

# Partial Deduction of the Ground Representation and its Application to Integrity Checking

Michael Leuschel                      Bern Martens

Department of Computer Science, K.U. Leuven  
Celestijnenlaan 200 A, B-3001 Heverlee, Belgium  
{michael,bern}@cs.kuleuven.ac.be  
Tel.: +32(0)16-327700 Fax: +32(0)16-327996

## 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 pre-compiled integrity checks for certain update patterns. We argue that the ground representation has to be used to write this meta-program declaratively. We however also show that, contrary to what one might expect, current partial deduction techniques are then unable to specialise this meta-interpreter in an interesting way and no pre-compilation of integrity checks can be obtained. In fact, we show that partial deduction (alone) is not able to perform any (sophisticated) specialisation at the object-level for meta-interpreters written in the ground representation. 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 pre-compiled integrity checks through partial deduction of meta-interpreters.

**Keywords:** Meta-Programming, Ground Representation, Partial Deduction, Integrity Checking, Declarative Programming.

This technical report combines and extends [31] and [32].

## 1 Introduction

*Partial evaluation* has received considerable attention both in functional (e.g. [23]) and logic programming (e.g. [13, 24, 42]). 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 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 [46].

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 (see e.g. [20, 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.

In the late 80's, the idea was proposed that partial deduction 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 pre-compiling the integrity checking. 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 (see e.g. [5] or also [25]).

In [30], it was shown that partial evaluation has the potential to pre-compile efficient integrity checking 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 Integrity Checking

### 2.1 Specialised Integrity Checking

Throughout this paper, we suppose familiarity with basic notions in logic programming ([33]) and partial deduction ([34]). 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. 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 [36].) 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 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 efficient constraint checking upon updates in a deductive database setting, are [10] and [35]. 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 [35] 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:*

$$\begin{aligned} pos^0(U) &= \{A \mid A \leftarrow Body \in P^+\} \\ neg^0(U) &= \{A \mid A \leftarrow Body \in P^-\} \\ pos^{i+1}(U) &= \{A\theta \mid A \leftarrow \dots, B, \dots \in P^=, C \in pos^i(U) \wedge mgu^*(B, C) = \theta\} \\ &\quad \cup \{A\theta \mid A \leftarrow \dots, \neg B, \dots \in P^=, C \in neg^i(U) \wedge mgu^*(B, C) = \theta\} \\ neg^{i+1}(U) &= \{A\theta \mid A \leftarrow \dots, B, \dots \in P^=, C \in neg^i(U) \wedge mgu^*(B, C) = \theta\} \\ &\quad \cup \{A\theta \mid A \leftarrow \dots, \neg B, \dots \in P^=, C \in pos^i(U) \wedge mgu^*(B, C) = \theta\} \\ pos(U) &= \bigcup_{i \geq 0} pos^i(U) \\ neg(U) &= \bigcup_{i \geq 0} neg^i(U) \end{aligned}$$

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

Simplified integrity checking then essentially boils down to evaluating the corresponding (instantiated through  $\theta$ ) *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 [35]. 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. [10], 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 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 [25]. [7] compares the efficiency of some major strategies on a range of examples. Finally, recent contributions can be found in, among others, [6, 11, 26].

## 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 may (in principle) be able to pre-compile the integrity checking for certain update patterns.

Take for instance the following program:

$$\begin{aligned} m(X, Y) &\leftarrow p(X, Y), w(Y) \\ false &\leftarrow m(a, Z) \end{aligned}$$

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:

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

Given the actual values for  $\mathcal{A}, \mathcal{B}$ , this 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*. Both [46] and [45] explicitly address this compilation aspect. Their approaches are however more limited in some important respects and both use ad-hoc techniques and terminology instead of well-established and general apparatus provided by meta-interpreters and partial deduction.

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.

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

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 and non-declarative built-in's, like  $var/1$  and  $=../2$ , have to be used to that end. Indeed for the non-ground representation 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 [40, 41]. 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 e.g. [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 [20, 40] or to appendix C.

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 [35] presented in the previous section, for instance, operates in a bottom-up, breadth-first (i.e. collecting) manner and requires some sort of instance check to guarantee termination for recursive databases. This cannot be done declaratively in a non-ground or a mixed<sup>1</sup> style. The top-down, depth-first approach in [30] does use a mixed style but is limited to hierarchical databases and relies on the non-declarative `verify/1` for efficiency.

## 3 A Limitation of Partial Deduction

### 3.1 Propagation of Partial Input

A partial deducer for logic programming is usually given a program  $P$ , a top-level goal  $G$  and the knowledge that all queries at run time will be instances of  $G$ . For example, a top-level goal  $G$  might be  $\leftarrow p(f(X), Y)$  and the program  $P$  the following one.

