Generating Specialised Update Procedures Through Partial Deduction of the Ground Representation

Michael Leuschel* Bern Martens‡

Department of Computer Science, K.U. Leuven
{michael,bern}@cs.kuleuven.ac.be

Abstract

Integrity constraints are very useful in many contexts, such as, for example, deductive databases, abductive and inductive logic programming. However, fully testing the integrity constraints after each update or modification can be very expensive and methods have been developed which simplify the integrity constraints. In this paper, we pursue the goal of writing this simplification procedure as a meta-program in logic programming and then using partial deduction to obtain specialised update procedures for certain update patterns. We argue that the ground representation has to be used to write this meta-program declaratively. However, contrary to what one might expect, current partial deduction techniques are then unable to specialise this meta-interpreter in an interesting way and no specialised update procedures can be obtained. We present a solution which uses a novel implementation of the ground representation and an improved partial deduction strategy. With this we are able to overcome the difficulties and produce highly specialised and efficient update procedures through partial deduction of meta-interpreters.

1 Introduction

Partial evaluation has received considerable attention both in functional programming and logic programming (e.g. [12, 28]). In the context of pure logic programs, partial evaluation is often referred to as *partial deduction*, the term partial evaluation being reserved for the treatment of non-declarative programs. A convention we will also adhere to in this paper.

Integrity constraints play a crucial role in (among others) deductive databases, abductive and inductive logic programs. They ensure that no contradictory data can be introduced and monitor the coherence of the program or database. From a practical viewpoint however, it can be quite expensive to check the integrity after each update. To alleviate this problem, special purpose integrity simplification methods have been proposed, taking advantage of the fact that the program/database was consistent prior to the update, and only verifying constraints possibly affected by the new information

Some techniques also address pre-compilation aspects and some even explicitly generate *specialised update procedures* for certain update patterns and partial descriptions of the database. The techniques are however restricted to very specific kinds of updates and specific kinds of partial knowledge. Usually the intensional database (i.e. the rules) and the integrity constraints are supposed to be fixed and known, the extensional database (i.e. the

*Supported by Esprit BR-project Compulog II.

‡Senior Research Assistant of the K.U.Leuven Research Council.

facts) is considered to be totally unknown and only updates to the extensional database are considered. This is for instance the case for the approach by Wallace in [31].

A *meta-program* is a program which takes another program, the *object-program*, as input and manipulates it in some way. Some of the applications of meta-programming are (much more detailed accounts can be found in [14, 1]): extending the programming language, debugging, program analysis, program transformation and of course specialised integrity checking. In the latter case, the object program is the (relevant) part of the database or program and the meta-program performs specialised integrity checking.

In the late 80's, the idea was proposed that partial evaluation could be used to derive specialised integrity checks for deductive databases by partially evaluating meta-interpreters. This would allow for a very flexible way of generating specialised update procedures. Any kind of update pattern and any kind of partial knowledge can be considered — it is not fixed beforehand which part of the database is static and which part is subject to change. This can be very useful in practice. For instance in [4], Bry and Manthey argue that it is in general not the case that facts change more often than rules and that rules are updated more often than integrity constraints. Furthermore, by implementing the specialised integrity checking as a meta-interpreter, we are not stuck with one particular method. For example, by adapting the meta-interpreter, we can implement different strategies with respect to testing phantomness and idleness.[1]

In [19], it was shown that partial evaluation has the potential to create highly specialised update procedures for deductive databases. The approach was however limited to hierarchical deductive databases. In this paper, we show how to overcome the substantial difficulties in moving towards recursive databases or programs.

Our exposition is structured as follows. In section 2, we recapitulate the basics of specialised integrity checking and comment on its formulation as a meta-program. In section 3, we uncover a fundamental flaw in an initial approach to the problem. Sections 4 and 5 show how to remedy the difficulties through a novel scheme for unification in the ground representation. We present the results of experiments in section 6, and conjecture that declarative programming might, at the end of the day, also pay off at the level of specialisation potential and (therefore) execution efficiency. Finally, a brief conclusion can be found in section 7.

2 Meta-Programming for Specialised Integrity Checking

2.1 Specialised Integrity Checking

Throughout this paper, we suppose familiarity with basic notions in logic programming ([21]) and partial deduction ([22]). Notational conventions are standard and self-evident. In particular, in programs, we denote variables through strings starting with (or usually just consisting of) an upper-case symbol, while the notations of constants, functions and predicates begin with a lower-case character. Throughout this paper, we consider normal logic programs and goals. Programs include integrity constraints which are clauses of the form $false \leftarrow Body$. (As is well-known, more general programs, goals and constraints can be reduced to this format through the transformations proposed in [24].) For the purposes of this paper, it is convenient to consider integrity violated iff $false$ is derivable via SLDNF.

As pointed out above, integrity constraints play a crucial role in several logic programming based research areas. It is however probably fair to say that they received most attention

[1] See for instance the survey by Bry, Manthey and Martens in [5] or also [16].

in the context of (relational and) deductive databases. Addressed topics are, among others, constraint satisfiability, semantic query optimisation, system supported or even fully automatic recovery after integrity violation and efficient constraint checking upon updates. It is the latter topic that we focus on in this paper.

