `Max` as its third argument and returns as its fourth argument the increment of the value of `Max` after all renaming has been performed. Thus specialising the renaming of the statement in Figure 5 of this statement would create the terms `Var("v",max+1)` and (assuming that the variable `u` also required renaming) `Var("v",max+2)`, where `max` is the current highest variable index.

4.1.2 Applying the Current Substitution

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

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

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

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

Note that in the third case, even though we must test whether the matching term in the atom is a term with a function at the top level, we do not necessarily need to apply the current substitution to the arguments of this term. In the statement in Figure 5 for example, if the fourth argument of an atom we wished to resolve with this statement were bound to some term `F(`$\mathcal{T}_1$`,`$\mathcal{T}_2$`)`, we would not need to apply the current substitution to the term $\mathcal{T}_2$ in order to unify it with the matching variable `u` in the term `F(x,u)`.

4.1.3 Head Unification in `Resolve`

The third operation to be performed in the resolution of an atom with a statement is the unification of the atom and the head of the statement. The unification algorithm employed enforces occur-checking for safeness. Although occur-checking is potentially very

expensive, this expense may be greatly reduced by enforcing occur-checking for repeated variables in the head of the statement only.

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

For example, when unifying an atom with the statement in Figure 5, the bindings for the variables `x`, `y` and `z` in the statement are recorded separately from any potential bindings for variables in the atom. These bindings may then be applied to the body of the statement, replacing the variables `x`, `y` and `z` by the terms to which they have been bound. There will only be one potential occur-check during the unification of an atom with the head of this statement and that will be if the fourth argument of the atom is a term `F(`$\mathcal{T}_1$`,`$\mathcal{T}_2$`)`. In this case the first argument of this function term will be unified with the first argument of the atom and an occur-check will be performed for this unification step alone.

4.1.4 Composition of the Mgu with the Current Substitution

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

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

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

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

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

For example, the term `F(x,u)` appears in the head of a statement in Figure 5. Thus we make a call to `GetFunction` which will succeed with `mode` set to `Write` if the atom's fourth argument is bound to a variable in the current substitution and will succeed with `mode` set to `Read` if the atom's fourth argument is bound in the current substitution to some term `F(`$\mathcal{T}_1$`,`$\mathcal{T}_2$`)`.

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

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

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

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

`UnifyFunction(mode,term,function,mode1,subst,subst1)` in `Write` mode will instantiate `term` to the term `function` and `mode1` is set to `Write`. In `Read` mode this

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

Example Figure 6 illustrates the result of specialising `Resolve` with respect to the statement in Figure 5. In the second argument in the head of this specialised statement, the term `statement` denotes the representation of the statement in Figure 5, which we have omitted for the sake of brevity. The residual calls in the body of the specialised call to `Resolve` unify the atom's fourth argument with a term with a function named `F` at the top level and two arguments. If the atom's fourth argument is bound to a variable in `subst_in` then `mode` is set to `Write` by the call to `GetFunction`, which also binds this variable, in `new_subst`, to a new term with this function at the top level. The subsequent calls to `UnifyValue` and `UnifyVariable` will then instantiate the arguments of this new function term to the atom's first argument, `arg1`, and a new variable, `var`. They will also set `subst_out = new_subst` and `v1 = v+1`. If the atom's fourth argument is bound in `subst_in` to a term with a matching function symbol at the top level then `mode` is set to `Read` and `new_subst = subst_in`. The call to `UnifyValue` then unifies, with occur-checking, the atom's first argument, `arg1`, with the first argument, `sub1`, of this function term. If successful, this unification will return the new substitution `subst_out`. The call to `UnifyVariable` then instantiates `var` to the second argument, `sub2`, of the atom's function term and sets `v1 = v`.

A more complex example of the specialised code for `Resolve` is given in Figure 7. Here, by specialising `Resolve` to the statement `P(x,x,A,F(y,F(x,A))) <- Q(y)` we may see an example of a call to each of the seven predicates described above.

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

Specialising the interpreter in Figure 3 with respect to an object program, we would replace the code for `Resolve` by its specialisation. This would consist of, for each statement in the object program, one statement that performed the resolution of some (unknown)

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

Figure 6: Specialised code for `Resolve`

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

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

Figure 7: More specialised code for `Resolve`

atom with the particular object statement. The residual code in these specialised versions of `Resolve` would be a conjunction of atoms with WAM-like predicates. These predicates are analogous to instructions in the WAM and substitutions may be represented in a format analogous to that of the WAM's heap (global stack). As such, these operations could be implemented by Gödel at a very low level, leading to a computation time for the specialised form of a meta-program, such as that in Figure 3, comparable to that of the object program itself.

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

5 Results.

Example Program	Runtime		Speedup
	Original	Specialised	
Model Elimination (1)	22.56s	029.s	77.79
Model Elimination (2)	26.19s	0.35s	74.83
Demo: Transpose(8x8)	2.94s	0.14s	21.00
Demo: Transpose(8x16)	5.80s	0.23s	25.21
Demo: Fib(10)	11.68s	0.13s	89.85
Demo: Fib(15)	118.34s	1.13s	104.73
Demo: Fib(17)	347.85s	2.84s	122.48
Coroutine: BmSort(7)	2.98s	0.14s	21.29
Coroutine: BmSort(13)	14.08s	0.52s	27.08
Coroutine: EightQueens	5.12s	0.21s	24.38

The above table gives the speedups seen in specialised meta-programs, as a factor of the runtime of the original versus the specialised program. The example meta-programs are

implemented using our efficient meta-programming techniques, as in the interpreter of figure 3, and are specialised with respect to particular object programs/theories to produce specialised ground meta-programs. The example programs are, respectively, a theorem prover, provided by André de Waal, based on the model elimination method and specialised with respect to two theories; an SLDNF interpreter specialised with respect to a program performing matrix transposition and a program to compute Fibonacci numbers; and a coroutining interpreter specialised with respect to a list sorting program that uses the 'British Museum' sorting algorithm and a program that solves the eight queens problem. An analysis of these results shows that a factor of approximately 3 times speedup is obtained by introducing better indexing and that the rest of the speedup is almost entirely due to specialising calls to `Resolve`.

