A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel (Extended Abstract)

C.A.Gurr*
Human Communication Research Centre
University of Edinburgh
2 Buccleuch Place
Edinburgh EH8 9LW
Scotland

February 1994

Abstract

Partial evaluation is a program specialisation technique that has been shown to have great potential in logic programming, particularly for the specialisation of meta-interpreters by the so-called "Futamura Projections". Meta-interpreters and other meta-programs are programs which use another program as data. The Futamura projections also show that partial evaluator which is self-applicable (able to specialise itself) can be used to generate a compiler-generator. This paper describes *SAGE*, a self-applicable partial evaluator for meta-programs in the logic programming language Gödel.

Keywords : Partial evaluation, self-application, logic programming, Gödel.

1 Introduction

Partial evaluation is a program specialisation technique that has been shown to have great potential in logic programming, particularly for the specialisation of meta-interpreters. It was explicitly introduced into Computer Science by Futamura [7] and into logic programming by Komorowski [16], for which it was put on a sound theoretical footing by Lloyd and Shepherdson [18]. In the context of [18] the basic technique for partially evaluating a program P wrt a goal G is to construct "partial" search trees for P with suitably chosen atoms from G as goals, and then extract the specialised program P' from the definitions associated with the leaves of these trees. A recent overview and bibliography for partial evaluation are given by [14] and [21] respectively.

The logic programming community's interest in partial evaluation stems primarily from the first Futamura projection, which illustrates how partial evaluation may be used to compile programs by the specialisation of meta-interpreters. The second Futamura projection shows that if the partial evaluator is self-applicable (able to specialise itself) a compiler may be produced.This is

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

taken one stage further in the third Futamura projection, where a *compiler-generator* is produced by specialising the partial evaluator with respect to itself.

This paper describes the development of a partial evaluator for meta-programs in the logic programming language Gödel [12] and forms an extended abstract for [8]. A key aim of [8] has been the construction of a declarative self-applicable partial evaluator, written in a logic programming language, which was capable of specialising any program in the language in which it was written. To date this result has been achieved in a functional programming language by Neil Jones *et al* [15] and several attempts have been made to construct such a program in a logic programming language [5, 6, 19]. However, these partial evaluators have been constructed in the Prolog language and, due to the non-logical features of Prolog, have only considered restricted subsets of the language and do not generally have a declarative semantics. Specialising full Prolog and the construction of an *effective* (capable of producing efficient results) self-applicable Prolog partial evaluator are tasks that are made most difficult by Prolog's non-logical features. This is illustrated by the sophistication needed to specialise these features.

As a declarative alternative to Prolog, we have chosen to implement our partial evaluator in Gödel. Gödel is a declarative, general-purpose language which provides a number of higher-level programming features, including extensive support for meta-programming with a *ground representation*. The ground representation is a standard tool in mathematical logic in which object level variables are represented by ground terms at the meta-level. The ground representation is receiving increasing recognition as being essential for declarative meta-programming, although the computational expense that it incurs has largely precluded its use in the past.

To achieve the above aim we have extended the basic techniques of partial evaluation to the facilities of Gödel. Particular attention has been given to the specialisation of the inherent overheads of meta-programs which use a ground representation and to the development of the foundations of a methodology for Gödel meta-programs.

We may summarise the three main aims of [8] as being to:

1. develop techniques to allow the specialisation of the full Gödel language
2. develop an implementation and a methodology for meta-programming with the ground representation which was
   - efficient
   - amenable to specialisation
3. design and implement an effectively self-applicable declarative partial evaluator in Gödel.

The layout of this paper is as follows. In the following section we give an overview of the Gödel language and the techniques we have developed to extend partial evaluation as defined in [18] to arbitrary Gödel programs. In the third section we describe in more detail the meta-programming facilities of Gödel, their implementation and specialisation. In the fourth section we give an overview of the partial evaluator *SAGE* (Self-Applicable Gödel partial Evaluator), which implements the techniques described in sections two and three. Finally in section five we present the speedups produced for a range of meta-programs specialised by *SAGE* and describe our preliminary results and conclusions on the self-application of *SAGE*.

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

Plus(Zero,x,x).
Plus(S(x),y,S(z)) <- Plus(x,y,z).