Two seminal contributions, providing first treatments of the subject in a deductive database setting, are [9] and [23]. In essence, what is proposed is reasoning forwards from an explicit addition or deletion, computing indirectly caused implicit potential updates. Consider the following clause:

$$p(X,Y) \leftarrow q(X), r(Y)$$

The addition of $q(a)$ might cause implicit additions of $p(a,Y)$-like facts. Which instances of $p(a,Y)$ will in fact be derivable depends of course on r. Moreover, some or all such instances might already be provable in some other way. Propagating such potential updates through the program clauses, we might hit upon the possible addition of $false$. Each time this happens, a way in which the update might endanger integrity has been uncovered. It is then necessary to evaluate the (properly instantiated) body of the affected integrity constraint to check whether $false$ is actually provable in this way.

Update propagation along the lines proposed in [23] can be described as follows.

Definition 2.1 (Update)
An *update* is a triple $\langle P^+, P^=, P^- \rangle$ such that $P^+, P^=, P^-$ are normal programs and $P^+ \cap P^= = P^+ \cap P^- = P^= \cap P^- = \emptyset$.

P^- contains the clauses removed and P^+ those added by the update. (Facts are just clauses with an empty body.) Below, $mgu^*(A,B)$ represents an idempotent, most general unifier of the set $\{A, B'\}$ where B' is obtained from B by standardising apart.

Definition 2.2 (Potential Updates)
Given an update $U = \langle P^+, P^=, P^- \rangle$, we define the set of positive potential updates $pos(U)$ and the set of negative potential updates $neg(U)$ inductively as follows:

$$pos^0(U) = \{A \mid A \leftarrow Body \in P^+\}$$
$$neg^0(U) = \{A \mid A \leftarrow Body \in P^-\}$$
$$pos^{i+1}(U) = \{A\theta \mid A \leftarrow \ldots, B, \ldots \in P^=, C \in pos^i(U) \text{ and } mgu^*(B,C) = \theta\}$$
$$\cup \quad \{A\theta \mid A \leftarrow \ldots, \neg B, \ldots \in P^=, C \in neg^i(U) \text{ and } mgu^*(B,C) = \theta\}$$
$$neg^{i+1}(U) = \{A\theta \mid A \leftarrow \ldots, B, \ldots \in P^=, C \in neg^i(U) \text{ and } mgu^*(B,C) = \theta\}$$
$$\cup \quad \{A\theta \mid A \leftarrow \ldots, \neg B, \ldots \in P^=, C \in pos^i(U) \text{ and } mgu^*(B,C) = \theta\}$$
$$pos(U) = \bigcup_{i \geq 0} pos^i(U)$$
$$neg(U) = \bigcup_{i \geq 0} neg^i(U)$$

These sets can be computed through a bottom-up fixpoint iteration.

Simplified integrity checking then essentially boils down to evaluating the corresponding (instantiated through θ) $Body$ every time a $false$ fact gets inserted into some $pos^i(U)$. In practice, these tests can be collected, those that are instances of others removed, and the remaining ones evaluated in a separate constraint checking phase. For further details, formalised slightly differently, we refer to [23]. Here, it suffices to say that the experiments reported in section 6 involve integrity checking along the above sketched lines.

Other proposals often feature more precise (but more laborious) update propagation. [9], for example, computes *actual* rather than potential updates. Many intermediate solutions are possible, all with their own weak as well as strong points. Additional details are not

Object level	Ground representation
X	$var(1)$
c	$struct(c, [])$
$f(X, a)$	$struct(f, [var(1), struct(a, [])])$
$p \leftarrow q$	$struct(clause, [struct(p, []), struct(q, [])])$

Figure 1: A ground representation

of immediate relevance to this paper. To conclude this subsection, we provide some further references for the benefit of the interested reader. An overview of the work during the 80's is offered in [5]. A clear exposition of the main issues in update propagation, emerging after a decade of research on this topic, can be found in [16]. [7] compares the efficiency of some major strategies on a range of examples. Finally, recent contributions can be found in, among others, [6, 10, 17].

2.2 Using a Meta-Interpreter

Update propagation, constraint simplification and verification can be implemented through a meta-interpreter, manipulating updates and programs as object-level expressions. As we already mentioned, a major benefit of such a meta-programming approach lies in the flexibility it offers: Any particular propagation and simplification strategies can be incorporated by the meta-program.

Furthermore, by partial deduction of this meta-interpreter we are (in theory) able to compile specialised update procedures for certain update patterns. Take for instance the following program:

$m(X, Y) \leftarrow p(X, Y), w(Y)$
$false \leftarrow m(a, Z)$

Given the update $P^+ = \{p(a, b) \leftarrow\}$, a meta-interpreter will, after propagating the update, verify the simplified constraint $false \leftarrow m(a, b)$. On the other hand, for the update $P^+ = \{p(b, a) \leftarrow\}$, it will realise that no constraint has to be checked. By specialising the meta-interpreter for an update pattern $P^+ = \{p(\mathcal{A}, \mathcal{B}) \leftarrow\}$, where $\mathcal{A}, \mathcal{B}$ are unknown at specialisation time, we obtain a specialised update procedure, efficiently checking integrity, essentially as follows:

$update\_procedure(add(p(a, X))) \leftarrow evaluate(m(a, X))$

Given the actual values for $\mathcal{A}, \mathcal{B}$, the specialised update procedure will verify the same simplified constraints as the meta-interpreter, but will obtain them much more quickly because the propagation and simplification process is already *pre-compiled*.