**Example 3.1**

- (1)  $p(g(X), f(Y)) \leftarrow q(X)$
- (2)  $p(X, X) \leftarrow q(X)$
- (3)  $q(g(Z)) \leftarrow$

The goal  $G$  provides *partial knowledge* or *input* about the first argument to  $p/2$ . To obtain effective specialisation, the partial deducer not only has to use this partial knowledge during unfolding (for instance to prune clause matches), but should also properly *propagate* this partial knowledge through unfolding steps. Using the partial knowledge boils down to detecting that no instance of  $\leftarrow p(f(X), Y)$  will unify with the head of clause (1) at run time. So, when unfolding  $\leftarrow p(f(X), Y)$ , only clause (2) has to be considered and we will obtain (via the substitution  $\{Y/f(X)\}$ , see figure 3) the goal  $\leftarrow q(f(X))$ . The partial knowledge “ $f(X)$ ” has been properly propagated through the unification  $p(f(X), X) = p(X', X')$  and applied to  $q(X')$ . This partial knowledge can now be used to prune the match with clause (3) and partial deduction is able to detect that the goal  $G$  will always fail. Using something like  $\leftarrow q(Z')$  as the goal obtained after the first unfolding would be wholly unsatisfactory and rob partial deduction of most of its specialisation power. In this case, partial deduction would then not be able to detect that  $G$  will always fail. We will see below that exactly this problem occurs when specialising meta-programs written in the ground representation.

When moving to the partial deduction of meta-programs, extra complications arise because now there are two kinds of partial knowledge and also (possibly) two different unification mechanisms: one at the meta-level and one at the object-level. It is of course also very important to propagate information at the object-level. Otherwise partial evaluation will not be able to perform any (sophisticated) specialisation at the object-level. This is for instance required to perform the first specialiser projection as defined in [16]. In fact, whereas the first Futamura projection specialises a meta-interpreter for a known meta-program but unknown object level parameters, the first specialiser projection specialises a meta-interpreter for a known meta-program and partially known object level parameters. The rest of this paper can also be seen in the light of making the first specialiser projection fully practical for meta-interpreters written in the ground representation. We return to the relevance of performing specialisation at the object level at the end of this section.

---

<sup>1</sup> A declarative style of writing meta-interpreters in which the program is in the ground representation and the goals are lifted to the non-ground one (see [20, 12, 30]).

$solve(empty) \leftarrow$ $solve(X \text{ and } Y) \leftarrow solve(X), solve(Y)$ $solve(X) \leftarrow clause(X, Y), solve(Y)$
--

Figure 2: Non-ground Vanilla *solve*

Let us first examine what happens when we specialise the non-ground representation. Suppose for instance that we use the standard “vanilla” *solve* in figure 2. Let the program  $P$  of example 3.1 be encoded as object-program (through *clause* facts) and let  $G$  be the goal  $\leftarrow solve(p(f(X), Y))$ . As one can verify in figure 3, unfolding this goal results in almost exactly the same picture as unfolding  $\leftarrow p(f(X), Y)$  above. The reason is of course that the same representation and unification mechanism is used at the object-level and the meta-level.<sup>2</sup>

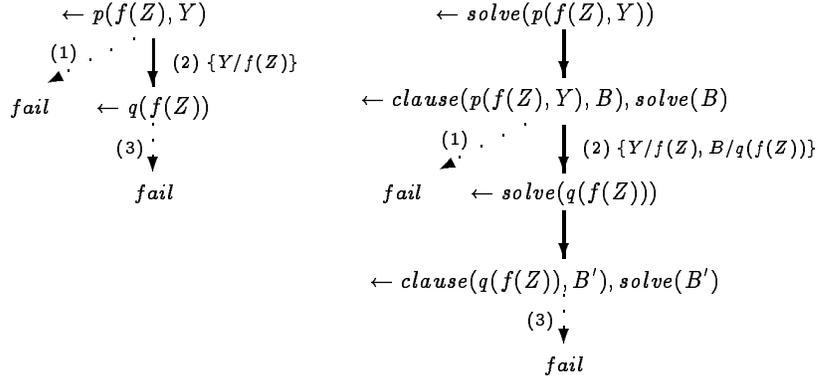


Figure 3: Propagating partial input through unfolding

Next, we turn to the ground representation. To improve readability, we henceforth occasionally use the notation “ $T$ ” to stand for the ground representation of  $T$  and “ $p$ ”(  $t_1 \dots, t_n$ ) to stand for  $struct(p, [t_1, \dots, t_n])$ . Suppose that we have implemented a “vanilla” *solve* using an explicit unification algorithm (e.g. *unify* in [9] or in appendix A). We then try to specialise the goal  $\leftarrow solve(["p"("f"(X, Y)])$  (where, as above,  $X$  and  $Y$  represent input unknown at specialisation time). As in the two examples above, one would like to obtain (after some unfolding) the goal  $\leftarrow solve(["q"("f"(X')])$ . Unfortunately, for the ground representation, this kind of information propagation is in general not obtainable by partial deduction since, as we will explain, this would require infinite unfolding. Consequently, the specialiser will in general come upon the goal  $\leftarrow solve(["q"(X')])$  and potential for specialisation has been lost.

### 3.2 The Culprit: Explicit Substitutions

Before turning to the proper problem, consider a small and simple example:

#### Example 3.2

Let  $P$  be the following program:

```

append([], L, L) ←
append([H|X], Y, [H|Z]) ← append(X, Y, Z)
last([X], X) ←
last([H|T], X) ← last(T, X)

```

<sup>2</sup>The same general picture will also be obtained when specialising a meta-interpreter written in the “mixed” style which lifts the ground representation to the non-ground one for resolution (see [12, 20, 30]).

Let  $G$  be the goal  $\leftarrow \text{append}(L, [a], R), \text{last}(R, X)$ . Thus  $R$  is a list of which we only know that the last element is  $a$  and if  $\text{last}(R, X)$  succeeds then  $X$  must be bound to the constant  $a$ .