Sunny(x) <-
  Daytime(x) &
  ~(Raining(x) \/ Cloudy(x)).
```

Figure 1: Example Gödel Statements

2 Gödel

Gödel is a declarative, general-purpose logic programming language whose main facilities are modules, types, control (in the form of constraint solving, control declarations and a pruning operator), meta-programming and input/output.

Gödel statements are of the form *Head* ← *Body*, where *Head* is an atom and *Body* is a (possible empty) first-order formula. We use the standard abbreviation of *Head* for statements with an empty body. In Gödel statements, such as those in Figure 1, constant, function and predicate symbol names begin with upper case characters, variable names begin with lower case characters and the symbols `&`, `~`, `\/` and `<-` stand for the logical operators of conjunction, negation, disjunction and implication respectively.

In this section we give an overview of the extensions to the definition of partial evaluation in [18] which permit the specialisation of Gödel programs containing negated formulas, pruning operators and multiple modules.

2.1 Constructive Negation and Conditional Formulas

For negated formulas in Gödel programs we have extended the definition of partial evaluation in [18] to include the concept of constructive negation [3]. Constructive negation may be described as follows. First a partial evaluation of the formula that has been negated is computed. If there are no residual resultants for this partial evaluation then the formula has failed finitely and the negation has therefore succeeded. If at least one of residual resultants has an empty body and has not bound any variables in the original formula then the negation fails safely, otherwise the negations of these resultant bodies and the bindings they compute are conjoined to produce a specialised version of the negation.

For example, let P be the program:

```
P(A).
P(B).
```

where `A`, `B` and `C` are constants. The partial evaluations of the formulas `P(C)`, `P(A)` and `P(x)` would be the sets {}, {`True`} and {`x=A`,`x=B`} respectively, where `True` is the truth-proposition.

The specialisations of the formulas `~P(C)`, `~P(A)` and `~P(x)` would therefore be `True`, `False` and `x~=A & x~=B` respectively, where `False = ~True` and `~=` is the inequality predicate.

Gödel also provides a conditional operator of the form

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

which is defined to mean

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

and is used to avoid the need to compute the formula *Condition* twice. We use constructive negation in the specialisation of this operator in the following manner. First a partial evaluation of *Condition* is computed. If there are no residual resultants for this partial evaluation then *Condition* has failed finitely and the conditional is replaced by *Formula2*, which may subsequently be specialised further. Alternatively the specialised version of *Condition* will be the disjunction of all the residual resultant bodies of this partial evaluation and the bindings they compute. If any of these disjuncts is an empty formula then, in at least one case, *Condition* has terminated successfully and bound no free variables. In this case we may replace the conditional with the conjunction of the specialised *Condition* and *Formula1*, which may subsequently be specialised further. Otherwise the specialisation of *Condition* has not indicated whether *Condition* will succeed or fail and so *Formula1* and *Formula2* are both partially evaluated and a new conditional is constructed in which *Conditional*, *Formula1* and *Formula2* are replaced by their specialised versions.

For example, suppose that the predicate `P` was defined as in the above example, {`R(x)`} was (in all cases) the partial evaluation of the atom `Q(x,y)` and the definition of the predicate `S` was:

```
S(x,y,z) <- IF P(x) THEN Q(y,z) ELSE Q(z,y).
```

then the specialisations of the formulas `S(A,A,B)`, `S(C,A,B)` and `S(x,A,B)` would be `R(A)`, `R(B)` and `IF (x=A \/ x=B) THEN R(A) ELSE R(B)` respectively.

2.2 Pruning

While there is a strong argument in favour of allowing pruning within logic programs, it is well established that the 'cut' operator of Prolog is unsound. Hill *et al* [13] propose an alternative pruning operator for logic programming, the *commit*, which, while naturally affecting completeness, is proved to be sound. This operator is supported by Gödel, although currently in a more restricted form than that presented in [13].

It is also shown in [13] that, provided two conditions are met, the computational equivalence of a program and its partial evaluation wrt a given goal (that is, [18, theorem 4.3]) can be extended to encompass programs containing commits. These conditions are restrictions on the structure of the SLDNF-trees used to obtain the partial evaluation and are referred to as the *freeness* and *regularity* conditions.

A detailed description of the commit operator and proof of its soundness would unfortunately be far too long for a paper of this form. Suffice it to say that it is shown in [8] that, while the freeness condition is acceptable, the regularity condition imposes restrictions on the computation of partial evaluations which are both expensive to enforce and lead to a significant reduction in the amount of specialisation that may be performed. In [8] a technique for partially evaluating

programs containing commit operators is presented which allows us to strengthen the partial evaluation theorem for such programs ([13, theorem 3.2]) by entirely removing the regularity condition.

2.3 Modules and Scripts

The usual software engineering advantages of a module system are well known and apply equally well to Gödel. In its most basic form, a module system simply provides a way of writing large programs so that various pieces of the program do not interfere with one another because of name clashes and also provides a way of hiding implementation details. The Gödel module system is based on these standard ideas. When we partially evaluate a program however, it is often almost impossible to retain more than the barest semblance of the original program's module structure.

When partially evaluating a Gödel program a process we refer to as *flattening* occurs. A symbol declared in any program module may be either promoted or demoted so that it might appear in any other module of the program, potentially violating rules which govern the module structure. In the worst case a partial evaluation will flatten a program to such an extent that the entire module structure of the program is lost. Consequently we construct the results of a partial evaluation as a Gödel *script*. A script is essentially a Gödel program from which all module structure has been removed.

Removing the module structure of a program by constructing its partial evaluation as a script is not as drastic a measure as it may seem, if we assume that the module structure is provided primarily for software engineering purposes. Here the module structure is a useful aid to the programmer when writing and debugging the original program. It seems safe to assume that a program will only be partially evaluated once it is complete and (hopefully) bug-free. In this case the user needs only to be certain that the answers computed by the partially evaluated program are correct with respect to the original program and he/she is unlikely to be concerned with the module structure of the specialised program. In fact, taking the widely accepted view that partial evaluation may be considered as a part of the compilation process for a program, the above argument is perfectly acceptable. All that programmers will generally require from the compilation of their programs is that the compiled version of a program should be correct with respect to the original program.

3 Meta-Programming in Gödel

A meta-program is essentially any program which uses another program (the object program) as data. Clearly many of the major applications of logic programming, such as knowledge base systems, interpreters, compilers, debuggers, program transformers and theorem provers will be meta-programs.

A key issue for meta-programming is the representation of the object programs, formulas and terms. Two main approaches to representation have been identified and these are referred to as the *non-ground* and *ground* representations respectively. The key difference between these two approaches is in the representation of object level variables. In the non-ground representation object level variables are represented by variables at the meta-level, while in the ground representation object level variables are represented by ground terms at the meta-level.

The removal of overheads in meta-programs by program specialisation is a topic that has attracted considerable attention in logic programming. However, to date attention has focused mainly on the elimination of the overheads in non-ground Prolog meta-programs as an example of the so-called "interpretation overhead". While Gödel meta-programs also suffer from these overheads we emphasise that the execution overheads we discuss below are caused specifically by using a ground representation and are additional to the recognised overheads of meta-programming. In this section we discuss how these extra overheads may be almost entirely removed by partial evaluation. These techniques may be applied to supplement techniques used to remove the more familiar interpretation overheads.

The use of a ground representation for meta-programming is a standard tool in mathematical logic which first appeared in logic programming in [2] and the theoretical foundations for meta-programming were laid in [11, 10]. In [11] the differences between the non-ground, referred to in that paper as *typed*, and the ground representations were discussed and it was shown that the ground representation is the more powerful of the two.

Although the ground representation is increasingly being recognised as being essential for declarative meta-programming, the expense that is incurred by the use of such a representation has largely precluded its use in the past. 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, 22] for the object program. This work has previously presented in more detail in [9]. The need to specialise an explicit unification algorithm for efficiency has also been investigated in [4, 17]. Specialising meta-interpreters for propositional logic to produce WAM-like code has been investigated in [20].

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 2 gives the main part of 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(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. `EmptyFormula` is true when its argument is the representation of an empty formula. `And` is true when its third argument is the representation of the conjunction of the formulas in its first two arguments. `StatementMatchAtom` is true when its first argument is the representation of a program, its second argument the name of a module in this program, its third argument the representation of an atom and its fourth argument the representation of a statement in this module whose predicate or proposition in the head matches that of this atom.

```
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 2: A Simple Gödel Meta-Interpreter

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.

When we specialise a meta-program such as the interpreter in Figure 2 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.

For example, Figure 3 illustrates the result of specialising the `Solve` interpreter of Figure 2 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 `Solve_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 `Solve/6` by the new predicate `Solve_1/5`.