Perhaps the most important issue when writing a meta-interpreter is how object-level expressions are to be represented at the meta-level. Two fundamentally different approaches are known. The first one uses the term $p(X, a)$ as the representation of the atom $p(X, a)$. This way of proceeding is called the *non-ground* representation on account of the fact that it denotes object-level variables by meta-level variables. The second approach, on the other hand, uses ground terms for the same purpose and is therefore labelled as the *ground* representation. It translates the same atom into a term like $struct(p, [var(1), struct(a, [])])$. Some examples of the particular ground representation that will be used throughout this paper are presented in figure 1.

The ground representation has the advantage that it can be handled purely declaratively by meta-programs, while treating the non-ground representation most often requires the use of extra-logical built-in's. For example in the non-ground representation we cannot test in a declarative way whether two atoms are variants (or instances) of each other. The only way to test whether $p(X, a)$ and $p(b, Y)$ are variants of each other is by using non-declarative built-in's, like $var/1$ and $= ../2$. In fact it can be quite easily shown that any implementation of the variant or instance test in the non-ground representation must be non-declarative. Suppose we have implemented a predicate $variant/2$ which succeeds if its two arguments represent two atoms which are variants of each other and fails otherwise. Then $\leftarrow variant(p(X), p(a))$ must fail and $\leftarrow variant(p(a), p(a))$ must succeed. This however means that the query $\leftarrow variant(p(X), p(a)), X = a$ fails when using a left to right computation rule and succeeds when using a right to left computation rule. Hence $variant/2$ cannot be declarative (the exact same reasoning holds for the predicate $instance/2$). Thus it is not possible to declaratively write meta-interpreters which require instance or variant checks (as is the case for most bottom-up interpreters). For the ground representation there is no problem whatsoever to write declarative predicates testing whether two ground representations of atoms are variants of each other (or one is an instance of the other).

The non-ground representation (in an untyped context) also has semantical problems due to the fact that object level variables also range over meta-level terms. In general this leads to undesired logical consequences and makes the non-ground representation semantically tricky at best. These problems have been solved for certain classes of programs and queries in [26, 27]. A major advantage of the non-ground representation is that the meta-interpreter can use the underlying unification mechanism while for the ground representation the meta-interpreter has to make use of an explicit unification algorithm. This (currently) induces a difference in speed reaching several orders of magnitude (see for instance [2]). The emerging consensus in the logic programming community is that both representations have their merits and the actual choice depends on the particular application. For a more detailed discussion we refer to [14, 20, 26].

In the particular application at hand, specialised integrity checking, we do not have a choice. If we want to write integrity checking via a declarative meta-interpreter executed under SLDNF, we have to use the ground representation. The method from [23] presented in the previous section, for instance, operates in a bottom-up, breadth-first (i.e. collecting) manner. This cannot be done declaratively in a non-ground or mixed style. The top-down, depth-first approach in [19] does use a mixed style but is limited to hierarchical databases and relies on the non-declarative `verify/1` for efficiency. Also any method for recursive databases requires some sort of instance check to guarantee termination. This can only be done (declaratively) in the ground representation. Details on these and related limitations of the non-ground representation can be found in [14].

3 A Fundamental Limitation of Partial Deduction

In [20] we have shown that partial deduction is incapable of propagating partial input (like for instance $f(Z)$ inside $\leftarrow p(f(Z))$) through the explicit unification algorithm of the ground representation although it is capable to do so for the underlying unification algorithm of the non-ground representation. This comes as a surprise as it is generally believed that using a declarative style of programming makes program analysis and transformation easier.

The following example illustrates this (for further details we have to refer the reader to

[20]). We will use the explicit unification predicate $unify/3$ of [8] which is already tailored towards partial deduction (see also [20]). We also suppose that we have some predicate $apply/3$ at our disposal which applies a substitution to a term (the exact implementation is of no relevance here). To improve readability, we henceforth occasionally use the notation "T" to stand for the ground representation of T.

Example 3.1
Let $G = \leftarrow unify(struct(p,[var(1),X]),\ struct(p,[\text{"}a\text{"},Y]),S),\ apply(var(1),S,Z)$. Then partial deduction is *not* able to deduce that after any successful refutation of G the variable Z will be instantiated to "a".

Unfolding G will yield $G' = \leftarrow unify(X,Y,[var(1)/\text{"}a\text{"}],S), apply(var(1),S,Z)$, where $[var(1)/\text{"}a\text{"}]$ is an incoming substitution representing the mgu of $var(1)$ and "a". In the ground representation, X and Y represent as yet unknown object-level expressions. Therefore, the explicit unification algorithm will consider an infinite number of cases, corresponding to the actual values that X and Y might possibly assume. Doing so, it generates an infinite number of different answers S, always containing $var(1)/\text{"}a\text{"}$ as its last binding. However, no matter how deeply we unfold, there is always one resultant where this binding has not yet been incorporated into S and the information that S *must* contain $var(1)/\text{"}a\text{"}$ is not obtainable by partial deduction. This is surprising as the corresponding example for the non-ground representation poses no problem for partial deduction (see section 4.2).