Unfortunately, partial deduction is unable to do so because, no matter how deep one unfolds  $G$ , there is always one resultant where  $X$  is not instantiated. Figure 4 gives the general picture (note that the unfolding uses a fair selection rule and that failing branches have been removed for clarity). Indeed, there is an infinite number of possibilities for  $L$  and infinite unfolding is required to deduce that in all (succeeding) cases,  $X$  must be bound to  $a$ .<sup>3</sup>

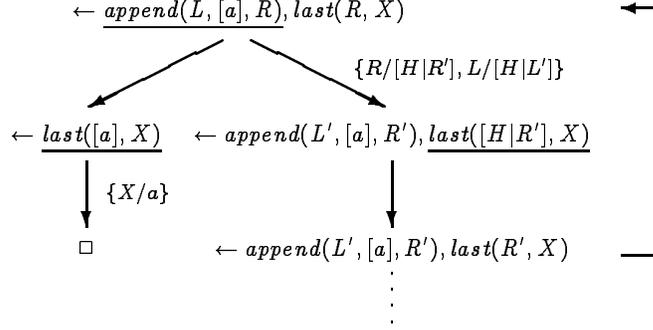


Figure 4: Fair unfolding for example 3.2

We now examine in detail *why* partial deduction is in general unable to propagate information through an explicit unification algorithm and show that the problems are very similar to the ones in example 3.2. We will use the *unify* of appendix A which is already tailored towards partial deduction. We also suppose that we have a predicate *apply* at our disposal which applies a substitution to a term (the exact implementation is of no relevance here).

Let us first inspect an example where the information propagation is obtainable by partial deduction. This does not imply that a given partial deduction system will actually be able to do so (for instance SAGE falls short on the following example; see appendix B).

### Example 3.3

Consider the goal (which might arise when a meta-interpreter resolves the ground representation of the goal  $\leftarrow \text{eq}(X, f(Y)), p(X)$  with the ground representation of the clause  $\text{eq}(X, X) \leftarrow$ ):

$$G = \leftarrow \text{unify}(X, \text{struct}(f, [Y]), S), \text{apply}(X, S, Z)$$

By unfolding to a depth of 5, we obtain two resultants:

$$\begin{aligned} & \text{unify}(\text{var}(N), \text{struct}(f, [Y]), S), \text{apply}(\text{var}(N), S, \text{struct}(f, [Y])) \leftarrow \\ & \quad \text{not}(\text{occur\_args}(\text{var}(N), Y, [])) \\ & \text{unify}(\text{struct}(f, [Y']), \text{struct}(f, [Y]), S), \text{apply}(\text{struct}(f, [Y']), S, \text{struct}(f, [Z'])) \leftarrow \\ & \quad \text{unifyargs}(Y', Y, [], S), \text{applyargs}(Y', S, Z') \end{aligned}$$

In that case partial deduction was able to deduce that, in all cases,  $Z$  must be of the form  $\text{struct}(f, [Z'])$  and the partial input  $\text{struct}(f, [Y])$  has been properly propagated (through non-determinate unfolding).

However, the above is a very simple case and in general, partial deduction is not able to properly propagate partial input.

### Example 3.4

Let  $G = \leftarrow \text{unify}(\text{"p"}(\text{var}(1), X), \text{"p"}(\text{"a"}, Y), S), \text{apply}(\text{var}(1), S, Z)$ . Then partial deduction

<sup>3</sup>Note that a goal like  $G = \leftarrow \text{append}([a], L, R), \text{first}(R, X)$  does not pose any problem.

is *not* able to deduce that after any successful refutation of  $G$  the variable  $Z$  will be instantiated to “ $a$ ”. Unfolding  $G$  will yield  $G' = \leftarrow \text{unify}(X, Y, [\text{var}(1)/\text{“}a\text{”}], S), \text{apply}(\text{var}(1), S, Z)$ , where  $[\text{var}(1)/\text{“}a\text{”}]$  is an incoming substitution representing the *mgu* of  $\text{var}(1)$  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  $\text{var}(1)/\text{“}a\text{”}$  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  $\text{var}(1)/\text{“}a\text{”}$  is not obtainable by partial deduction.

Note that the problem is quite similar to the one in example 3.2. Instead of adding the constant  $a$  to the end of an unknown list, we add the binding  $\text{var}(1)/\text{“}a\text{”}$  to the end of an unknown substitution and partial deduction is not powerful enough to derive the desired information.

One might be tempted to infer that the 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. The following example shows that this is not the case: Its resulting substitutions contain the interesting information “somewhere in the middle”. It also reveals an additional difficulty related to the combination of substitutions.

### Example 3.5

Let  $G = \leftarrow \text{unify}(\text{struct}(p, [X1, \text{var}(1), X2]), \text{struct}(p, [Y1, \text{“}a\text{”}, Y2]), S), \text{apply}(\text{var}(1), S, Z)$ . Again we would like to be able to deduce that, after a successful refutation of  $G$ ,  $Z$  must be instantiated to “ $a$ ”.