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

Specialised interpreter:
    Solve_1(Empty', v, v, subst, subst).
    Solve_1(left &' right, v_in, v_out, subst_in, subst_out) <-
      Solve_1(left, v_in, new_v, subst_in, new_subst) &
      Solve_1(right, new_v, v_out, new_subst, subst_out).
    Solve_1(Append'(arg1, arg2, arg3), v_in, v_out, subst_in, subst_out) <-
      GetConstant(arg1, Nil', subst_in, s1) &
      GetValue(arg2, arg3, s1, new_subst)
      Solve_1(Empty', v_in, v_out, new_subst, subst_out).
    Solve_1(Append'(arg1, arg2, arg3), v_in, v_out, subst_in, subst_out) <-
      GetFunction(arg1, .'(sub11, sub12), mode, subst_in, s1) &
      UnifyVariable(mode, sub11, var, v_in, v1) &
      UnifyVariable(mode, sub12, a1, v1, v2) &
      GetFunction(arg3, .'(sub21, sub22), mode1, s1, s2) &
      UnifyValue(mode1, var, sub21, s2, new_subst) &
      UnifyVariable(mode1, sub22, a2, v2, new_v) &
      Solve_1(Append'(a1, arg2, a2), new_v, v_out, new_subst, subst_out).
```

Figure 3: Specialisation of `Solve` wrt `Append`

When a call to `Resolve` is specialised wrt a known object statement the specialised code is guaranteed to be a single (possible empty) conjunction of atoms. The predicates of these atoms will each be one of `UnifyTerms`, `GetConstant`, `GetFunction`, `UnifyValue`, `UnifyVariable`, `UnifyConstant` or `UnifyFunction`. These predicates perform highly specialised unification operations and together form a crucial part of the implementation of `Resolve`.

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[1]. 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 2, comparable to that of the object program itself.

4 A Strategy for a Self-Applicable Partial Evaluator

The greatest difficulty in achieving an effective, self-applicable partial evaluator is that the more complex that the partial evaluator becomes the harder it is to specialise and thus the less efficient the residual code is likely to be. There are two major reasons for the complexity of most current full-language partial evaluators, these being the ability to specialise the more complex facilities

[1]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.

of the full language and the strategy employed in computing a partial evaluation.

We have overcome the first of these obstacles by implementing *SAGE* in Gödel, which has few non-logical (and therefore hard to specialise) features. To overcome the second we note the comparison in difficulty between specialising the implementation of a strategy which relies mainly on a static analysis of the program before specialisation as opposed to a strategy which relies mainly on a dynamic analysis of goals during specialisation.

Consider the second Futamura projection, where a partial evaluator, PE, will specialise the representation of itself, PE', wrt a meta-interpreter, I'', to produce a compiler Co. Using a functional notation we may represent this as the formula:

$Co = PE(PE', I'')$

A static analysis-based strategy would involve analysis, by PE', of the program to be specialised, I''. Both of these are known at the time of the above specialisation by PE and consequently much of the implementation of this strategy in PE' could be specialised. By contrast, a dynamic analysis-based strategy would involve analysis of goals unfolded during the specialisation of I'' by PE'. These goals are unknown at the time of the above specialisation and thus such a strategy could not be specialised to any great extent. The natural conclusion is therefore that a self-applicable partial evaluator should implement a mainly static analysis-based strategy for specialisation. Previous implementations of self-applicable partial evaluators [6, 15, 19] have also recognised this point.

The primary motivation of the static analysis performed by *SAGE* is for 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.

Having identified certain predicates as being unsafe our strategy is to not unfold atoms with these predicates but rather to produce specialised (recursive) definitions of these predicates.

For example, analysing the partial evaluation of the goal `<-P` wrt the program:

```
P <- Q & R.
Q.
R <- S & R.
S.
```

we detect that the predicate `R` is unsafe. The subsequent partial evaluation of the atom `P` would not unfold `R` when specialising the definition of `P`, but would produce a specialised version of `R`. The specialisation of the above program would therefore be:

```
P <- R.
R <- R.
```

This strategy is somewhat conservative and as such is open to enhancement. However, it has the advantage of being relatively simple to implement and thus easy to specialise and has been demonstrated as being sufficient to specialise a range of Gödel meta-programs.

In [8] termination and correctness proofs are presented for *SAGE* 's static analysis. Together with the partial evaluation theorem ([18, theorem 4.3]) this is used to show the correctness and termination of all partial evaluations computed by *SAGE*

5 Results and Conclusions

5.1 The First Futamura Projection

	Runtime		
Example Program $(I : P)$	$I(P, Q)$	$I_P(Q)$	Speedup
Model Elimination: T_1	22.56s	0.29s	77.79
Model Elimination: T_2	26.19s	0.35s	74.83
SLD: Transpose (8x8)	2.94s	0.14s	21.00
SLD: Transpose (8x16)	5.80s	0.23s	25.21
SLD: Fibonacci (10)	11.68s	0.13s	89.85
SLD: Fibonacci (15)	118.34s	1.13s	104.73
Coroutine: BM-Sort(7)	2.98s	0.14s	21.29
Coroutine: BM-Sort(13)	14.08s	0.52s	27.08
Coroutine: EightQueens	5.12s	0.21s	24.38

The table above gives the speedups seen in specialised meta-programs, as a factor of the runtime of the original versus the specialised program. 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; the `Solve` interpreter of Figure 2 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 gained through better indexing obtained by promoting the symbols hidden by Gödel's ground representation and that the rest of the speedup is almost entirely due to specialising calls to `Resolve`.

For the two meta-interpreters `SLD` and `Coroutine` we have been able to estimate that the interpreted programs above execute at between 100 and 200 times slower than when they are executed at the object level. We have found that a corresponding comparison for the *specialised* interpreters indicates that they will execute at approximately 4-6 times slower for `SLD` and 7-8 times slower for `Coroutine` when compared to the relevant object programs. These specialised programs could be made to run yet faster with a more efficient implementation of the WAM-like predicates mentioned previously.

5.2 Self-Application: Preliminary Results

While the above table demonstrates *SAGE* 's potential, our preliminary results for its self-application by the second and third Futamura projections show speedups of a more modest nature, being approximately 3-4 times speedup for examples of both. This is largely a consequence of the fact that the development of *SAGE* has run concurrently with the development and current implementation of Gödel and in fact has often run ahead of the implementation of

Gödel. The main consequences of this are firstly that not all of the techniques available to *SAGE*, most notably the specialisation of the commit described in section two, are currently supported by Gödel and secondly that *SAGE* 's inability to specialise certain low-level facilities of Gödel has made the speedups appear misleadingly low when we consider the proportion of unfolding steps performed during partial evaluation.

Examination of the programs resulting from the self-application of *SAGE* leads us to believe that were the above two problems addressed, by extending and improving the implementation of Gödel, we would produce results for the self-application of *SAGE* showing speedups of approximately 20 times. We emphasise that it is only improvements to the implementation of Gödel that are needed to achieve this, and not a significant refinement or alteration of *SAGE*. We hope that in the near future an improved implementation Gödel will allow us to demonstrate *SAGE* 's effectiveness for self-applicability.

5.3 Conclusions

In this paper we have given an overview of the partial evaluator *SAGE* and, it is hoped, demonstrated with the above results that in its development we have succeeded in the first two of the aims stated in the introduction and made significant progress in achieving the third.

The experience of developing and testing *SAGE* has also convinced the author of the power Gödel gains by virtue of being a declarative programming language with considerable support for declarative meta-programming. We claim that this work provides strong support in arguing that Gödel provides a highly suitable language in which to:

1. implement program transformation, specialisation and other meta-programming techniques, and

2. write object programs for the testing of such techniques.

Together with tools such as declarative debuggers, *SAGE* and the compilers and compiler-generators it can produce, this makes Gödel an ideal environment for meta-programming.

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] D Chan and M Wallace. A treatment of negation during partial evaluation. In H D Abramson and M H Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 299–318. MIT Press, 1989.

[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] H Fujita and K Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6:91–118, 1988.

[6] D A Fuller and S Abramsky. Mixed computation of prolog programs. *New Generation Computing*, 6:119–142, 1988.

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

[8] C A Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol, 1993. Accepted January 1994.

[9] C A Gurr. Specialising the Ground Representation in the Logic Programming Language Gödel. In *Proceedings of the Third International Workshop on Logic Program Synthesis and Transformation*. Springer-Verlag, July 1993.

[10] P M Hill and J W Lloyd. Meta-programming for dynamic knowledge bases. Technical Report CS-88-18, Department of Computer Science, University of Bristol, 1988.

[11] 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.

[12] 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.

[13] P M Hill, J W Lloyd, and J C Shepherdson. Properties of a pruning operator. *Journal of Logic and Computation*, 1(1):99–143, 1990.

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

[15] N D Jones, P Sestoft, and H Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. *Rewriting Techniques and Applications, Lecture Notes in Computer Science 202*, pages 124–140, 1985.

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

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

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

[19] T Mogenson and A Bondorf. Logimix: A Self-Applicable Partial Evaluator for Prolog. In K-K Lau and T Clement, editors, *LOPSTR 92. Workshops in Computing*, pages 214–227. Springer-Verlag, January 1993.

[20] 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.

[21] P Sestoft and A V Zamulin. Annotated bibliography on partial evaluation and mixed computation. *New Generation Computing*, 6:309–354, 1988.

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