It can be noted that the approach in [8], combining abstract interpretation with partial deduction, falls short in dealing with such cases. One might also be tempted to infer that the above problem boils down to not being able to represent and access the last element in a (partially unknown) list, and that therefore using some kind of difference lists to denote and manipulate substitutions might offer salvation. However more contrived examples can be easily constructed (see [20]) where the resulting substitutions contain the interesting information "somewhere in the middle".

It follows that partial deduction of meta-interpreters written in the ground representation will never perform any (sophisticated) specialisation at the object-level[2]. Speedups are still obtained by removing part of the interpretation overhead of the ground representation from the meta-program (which can still yield considerable gains, see [13, 2]), but no substantial knowledge about the object-program O will be used.

In our particular case, where we want to obtain specialised update procedures, the lack of information propagation has dramatic effects. A meta-interpreter which implements the specialised integrity checking of section 2.1 for instance, will select an atom C from pos^i and unify this atom with an atom B in the body of a clause $A \leftarrow \ldots, B, \ldots$ and then apply the unifier to the head A to obtain an element of pos^{i+1}. At partial deduction time, the atom C (and maybe also A and B) is not fully known. However, if we want to obtain effective specialisation, it is vital that the information we do possess about C (and B) is propagated "through" unification towards A. If this knowledge is not carried along, we will not be able to fully use the information provided for specialisation. Partial deduction will remove part of the overhead of the ground representation, but no substantial compilation at the level of the object-program/database will occur. In other words, we cannot accomplish our goal of obtaining effective specialised update procedures through partial deduction alone!

[2]First partially deducing the object program separately remedies this problem for the Vanilla interpreter. It is useless and/or incorrect in almost all other cases.

4 Improving Partial Deduction of the Ground Representation

We now present a technique which ensures that *partial input does get propagated through the explicit unification algorithm* when using the ground representation. Our solution is inspired by how the underlying unification provides its results and by how partial deduction is able to propagate partial input for it in the non-ground representation.

4.1 Hiding Substitutions

In all implementations of the ground representation known to the authors, the substitutions generated and manipulated in explicit unification and resolution are explicitly accessible (even in the scheme based on $Resolve/7$ in [13, 2]). As opposed to that, through the *underlying* unification or resolution mechanism, one does *not* gain explicit access to most general unifiers. In fact, when executing $X = Y$ in Prolog or Gödel, we do not obtain a most general unifier — the unifying substitution is directly applied to the environment consisting of the other literals in the goal and of the variables of the top-level query. This observation is of crucial importance for partial deduction. In fact there are usually an infinite number of possibilities for the most general unifiers of two partially known terms. However, when we apply each of these possibilities to some given term, a common structure emerges. It is this common structure that partial deduction *can* obtain for the underlying unification/resolution but *cannot* obtain for an explicit unification/resolution mechanism. Further details can be found in [20].

The core idea proposed and implemented in this paper is then to model the explicit unification and resolution schemes of the ground representation after the underlying unification and resolution, i.e. hide the unifiers and apply them directly to some environment. We will see that through this move we are able to apply the same powerful specialisation technique of the underlying schemes in a straightforward way. More surprisingly, we will obtain a *very efficient* declarative implementation scheme for the ground representation.

We have implemented unification through $env\_unify(T_1, T_2, E, E_\theta)$, a fully declarative predicate where T_1, T_2 are ground representations of the terms (or expressions) to be unified and E is an environment. An *environment* is a list of ground representations of terms. The predicate will calculate the most general unifier of T_1 and T_2 and apply this unifier to the environment yielding the fourth argument E_θ. The environment should contain all the terms for which we want to know how they are affected by the unification of T_1 and T_2. So, substitutions are not represented explicitly but only through their effect on a given set of terms. Unlike in Prolog and Gödel, however, the original environment is still available when using $unify/4$ and the ground representation.

Resolution has been implemented through $env\_unify^*(T_1, T_2, Ein_1, Eout_1, Ein_2, Eout_2)$. Here the arguments T_2 and Ein_2 are always renamed apart with respect to T_1 and Ein_1 before unification, yielding T_2' and Ein_2'. $Eout_1$ is then obtained by applying the unifier of T_1 and T_2' to all terms in Ein_1. Similarly, $Eout_2$ is obtained by applying the unifier of T_1 and T_2' to the renamed apart version Ein_2'.

Using env_unify and env_unify^* requires a different style of programming than for instance the ground representation of Gödel (see [15]). In a sense, it is easier to learn this style of programming as we do not have to worry about substitutions and basically only one predicate, $env\_unify^*/6$, has to be understood to do meta-programming. For instance, it is very easy to

write a vanilla meta-interpreter as the following code shows. If P is the usual append program then this meta-interpreter can be called with $solve(\text{“}P\text{”}, [\text{“}append([1],[2],X)\text{”}], [\text{“}X\text{”}], Res)$, yielding the computed answer $\{Res/[\text{“}[1,2]\text{”}]\}$.

*Vanilla solve/4 with env_unify**
$solve(Prog, [], Env, Env) \leftarrow$ $solve(Prog, [Atom\|Rest], InEnv, OutEnv) \leftarrow$ $\quad member(clause(Head, Body), Prog),$ $\quad env\_unify^*(Atom, Head, [InEnv, Rest], [InEnv1, Rest1], Body, Body1),$ $\quad solve(Prog, Body1, [InEnv1, Rest1], [InEnv2, Rest2]),$ $\quad solve(Prog, Rest2, InEnv2, OutEnv)$

4.2 Treating Unknown Input as Variables

Let us now analyse the fundamental mechanism that allows partial deduction to propagate partial input through the underlying unification and resolution.