The goal  $G' = \leftarrow \text{unify}(X1, Y1, [], S1), \text{unify}(\text{var}(1), \text{“}a\text{”}, S1, S2), \text{unify}(X2, Y2, S2, S), \text{apply}(\text{var}(1), S, Z)$  will be obtained after unfolding. The call  $\text{unify}(\text{var}(1), \text{“}a\text{”}, S1, S2)$  will incorporate the binding  $\text{var}(1)/\text{“}a\text{”}$  into the incoming substitution  $S1$ . Whether the binding has to be actually added to  $S1$  depends on whether  $S1$  already implies the binding or not. This cannot be determined at partial deduction time, even by further unfolding  $\text{unify}(X1, Y1, [], S1)$ , for which there are again an infinite number of different cases. This problem could be overcome by rewriting the *unify* of appendix A such that it always adds the binding, whether it is already implied by the current substitution or not. But this would still leave us with a problem analogous to the one in example 3.4, but more complex since unknown information now both precedes and follows the binding  $\text{var}(1)/\text{“}a\text{”}$  in  $S$ .

Still further intricacies arise when more complex expressions with more variables come into play and also when (as is custom) one of the arguments gets renamed apart before unification. Consequently partial deduction (alone) of meta-interpreters written in the ground representation does not perform any (sophisticated) specialisation at the object-level<sup>4</sup> (as we already pointed out earlier this is required to perform the first specialiser projection as defined in [16]). Speedups are mainly obtained by removing part of the interpretation overhead of the ground representation (which can still yield considerable gains, see [18, 17, 2]). In summary, for certain aspects at least, the analysis and transformation of meta-programs written in the ground representation is much more difficult than for meta-programs written declaratively in the non-ground representation. However, as pointed out in the previous section, a lot of meta-programming tasks *cannot* be written declaratively using the non-ground representation (see also the results in section 6).

In our particular case, where we want to pre-compile integrity checking, 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*<sup>i</sup> and unify this atom with an atom  $B$  in the body of a clause  $A \leftarrow \dots, B, \dots$  and then apply the unifier to the

<sup>4</sup>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.

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, 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 pre-compilation of integrity checks through partial deduction alone!

## 4 Enhancing Partial Deduction

### 4.1 Treating Term Variables as Static Variables

We will now translate example 3.4 to the non-ground representation and examine how partial deduction is able to solve it.

First note that in the goal  $G = \leftarrow \text{unify}(\text{“}p\text{”}(var(1), X), \text{“}p\text{”}(\text{“}a\text{”}, Y), S), \text{apply}(var(1), S, Z)$  of example 3.4 there are two types of “variables” in the representation of object level constructs. First, there is the ground representation  $var(1)$  of a variable. This could be termed a *static* variable because it is guaranteed to remain exactly the same at run-time. Then there are the meta-level variables  $X, Y, Z$ . These variables do not necessarily translate to ground representations of variables at run-time and could be replaced by any ground representation of a term. This is why we call  $X, Y, Z$  *term* variables.<sup>5</sup>

When we move to the non-ground representation, there is no possibility to express this difference between *static* and *term* variables explicitly, both are represented by logical variables. The situation of example 3.4 will arise when we unfold the goal  $Q = \leftarrow \text{eq}(p(V, X), p(a, Y)), q(V, W)$  with the clause  $C = \text{eq}(Z, Z) \leftarrow$ . Partial deduction will treat  $X, Y$  (and  $V, W$ ) just like static variables ignoring that they do not necessarily translate to variables at run-time. So the unifier  $\theta = \{V/a, X/Y, Z/p(a, Y)\}$  of  $p(V, X)$  and  $p(a, Y)$  is calculated and applied to  $q(V, W)$  yielding the leaf  $R = \leftarrow q(a, W)$ . (and in contrast to example 3.4 the partial input  $a$  has been properly propagated).

The correctness of this technique is surprising, but is established by the following lemma (slightly adapted from [34], where it is lemma 4.9).

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

At Partial Deduction Time	At Run Time
$Q$	$Q\alpha$
$\downarrow_D$	$\downarrow_{D'}$
$R$	$R\beta$

The lemma tells us that by treating the *term* variables in  $Q$  just like *static* variables in the unfolding process (even though they can represent any term at run-time via the application of  $\alpha$ ) we obtain a sound approximation  $R$  of the actual leaves at run-time. Now, with underlying (non-ground) 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 resolution of the (abstract) goal  $Q$  with resolution of the (concrete) goal  $Q\alpha$ . Note that although we can infer an interesting property about the leaves, we can not state anything substantial about what the most general unifiers will look like at run time.

<sup>5</sup>Note that there are still further possibilities. For instance a variable  $F$  inside  $\text{struct}(F, [])$  could be called a *functor* variable. We will abstract from these details in this paper.

## 4.2 Hiding Substitutions

As we have already seen in example 3.4, there are usually an infinite number of possibilities for the most general unifiers of two partially known terms. For instance, for the example above, some possibilities at run-time are given in the following table. As one can see, no common structure for the run-time *mgu* can be deduced. However, when applying the *mgu* (directly) on a given term (in this case  $q(V, W)$ ) a common structure re-emerges.

run-time goal $\leftarrow Q\alpha$	run-time mgu	run-time leaf $R'$
$\leftarrow eq(p(V, X), p(a, Y)), q(V, W)$	$\{V/a, X/Y\}$	$\leftarrow q(a, W)$
$\leftarrow eq(p(a, b), p(a, Y)), q(a, W)$	$\{Y/b\}$	$\leftarrow q(a, W)$
$\leftarrow eq(p(a, b), p(a, b)), q(a, W)$	$\{\}$	$\leftarrow q(a, W)$
$\leftarrow eq(p(Z, Z), p(a, Z)), q(Z, W)$	$\{Z/a\}$	$\leftarrow q(a, W)$
$\leftarrow eq(p(Y, f(V, Z)), p(a, f(g(X), b))), q(Y, W)$	$\{Y/a, V/g(X), Z/b\}$	$\leftarrow q(a, W)$

As a result, applying lemma 4.1 to infer extra information about leaves is much more difficult (or even impossible) if a program is able to access and manipulate unifiers (e.g. by composing them).