With a lower-level implementation of both the WAM-like predicates mentioned above and the representations of substitutions, these results may be improved yet further, as the expense of emulating these WAM-like instructions in Gödel is removed. This will lead to an execution time for specialised versions of `Resolve` that will be comparable to the WAM code for the object statements themselves. Such improvements will be most noticeable in the specialised code for statements such as those in the matrix transposition program. Thus such an implementation would cause the greatest speedups to the above example of interpreting the matrix transposition program, bringing the results for this example into line with those of the other examples.

All of the specialisations described in this paper are performed automatically by *SAGE*. This means that users, without knowledge of the specific implementation of Gödel's ground representation, may write declarative ground Gödel meta-programs and, without further intervention on the part of the user, such programs can be specialised to produce equivalent programs which will potentially execute in a time comparable to similar Prolog non-ground meta-programs.

6 Conclusions.

The ground representation, provided by Gödel as an abstract data type, leads to clear and easily readable programs. In addition, Gödel's ground representation aids the user by internally handling the majority of any necessary manipulation of substitutions, when using the Gödel predicates `UnifyTerms`, `UnifyAtoms` and `Resolve`. These predicates deal with almost all of the unification and composition and application of substitutions necessary in meta-programming, thus leading to clearer, simpler, meta-programs (e.g. figure 3, as

opposed to figure 4).

Having written a Gödel meta-program, an *automatic* specialisation may be performed by *SAGE*. Thus, without any further involvement on the part of the user, the overheads imposed by using an abstract data type may be removed. At the same time, specialising the meta-program with respect to an object program, a version of the meta-program is produced that, while still declarative, will execute in a significantly improved time. With a suitable implementation of Gödel's representation of substitutions, and the relevant primitive operations upon them, such a meta-program would execute in a time comparable to an equivalent Prolog meta-program which utilised Prolog's (non-declarative) non-ground representation.

We claim that the above results demonstrate that the ground representation is not only an essential tool for declarative meta-programming, but also that it is a practical one, as, through program specialisation, we may remove the expense incurred by its use. Using the ground representation in this way, many of the potential applications of meta-programming that have so far proved impossible in Prolog, such as effective self-applicable partial evaluators, now seem eminently achievable.

7 Acknowledgements.

I would like to thank firstly my supervisor, John Lloyd, and also Tony Bowers, John Gallagher and André de Waal, for advice and stimulating discussions concerning the realisation and implementation of a self-applicable partial evaluator. This work was supported by an SERC studentship award.

References

[1] H Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

[2] K A Bowen and R A Kowalski. Amalgamating language and metalanguage in logic programming. In K L Clark and S-A Tarnlund, editors, *Logic Programming*, pages 153–172, 1982.

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

[4] D.A. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation,* Manchester 1991, pages 205–221. Workshops in Computing, Springer-Verlag, 1992.

[5] Y Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[6] J Gallagher. Transforming logic programs by specialising interpreters. In *ECAI-86*, pages 109–122, Brighton, 1986.

[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] P M Hill and J W Lloyd. Analysis of meta-programs. In H D Abramson and M H Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*. MIT Press, 1989.

[9] P M Hill and J W Lloyd. The Gödel Programming Language. Technical Report CSTR-92-27, Department of Computer Science, University of Bristol, 1992. Revised May 1993. To be published by MIT Press.

[10] H J Komorowski. A specification of an abstract prolog machine and its application to partial evaluation. Technical Report LSST 69, Linkoping University, 1981.

[11] P Kursawe. How to invent a prolog machine. *New Generation Computing*, 5:97–114, 1987.

[12] A Lakhotia and L Sterling. How to control unfolding when specialising interpreters. *New Generation Computing*, 8:61–70, 1990.

[13] G Levi and G Sardu. Partial evaluation of metaprograms in a "multiple worlds" logic. *New Generation Computing*, 6:227–248, 1988.

[14] J W Lloyd and J C Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[15] U Nilsson. Towards a methodology for the design of abstract machines for logic programming languages. *Journal of Logic Programming*, 16(1&2):163–189, May 1993.

[16] S Owen. Issues in the partial evaluation of meta-interpreters. In H D Abramson and M H Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 319–340. MIT Press, 1989.

[17] S Safra and E Shapiro. Meta interpreters for real. In H J Kugler, editor, *Information Processing 86*, pages 271–278. North-Holland, 1986.

[18] L S Sterling and R D Beer. Meta-interpreters for expert system construction. *Journal of Logic Programming*, 6:163–178, 1989.

[19] A Takeuchi and K Furukawa. Partial evaluation of Prolog programs and its application to meta-programming. In H J Kugler, editor, *Information Processing 86*, pages 415–420, Dublin, 1986. North Holland.

[20] R Venken. A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query optimisation. In *ECAI-84: Advances in Artificial Intelligence*, pages 91–100, Pisa, 1984. North-Holland.

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