First, partial deduction in general does not distinguish between “real” or “static” variables and unknown input. With *static variables*, we mean variables at partial deduction time which remain variables at run time. For example, variables inside clauses of the program to be specialised are static. At partial deduction time, such variables can be represented by $var(.)$ in the ground representation. *Unknown input*, on the other hand, translates to variables at partial deduction time which can be replaced by any term at run time. When using the ground representation, unknown input is represented by (meta-level) variables X at partial deduction time, as for instance in $struct(p, [X])$.

Partial deduction of the underlying resolution treats unknown inputs as if they were variables even though it is not yet known what these unknown inputs will look like at run time and even though the unknown inputs are not guaranteed to be variables at run time. The following lemma, taken from [22] (where it is lemma 4.9), establishes the correctness of this technique.

Lemma 4.1
Let R be the resultant of an SLDNF-derivation D from a normal goal $\leftarrow Q$ and α a substitution. If there is a corresponding derivation D' from $\leftarrow Q\alpha$, then its resolvent R' is an instance of R.

This lemma states that treating unknown inputs in Q as variables produces “sound” results for the resolvent R. For instance if $Q = \leftarrow eq(p(V,X),p(a,Y)),q(V)$ and if D is a derivation which just resolves Q with the clause $eq(V,V) \leftarrow$ then $R = \leftarrow q(a)$ and any resolvent R' at run time will be an instance of R. (In example 3.1 we have seen that this simple knowledge about R cannot be derived by partial deduction of an explicit unification algorithm.) It is this property which captures (part of) the soundness of partial deduction in logic programming.

Now, with underlying unification and resolution, lemma 4.1 is *used only implicitly* by a partial deduction system. It can however also be interpreted as an *explicit* property connecting (abstract) resolution of the (abstract) goal Q with (concrete) resolution of the (concrete) goal $Q\alpha$. In this explicit sense, we will use it to enhance basic partial deduction and thus specialise our env_unify predicate while treating unknown inputs as if they were variables.

Note that although we can infer an interesting property about the resolvent, we can not state anything substantial about what the most general unifiers will look like at run time. For instance, for the example above, some possibilities at run-time are:

instance of **p(V,X)**	instance of **p(a,Y)**	run-time **mgu**	run-time **resolvent**
$p(V, X)$	$p(a, Y)$	$\{V/a, X/Y\}$	$\leftarrow q(a)$
$p(a, b)$	$p(a, Y)$	$\{Y/b\}$	$\leftarrow q(a)$
$p(a, b)$	$p(a, b)$	$\{\}$	$\leftarrow q(a)$
$p(Y, f(V, Z))$	$p(a, f(g(X), b))$	$\{Y/a, V/g(X), Z/b\}$	$\leftarrow q(a)$

This means that treating unknown input as variables is sound with respect to resolvents but *not* with respect to unifiers. As a result, applying the above method is much more difficult (and sometimes even impossible) if a program is able to access and manipulate unifiers (for instance composing them).

Fortunately, our implementation of the ground representation guarantees that unifiers are not manipulated in complex ways and makes it very easy to apply lemma 4.1 at the object level and to ensure propagation of partial input. More details about the particular specialisation scheme can be found in [20]. We are thus able to overcome the limitations of partial deduction obtain specialised update procedures in section 6.

5 Practical Considerations

One can distinguish two main differences between the meta-programming style using the ground representation of Gödel[3] and the one using env_unify: substitutions are not made visible when using env_unify and they are automatically applied to an entire environment. The fact that the substitutions are not visible has the immediate advantage that internally (i.e. to implement env_unify) we can use a very efficient representation for the substitutions. We can go even further and implement the declarative predicate env_unify as a sort of built-in and encode it in the most efficient way possible (for instance using a hashed array representation for substitutions). Developing an efficient representation for the (explicit) substitutions in the Gödel ground representation is a non-trivial task and is still a matter of ongoing research (see [2]).

For the experiments conducted in this paper, we have used a somewhat hybrid approach and implemented the env_unify in terms of two built-in's of Prolog by BIM [29]. To be able to do this, we only had to represent variables as ground terms using the functor $'\$VAR'/1$ instead of $var/1$. The first built-in that was used is $numbervars/3$ which instantiates variables to $'\$VAR'(.)$ ground terms, i.e. it (partially) converts the non-ground to the ground representation. This predicate is non-declarative (but env_unify remains of course declarative and can be treated as such by partial deduction independent of how the predicate is implemented). The second built-in is the new (declarative) built-in $unnumbervars/2$.[4] This built-in converts ground representations of variables to real non-ground variables in a very efficient way. A similar predicate is used in the lifting meta-interpreters using the mixed representation, see [14, 19].

Let us now return to the second difference between the env_unify and the Gödel style, namely that in the env_unify style the unifiers are automatically applied to the entire environment. This means that at each unification an entire environment gets copied and modified. For large environments, this disadvantage can possibly cancel the gain we have obtained by being able to efficiently represent substitutions. However the meta-intepreter of section 4.1

[3] Independent of whether we use something like $UnifyDemo$ or $SLDDemo$, see [2].