Fortunately for the *underlying* unification or resolution mechanism, one does *not* gain explicit access to most general unifiers. In fact, when executing  $X = Y$  in Prolog, Gödel (see [21]) or any other logic programming language, 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. As opposed to that, in all implementations of the ground representation known to the authors, the substitutions generated and manipulated in explicit unification and resolution are explicitly accessible. (for instance in [18, 17, 2, 21]). In order to apply lemma 4.1 in a straightforward way, we have to come up with a new scheme of implementation of the ground representation which hides the unifiers and applies them directly to some environment (guaranteeing that unifiers are not manipulated in complex ways).

We have implemented unification via  $env\_unify(T_1, T_2, E_{in}, E_{out})$ , a fully declarative predicate where  $T_1, T_2$  are ground representations of the terms (or expressions) to be unified and  $E_{in}$  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  $E_{out}$ . 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. The  $env\_unify/4$  works in a way similar to the underlying unification in Prolog or Gödel, however leaving the original environment still available. Resolution has been implemented through the predicate  $env\_unify^*(T_1, T_2, E_{in1}, E_{out1}, E_{in2}, E_{out2})$ . Here the arguments  $T_2$  and  $E_{in2}$  are always renamed apart with respect to  $T_1$  and  $E_{in1}$  before unification, yielding  $T'_2$  and  $E'_{in2}$ .  $E_{out1}$  is then obtained by applying the unifier of  $T_1$  and  $T'_2$  to all terms in  $E_{in1}$ . Similarly,  $E_{out2}$  is obtained by applying the unifier of  $T_1$  and  $T'_2$  to the renamed apart version  $E'_{in2}$ .

With this predicate we can program meta-interpreters for the ground representation in a high level style (not unlike *Resolve/7* of [18, 17, 2]). 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  $solve("P", ["append([1], [2], X)"], ["X"], Res)$ , yielding the computed answer  $\{Res/["[1, 2]"]\}$ .

Vanilla <i>solve</i> with $env\_unify^*$
<pre> solve(Prog, [], Env, Env) ← solve(Prog, [Atom Rest], InEnv, OutEnv) ←   member(clause(Head, Body), Prog),   env_unify*(Atom, Head, [InEnv, Rest], [InEnv1, Rest1], Body, Body1),   solve(Prog, Body1, [InEnv1, Rest1], [InEnv2, Rest2]),   solve(Prog, Rest2, InEnv2, OutEnv) </pre>

We will now sketch how lemma 4.1 can be used to specialise the *env\_unify* predicates. For instance, the example 3.4 can be mapped to *env\_unify* in the following way (where we assume the presence of a variable *var(2)* in the environment to illustrate an additional point):

$$\leftarrow \text{env\_unify}(\text{"p"}(\text{var}(1), X), \text{"p"}(\text{"a"}, Y), [\text{var}(1), \text{var}(2)], Z)$$

To propagate partial information from  $T_1, T_2$  over to  $E'$  in the goal  $\leftarrow \text{env\_unify}(T_1, T_2, E, E')$  and thus deal satisfactorily with example 3.4 in practice, we have to take the following steps:

- Transform term variables in  $T_1, T_2, E$  into (fresh) ground representations of variables:  
 $T_1^* = \text{"p"}(\text{var}(1), \text{var}(3)), T_2^* = \text{"p"}(\text{"a"}, \text{var}(4)), E^* = [\text{var}(1), \text{var}(2)].$
- Calculate an explicit unifier  $\theta^*$  of  $T_1^*$  and  $T_2^*$ :  
 $\theta^* = [\text{var}(1)/\text{"a"}, \text{var}(3)/\text{var}(4)].$
- Apply  $\theta^*$  to  $E^*$  yielding  $E^*\theta^* = [\text{"a"}, \text{var}(2)].$
- According to lemma 4.1 (adapted for unification), any  $E'$  at run time will be an instance (at the object-level) of  $E^*\theta^*$ . So, by transforming all representations of object-level variables in  $E^*\theta^*$  into variables at the meta-level, we obtain the term  $\hat{E} = [\text{"a"}, W]$  such that any  $E'$  at run time will be a real instance at the meta-level of  $\hat{E}$ .

We are thus able to deduce that after every refutation,  $Z$  will be bound to an instance of  $[\text{"a"}, W]$ .<sup>6</sup>

Finally one might wonder whether a more general solution may be provided by an abstract interpretation (see [8]) mechanism which propagates partial input. However it seems that obtaining the same level of precision is a non-trivial task. While performing preliminary experiments, we noticed that neither regular approximations ([15]), nor the abstract interpretation method presented in [37], nor set-based analysis ([19]) were able to solve the *unify* examples of this section.<sup>7</sup> Also, the restrictions imposed on all implementations (known to the authors) of the type graphs of [22] make it impossible to derive the desired information. It can further be noted that the approach in [9], combining abstract interpretation with partial deduction, likewise falls short (it only uses abstract interpretation to prune but not to instantiate). So currently there seems to be no alternative to the method sketched in this section. In future work, we plan to draw upon [37] to augment partial deduction so as to solve the problem in a more general manner.

## 5 Practical Considerations

One can distinguish two main differences between the meta-programming style using *env\_unify* and the one using the ground representation of Gödel<sup>8</sup>: 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 we can implement the predicate *env\_unify* as an extremely efficient, but still declarative, built-in. Performing a similar feat for the meta-programming style of Gödel 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 [43]. 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).

<sup>6</sup>Note that the object level variable corresponding to *var(2)* might get further instantiated through the unification of  $X$  and  $Y$ . Returning  $Z = [\text{"a"}, \text{var}(2)]$  as a result would therefore be incorrect.

<sup>7</sup>Also note that set-based analysis was the only method able to solve the simpler *append/last* example 3.2.

<sup>8</sup>Independent of whether we use something like *UnifyDemo* or *SLDDemo*, see [2].

The second built-in is the new (declarative) built-in *unnumbervars/2*.<sup>9</sup> 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 [12, 20, 30].

However, the fact that, in the *env\_unify* style, the unifiers are automatically applied to the entire environment means that at each unification this 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. In the vanilla *solve/4* above, the second environment of *env\_unify\** is the body of a clause which is in general very small (and bounded). However, the first environment will contain all the literals in the current goal that have not been selected so far in the derivation. For highly recursive predicates like the naive reverse this can be a problem. This can be remedied to some extent by implementing “local” SLD as defined in [3].<sup>10</sup> As can be seen in the following piece of code, “local” SLD limits the number of atoms in the environment. The size of the atoms can however still grow in an unbounded way and this approach is not the ultimate solution. Future work will examine the practical merits of the new style of meta-programming (preliminary experiments look promising when compared to the Gödel style, even for the naive reverse program) and also examine whether *env\_unify* itself can be adapted to avoid explicitly applying the unifier to the entire environment.

<pre> Local SLD <i>solve</i> with <i>env_unify*</i> <i>solve</i>(<i>Prog</i>, [], <i>Env</i>, <i>Env</i>) ← <i>solve</i>(<i>Prog</i>, [<i>Atom</i> <i>Rest</i>], <i>InEnv</i>, <i>OutEnv</i>) ←   <i>member</i>(<i>clause</i>(<i>Head</i>, <i>Body</i>), <i>Prog</i>),   <i>env_unify*</i>(<i>Atom</i>, <i>Head</i>, [], [], [<i>Head</i> <i>Body</i>], [<i>Head1</i> <i>Body1</i>]),   <i>solve</i>(<i>Prog</i>, <i>Body1</i>, [<i>Head1</i>], [<i>Head2</i>]),   <i>env_unify*</i>(<i>Atom</i>, <i>Head2</i>, [<i>InEnv</i> <i>Rest</i>], [<i>InEnv1</i> <i>Rest1</i>], [], []),   <i>solve</i>(<i>Prog</i>, <i>Rest1</i>, <i>InEnv1</i>, <i>OutEnv</i>) </pre>
---

However, for the particular application at hand, specialised integrity checking in the style of [35], 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<sup>11</sup> 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 can be found in appendix D.

The partial deduction system we have used for the experiments is based on [28] and uses an abstraction operator which preserves characteristic trees (see also [14, 12, 29]). Automatically unfolding meta-interpreters in a satisfactory way is still an open research problem and therefore some of the results are sub-optimal.<sup>12</sup> A generalisation of the method sketched in section 4.2 has been incorporated<sup>13</sup> into the partial deducer to specialise the *env\_unify* construct. The resulting system has been used to partially deduce the above mentioned meta-interpreter. We report on two experiments in this paper (an extensive study is currently being conducted). The

<sup>9</sup>Developed for us by Bart Demoen.

<sup>10</sup>Thanks for Maurice Bruynooghe for pointing this out to us.

<sup>11</sup>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.

<sup>12</sup>The partial deduction system used for the experiments also does not yet handle the (logical) *if-then-else* which would have come in very handy (especially for the predicates *add\_atom/4* and *bup\_treat\_body\_atom/7* in the code of appendix D) as well for writing the meta-interpreter as for specialising it. Incorporating a technique based on [27] will improve upon this.

<sup>13</sup>Via the calculation of *more specific versions* in a way very similar to the use of “more specific resolution steps” in [12] and based on ideas from [37].

following recursive deductive database with 4 rules and 3 integrity constraints (and unknown facts) has been used for the experiments:

```

father(X, Y) ← parent(X, Y), male(X)
mother(X, Y) ← parent(X, Y), female(X)
anc(X, Y) ← parent(X, Y)
anc(X, Z) ← parent(X, Y), anc(Y, Z)

false ← male(X), female(X)
false ← parent(X, Y), parent(Y, X)
false ← anc(X, adam)

```

The first experiment features 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 also implemented a non-declarative, non-ground meta-interpreter performing the same specialised integrity checking. The partial evaluator Mixtus (see [44]) for full Prolog has been used to specialise this meta-program (the above partial deducer cannot handle the required non-declarative features).

The results are summarised in table 1. In each case, the time needed to generate the specialised integrity checks was measured. The timings were obtained by using the *time/2* predicate of Prolog by BIM on a Sparc Classic under Solaris.

Test	Ground	Specialised Ground	Non-ground	Specialised Non-ground
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

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-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 [38, 39], will remedy this and render the (declarative) ground representation based approach universally superior.