[4] Developed for us by Bart Demoen.

can be adapted to avoid the explicit application of the unifier to the entire environment by implementing "local" SLD as defined in [3].[5] First results look promising, but more experiments have to be conducted to analyse the general merit of the new implementation scheme.

Furthermore, for the particular application at hand, specialised integrity checking in the style of [23], the size of the environment will always be very small (it will only contain the head of a clause). As the results in the following section show, this leads to an extremely efficient meta-interpreter using the ground representation.

6 Results

We have implemented the specialised integrity checking method of section 2 for definite[6] recursive databases as a meta-program. This meta-program is fully declarative and uses the new implementation scheme of sections 4 and 5. The code is presented in appendix A.

The partial deduction system we have used for the experiments is based on characteristic trees for the control of polyvariance (see e.g. [11, 18]) and uses a determinate unfolding rule with look-ahead (see e.g. [12]). It is an extension of the approach in [18], is able to handle programs with negation and uses an abstraction operator which preserves characteristic trees. The specialisation technique for env_unify, outlined in section 4.2, has been incorporated into the partial deducer.

The resulting system has been used to partially deduce the meta-interpreter of appendix A for update propagation and simplification of integrity checks, thus compiling specialised update procedures. We report on two experiments in this paper (an extensive study is currently being conducted). The first experiment features a recursive deductive database with 4 rules and 3 integrity constraints (and unknown facts) and an update pattern $P^+ = \{man(X) \leftarrow\}$ where X is a constant unknown at partial deduction time. The second experiment uses the same database with the different update pattern $P^+ = \{parent(X, b) \leftarrow\}$ (resulting in a more involved update propagation).

For comparison's sake, we have also implemented a non-ground and thus non-declarative meta-interpreter performing the same specialised integrity checking (the column labelled "Non-ground" in table 1). The partial evaluator Mixtus (see [30]) for full Prolog has been used to specialise this meta-program (the above partial deducer cannot handle the required non-declarative features) for the already mentionned update patterns (the column labelled "Non-ground (specialised)" in table 1).

The results are summarised in table 1. The timings were obtained by using the $time/2$ predicate of Prolog by BIM running on a Sparc Classic under Solaris. For each test the first line contains the time in seconds needed to generate the specialised integrity checks for 100 updates. The time needed to run the integrity checks was not measured (as this part was not subject to specialisation). The second line contains the relative times wrt the specialised ground representation.

First, note that the unspecialised env_unify ground representation is already very competitive compared to the non-ground one, showing the potential of the new implementation scheme presented in this paper (often the "classical" ground representation runs several orders of magnitude slower than the non-ground one, see [2]). Next, for the non-ground meta-

[5] Thanks for Maurice Bruynooghe for pointing this out to us.

[6] Adding negation to the object program/database does not pose any new problem for generating the specialised integrity checks, e.g. it does not translate to negation at the meta-level.

Test	Ground	Ground (specialised)	Non-ground	Non-ground (specialised)
1	2.06 s	0.02 s	0.42 s	0.38 s
	103	1	21	19
2	6.95 s	1.71 s	1.48 s	1.42 s
	4.06	1	0.87	0.83

Table 1: Results

interpreter, the partial evaluator is forced to preserve the operational behaviour of the non-declarative predicates (in this case `copy/2`, `not(not(.))`, `numbervars/3`, `if-then-else/3`). This probably explains the very disappointing speedups observed. Finally, to the best of our knowledge, test 1 provides the first example where a specialised meta-program using the ground representation has been made to run (spectacularly) faster than a non-specialised meta-program using the non-ground representation. It even vastly outperforms the specialised non-ground representation! In test 2, while exhibiting very satisfactory results, the system does not yet attain the same level of performance. We conjecture that improved automatic unfolding of meta-interpreters, elaborating e.g. the initial effort in [25], will remedy this and render the (declarative) ground representation based approach universally superior.

7 Conclusion

We have started out from the goal of obtaining specialised update procedures for integrity checking through partial deduction of a meta-interpreter. We have argued that (in general) such meta-interpreters can only be written declaratively using the ground representation. Unfortunately, partial deduction is incapable of propagating input through the explicit unification algorithm of the ground representation.

To overcome this substantial problem, we propose a new implementation scheme for the ground representation, hiding substitutions as in implicit, underlying unification and resolution. This allows us to apply the specialisation technique of the non-ground representation to the ground one, thus obtaining specialised update procedures in an elegant and gratifying way. First experiments with this approach look very promising and lead us to believe that the apparent efficiency cost involved in declarative (meta-)programming can be overcome and even transformed into a gain.

Acknowledgements

We thank Bart Demoen for implementing *unnumbervars* (which was used to obtain an efficient env_unify) and for sharing with us his large expertise in running and testing Prolog programs. Danny De Schreye carefully read earlier versions of this paper, provided insightful comments, and made valuable suggestions for its improvement. We are also grateful to him for his continuous support and enthusiastic encouragement. We appreciated stimulating discussions with Tony Bowers, Maurice Bruynooghe and John Gallagher. Finally we thank anonymous referees for their suggestions and comments.

References

[1] J. Barklund. Metaprogramming in logic. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*. Marcell Dekker, Inc., New York. To Appear.

[2] A. F. Bowers and C. A. Gurr. Towards fast and declarative meta-programming. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 137–166. MIT Press, 1995. To Appear.

[3] M. Bruynooghe and D. Boulanger. Abstract interpretation for (constraint) logic programming. Technical Report CW 183, Departement Computerwetenschappen, K.U. Leuven, Belgium, November 1993.

[4] F. Bry and R. Manthey. Tutorial on deductive databases. In *Logic Programming Summer School*, 1990.

[5] F. Bry, R. Manthey, and B. Martens. Integrity verification in knowledge bases. In A. Voronkov, editor, Logic Programming. *Proceedings of the First and Second Russian Conference on Logic Programming*, Lecture Notes in Computer Science 592, pages 114–139. Springer-Verlag, 1991.

[6] M. Celma and H. Decker. Integrity checking in deductive databases — the ultimate method ? In *Proceedings of the 5th Australasian Database Conference*, January 1994.

[7] M. Celma, C. Garcí, L. Mota, and H. Decker. Comparing and synthesizing integrity checking methods for deductive databases. In *Proceedings of the 10th IEEE Conference on Data Engineering*, 1994.

[8] D. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T. Clement and K.-K. Lau, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'91*, pages 205–220, Manchester, UK, 1991.

[9] H. Decker. Integrity enforcement on deductive databases. In L. Kerschberg, editor, *Proceedings of the 1st International Conference on Expert Database Systems*, pages 381–395, Charleston, South Carolina, 1986. The Benjamin/Cummings Publishing Company, Inc.

[10] H. Decker and M. Celma. A slick procedure for integrity checking in deductive databases. In P. Van Hentenryck, editor, *Proceedings of ICLP'94*, pages 456–469. MIT Press, June 1994.

[11] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[12] J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[13] 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.

[14] P. Hill and J. Gallagher. Meta-programming in logic programming. Technical Report 94.22, School of Computer Studies, University of Leeds, 1994. To be published in *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5.* Oxford Science Publications, Oxford University Press.

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

[16] V. Küchenhoff. On the efficient computation of the difference between consecutive database states. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object-Oriented Databases, Second International Conference*, pages 478–502, Munich, Germany, 1991. Springer Verlag.

[17] S. Y. Lee and T. W. Ling. Improving integrity constraint checking for stratified deductive databases. In *Proceedings of DEXA '94*, 1994.

[18] M. Leuschel and D. De Schreye. An almost perfect abstraction operator for partial deduction. Technical Report CW 199, Departement Computerwetenschappen, K.U. Leuven, Belgium, December 1994.

[19] M. Leuschel and D. De Schreye. Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, La Jolla, California, 1995. To Appear.

[20] M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. Technical Report CW210, Departement Computerwetenschappen, K.U.Leuven, Belgium, April 1995.

[21] J. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.

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

[23] J. W. Lloyd, E. A. Sonenberg, and R. W. Topor. Integrity checking in stratified databases. *Journal of Logic Programming*, 4(4):331–343, 1987.

[24] J. W. Lloyd and R. W. Topor. Making PROLOG more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[25] B. Martens. Finite unfolding revisited (part II): Focusing on subterms. Technical Report Compulog II, D 8.2.2.b, Departement Computerwetenschappen, K.U. Leuven, Belgium, 1994.

[26] B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57–82. MIT Press, 1995. To Appear.

[27] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *Journal of Logic Programming*, 22(1):47–99, 1995.

[28] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19 & 20:261–320, May 1994.

[29] *Prolog by BIM 4.0*, October 1993.

[30] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[31] M. Wallace. Compiling integrity checking into update procedures. In J. Mylopoulos and R. Reiter, editors, *Proceedings of IJCAI*, Sydney, Australia, 1991.