## 7 Conclusion

We have started out from the goal of pre-compiling 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, it turns out that partial deduction is incapable of propagating input through the explicit unification algorithm of the

ground representation. This comes as a surprise as it is generally believed that using the ground representation makes program analysis and transformation easier.

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 pre-compiling integrity checking 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

Michael Leuschel is supported by Esprit BR-project Compulog II. Bern Martens is senior research assistant of the K.U.Leuven research council. We thank Bart Demoen for implementing *unnumbervars* (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. Finally, we appreciated stimulating discussions with Tony Bowers, Maurice Bruynooghe, Dirk Dussart, John Gallagher, Corin Gurr and Anne Mulkers.

## 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] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
- [9] 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. Proceedings of LOP-STR'91*, pages 205–220, Manchester, UK, 1991.

- [10] 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.
- [11] 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.
- [12] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
- [13] 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.
- [14] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.
- [15] J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.
- [16] R. Glück. On the generation of specialisers. *Journal of Functional Programming*, 4(4):499–514, 1994.
- [17] 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.
- [18] C. A. Gurr. Specialising the ground representation in the logic programming language Gödel. In Y. Deville, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR '93*, Workshops in Computing, pages 124–140, Louvain-La-Neuve, Belgium, 1994. Springer-Verlag.
- [19] N. Heintze. Practical aspects of set based analysis. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Washington D.C., 1992. MIT Press.
- [20] 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.
- [21] P. Hill and J. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
- [22] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *The Journal of Logic Programming*, 13(2 & 3):205–258, 1992.
- [23] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
- [24] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta '92*, pages 49–69. Springer-Verlag, LNCS 649, 1992.
- [25] 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.
- [26] S. Y. Lee and T. W. Ling. Improving integrity constraint checking for stratified deductive databases. In *Proceedings of DEXA '94*, 1994.

- [27] M. Leuschel. Partial evaluation of the “real thing”. In L. Fribourg and F. Turini, editors, *Logic Program Synthesis and Transformation — Meta-Programming in Logic. Proceedings of LOPSTR’94 and META’94*, Lecture Notes in Computer Science 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.
- [28] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In *Logic Program Synthesis and Transformation. Pre-Proceedings of LOPSTR’95*, Utrecht, Netherlands, September 1995. To Appear.
- [29] 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.
- [30] 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*, pages 253–263, La Jolla, California, June 1995. ACM Press.
- [31] M. Leuschel and B. Martens. Obtaining specialised update procedures through partial deduction of the ground representation. In H. Decker, U. Geske, T. Kakas, C. Sakama, D. Seipel, and T. Urpi, editors, *Proceedings of the ICLP’95 Joint Workshop on Deductive Databases and Logic Programming and Abduction in Deductive Databases and Knowledge Based Systems*, GMD-Studien Nr. 266, pages 81–95, Kanagawa, Japan, June 1995.
- [32] M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J. Lloyd, editor, *Proceedings of ILPS’95, the International Logic Programming Symposium*, Portland, USA, December 1995. MIT Press. To appear.
- [33] J. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1987.
- [34] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.
- [35] J. W. Lloyd, E. A. Sonenberg, and R. W. Topor. Integrity checking in stratified databases. *Journal of Logic Programming*, 4(4):331–343, 1987.
- [36] J. W. Lloyd and R. W. Topor. Making PROLOG more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.
- [37] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Seattle, 1988. IEEE, MIT Press.
- [38] 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.
- [39] B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, K.U. Leuven, February 1994.
- [40] 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.
- [41] 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.
- [42] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19 & 20:261–320, May 1994.

- [43] *Prolog by BIM 4.0*, October 1993.
- [44] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
- [45] R. Seljée. A new method for integrity constraint checking in deductive databases. *Data & Knowledge Engineering*, 15:63–102, 1995.
- [46] M. Wallace. Compiling integrity checking into update procedures. In J. Mylopoulos and R. Reiter, editors, *Proceedings of IJCAI*, Sydney, Australia, 1991.

## A Ground Unification Algorithm

Below, we include an explicit, ground representation *unify* slightly adapted from [9] (which uses `\ ==` instead of `not(eq(.))`).

Note that unifiers are not calculated in idempotent form, meaning that new bindings do not have to be explicitly composed with the incoming substitution inside the unification algorithm. Partial deduction would be even more complicated if this was the case.

```

unify(X,Y,S) :-
    unify(X,Y,[],S).

unify(var(N),T,S,S1) :-
    bound(var(N),S,B,V),
    unify(var(N),T,S,S1,B,V).
unify(struct(F,Args),var(N),S,S1) :-
    unify(var(N),struct(F,Args),S,S1).
unify(struct(F,Args1),struct(F,Args2),S,S2) :-
    unifyargs(Args1,Args2,S,S2).

unify(var(_),T,S,S1,B,true) :-
    unify(B,T,S,S1).
unify(var(N),T,S,S1,_,false) :-
    unify1(T,var(N),S,S1).

unifyargs([],[],S,S).
unifyargs([T|Ts],[R|Rs],S,S2) :-
    unify(T,R,S,S1),
    unifyargs(Ts,Rs,S1,S2).

unify1(struct(F,Args),var(N),S,[var(N)/struct(F,Args)|S]) :-
    not(occur_args(var(N),Args,S)).
unify1(var(N),var(N),S,S).
unify1(var(M),var(N),S,S1) :-
    diff(M,N),
    bound(var(M),S,B,V),
    unify1(var(M),var(N),S,S1,B,V).
unify1(var(_),var(N),S,S1,B,true) :-
    unify1(B,var(N),S,S1).
unify1(var(M),var(N),S,[var(N)/var(M)|S],_,false).

bound(var(N),[var(N)/T|_],T,true) :-
    diff(T,var(N)).
bound(var(N),[B/_|S],T,F) :-
    diff(B,var(N)),
    bound(var(N),S,T,F).
bound(var(_),[],_,false).