A The Meta-Interpreter for Update Propagation

We present a meta-interpreter which performs update propagation and integrity specialisation along the lines of the Lloyd, Topor and Sonenberg method of [23] using the new scheme for the ground representation presented in this paper. The predicate *unnumbervars*/2 will be available in the next release of Prolog by BIM.

```
/* ----------- */
/* lloyd_bup/3 */
/* ----------- */
lloyd_bup(Rules,Update,ICQueries) :-
       bup(Rules,Update,Update,Pos),
       extract_ic(Pos,ICQueries).
extract_ic([],[]).
extract_ic([struct(false,[Arg])|Rest],[Arg|ERest]) :-
       extract_ic(Rest,ERest).
extract_ic([Pos1|Rest],ERest) :-
       not(is_integrity_pos(Pos1)),
       extract_ic(Rest,ERest).
is_integrity_pos(struct(false,[Arg])).
/* ----- */
/* bup/4 */
/* ----- */
bup(Rules,Update,InAllPos,OutAllPos) :-
       bup_step(Rules,Update,[],NewPos,InAllPos,InAllPos1),
       bup2(Rules,NewPos,InAllPos1,OutAllPos).
bup2(Rules,[],AllPos,AllPos).
bup2(Rules,[Atom|T],InAllPos1,OutAllPos) :-
       bup(Rules,[Atom|T],InAllPos1,OutAllPos).
/* ---------- */
/* bup_step/6 */
/* ---------- */
bup_step([],_Pos,NewPos,NewPos,AllPos,AllPos).
bup_step([clause(Head,Body)|RestClauses],Pos,InNewPos,OutNewPos,InAllPos,OutAllPos) :-
       bup_treat_clause(Head,Body,Pos,InNewPos,InNewPos1,InAllPos,InAllPos1),
       bup_step(RestClauses,Pos,InNewPos1,OutNewPos,InAllPos1,OutAllPos).
/* ----------------- */
/* bup_treat_clause/7 */
/* ----------------- */
bup_treat_clause(_Head,[],_Pos,NewPos,NewPos,AllPos,AllPos).
bup_treat_clause(Head,[BodyAtom|R],Pos,InNewPos,OutNewPos,InAllPos,OutAllPos) :-
       bup_treat_body_atom(Head,BodyAtom,Pos,InNewPos,InNewPos1,InAllPos,InAllPos1),
       bup_treat_clause(Head,R,Pos,InNewPos1,OutNewPos,InAllPos1,OutAllPos).
/* -------------------- */
/* bup_treat_body_atom/7 */
/* -------------------- */
/* bup_treat_body_atom(Head,BodyAtom,Pos,InNwPos,OutNwPos,InAllPos,OutAllPos */
```

```
/*  - Pos are the positive atoms added at the last bottom up step */
/*  - InNwPos are the positive atoms added so far in this bottom up step */
/*  - InAllPos are all the positive atoms added so for in all steps */
/*  - BodyAtom is the atom in the body of a clause with head Head */
/* The predicate will test whether BodyAtom unifies with some atom in Pos and */
/* if that is the case test whether the corresponding Head (after applying */
/* the mgu) has to be added to InNwPos,InAllPos. If it has to be added all */
/* subsumed atoms in InNwPos,InAllPos will be removed */
/* The new state is returned in OutNwPos,OutAllPos */
bup_treat_body_atom(Head,BodyAtom,[],NewPos,NewPos,AllPos,AllPos).
bup_treat_body_atom(Head,BodyAtom,[Pos1|Rest],InNewPos,OutNewPos,InAllPos,OutAllpos) :-
       env_unify(BodyAtom,Pos1,Head,UHead,struct([],[]),struct([],[])),
       add_atom(UHead,InAllPos,InAllPos1,Answer),
       bup_treat_body_atom2(Answer,UHead,Rest,InNewPos,InNewPos2,InAllPos1,InAllPos2),
       bup_treat_body_atom(Head,BodyAtom,Rest,InNewPos2,OutNewPos,InAllPos2,OutAllpos).
bup_treat_body_atom(Head,BodyAtom,[Pos1|Rest],InNewPos,OutNewPos,InAllPos,OutAllpos) :-
       not(env_unify_rename_test(BodyAtom,Pos1)),
       bup_treat_body_atom(Head,BodyAtom,Rest,InNewPos,OutNewPos,InAllPos,OutAllpos).
bup_treat_body_atom2(dont_add,UHead,Rest,InNewPos,InNewPos,InAllPos,InAllPos).
bup_treat_body_atom2(add,UHead,Rest,InNewPos,[UHead|InNewPos1],InAllPos,[UHead|InAllPos]) :-
       add_atom(UHead,InNewPos,InNewPos1,add).
        /* will always succeed, just remove covered atoms from InNewPos */
/* ---------- */
/* add_atom/4 */
/* ---------- */
/* add_atom(NewAtom,ListOfAtoms,NewListOfAtoms,Answer) */
/* Decides whether NewAtom has to be added to the ListOfAtoms */
/*   - if NewAtom is an instance of any atom in ListOfAtoms then
       the answer is dont_add and NewListOfAtoms is ListOfAtoms*/
/*   - if NewAtom is not an instance of any atom in ListOfAtoms then
       the answer is add and NewListOfAtoms is ListOfAtoms with
       all instances of NewAtom removed */
/* It is supposed that ListOfAtoms does not contain elements which
       are instances of other elements (except itself) */
add_atom(NewAtom,[],[],add).
add_atom(NewAtom,[Pos1|Rest],[Pos1|Rest],dont_add) :-
       instance_of(NewAtom,Pos1).
add_atom(NewAtom,[Pos1|Rest],OutPos,add) :-
       not(instance_of(NewAtom,Pos1)),
       instance_of(Pos1,NewAtom),
              /* now we already know that the answer is add */
add_atom(NewAtom,Rest,OutPos,add).
add_atom(NewAtom,[Pos1|Rest],[Pos1|OutRest],Answer) :-
       not(instance_of(NewAtom,Pos1)),
       not(instance_of(Pos1,NewAtom)),
       add_atom(NewAtom,Rest,OutRest,Answer).
/* -------------------------- */
/* env_unify/6, instance_of/2 */
/* -------------------------- */
env_unify(Term1,Term2,Env1,EnvAfterSub1,Env2,EnvAfterSub2) :-
       unnumbervars(group(Term1,Env1),group(T12,EnvAfterSub1)),
       unnumbervars(group(Term2,Env2),group(T12,EnvAfterSub2)),
       numbervars(gr(EnvAfterSub1,EnvAfterSub2),1,_VarIndex).
env_unify_rename_test(Term1,Term2) :-
       /* simplified version of env_unify/6, for testing unifiability */
       unnumbervars(Term1,T12),unnumbervars(Term2,T12).
instance_of(X,Y) :- unnumbervars(Y,X).
```