```

```

dereference(var(N), [var(N)/T|_], T) :-
    diff(T, var(N)).
dereference(var(N), [B/_|S], T) :-
    diff(B, var(N)),
    dereference(var(N), S, T).

occur(var(N), var(M), S) :-
    dereference(var(M), S, T),
    occur(var(N), T, S).
occur(var(N), var(N), _) .
occur(var(N), struct(_, Args), S) :-
    occur_args(var(N), Args, S).

occur_args(var(N), [A|_], S) :-
    occur(var(N), A, S).
occur_args(var(N), [_|As], S) :-
    occur_args(var(N), As, S).

diff(X, Y) :-
    not(eq(X, Y)).
eq(X, X) .

```

## B Experiment with SAGE

SAGE is a self-applicable partial deducer for the logic programming language Gödel (see [21]). It is presented in [18, 17, 2] and is heavily optimised towards the treatment of the ground representation. It performs a very good job of removing the overhead of the ground representation for most meta-interpreters. However, with respect to the propagation of partial input at the object level, SAGE falls short.

The following is the program to be specialised to test the propagation of partial input:<sup>14</sup>

```

MODULE      Test.
IMPORT      Syntax, ProgramsIO.

PREDICATE MyUnify: Term * Term * TermSubst.
MyUnify(t1,t2,s) <- EmptyTermSubst(e) & UnifyTerms(t1,t2,e,s).

PREDICATE Test: Name * Term * Term * Term.
Test(funcname,x,y,res) <-
    NewProgram("SimpleProgram",simple) &
    ProgramFunctionName(simple,"SimpleProgram","f",1,funcname) &
    FunctionTerm(fy,funcname,[y]) &
    MyUnify(x,fy,s) &
    ApplySubstToTerm(x,s,res) &
    ConstantTerm(res).           % this will always fail !

```

This program has been specialised with SAGE for the atom `Test(x,y)`. The specialised program is depicted below. The partial deducer was unable to infer any information about `res` and has not detected that the call `ConstantTerm(res)` will always fail. Note that this specialisation is in theory obtainable by partial deduction (see example 3.3). Also note that the `UnifyTerms` call has been duplicated, probably yielding a slowdown instead of a speedup.

---

<sup>14</sup>Note that giving an actual value for the functor `funcname` at partial deduction time is not straightforward because the ground representation of Gödel is hidden inside an abstract data type. This explains the use of the predicates `NewProgram/2` (thanks to Tom Van Den Broeck for finding this predicate for us) and `ProgramFunctionName/5`.

```

Test(v_11,v_1) <-
  UnifyTerms(v_11,Term(Name("SimpleProgram","f",Function,1),[v_1]),
    TermSubst(Heap(0,H(N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N)),[],v_12)
  &
  ApplySubstToTerm(v_11,v_12,CTerm(v_7)).

Test(v_11,v_1) <-
  UnifyTerms(v_11,Term(Name("SimpleProgram","f",Function,1),[v_1]),
    TermSubst(Heap(0,H(N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N)),[],v_12)
  &
  ApplySubstToTerm(v_11,v_12,Int(0)).

```

Experiments with the *Resolve/7* predicate have not yet been conducted due to a problem in the post-processing of SAGE (the post-processing failed without generating a specialised program, the problem will be fixed in the next release).

## C Ground vs Non-Ground Representation

In this appendix, we investigate in some detail the possibilities linked to using a ground or a non-ground representation. To simplify notations we will use “ $p$ ”( $t_1, \dots, t_n$ ) as a shorthand for *struct*( $p, [t_1, \dots, t_n]$ ).

### Unification and Collecting Behaviour

For meta-interpreters using the non-ground representation we can simply use the “underlying”<sup>15</sup> unification. For instance, to unify  $p(X, a)$  and  $p(Y, Y)$  we simply write  $p(X, a) = p(Y, Y)$ <sup>16</sup>. Note that after a call to  $p(X, a) = p(Y, Y)$ , both atoms will have been transformed into  $p(a, a)$  by the underlying system (in more logical terms: the only correct answer for  $p(X, a) = p(Y, Y)$  is  $\{X/a, Y/a\}$ ). This means that the original atom  $p(X, a)$  is no longer “accessible” (for instance in Prolog the only way to “undo” the binding affecting  $p(X, a)$  is via failing and backtracking), i.e. we cannot test “in the same branch” whether the atom  $p(X, a)$  might unify with another atom, say  $p(b, a)$ . This in turn means that it is not possible to write a breadth-first like or a collecting<sup>17</sup> meta-interpreter declaratively using the non-ground representation (it is possible to do this non-declaratively by using for instance the Prolog *copy/2* built-in).

In the ground representation we cannot use the underlying unification as the variables  $X, Y$  are represented as ground terms (for instance “ $p$ ”( $var(1), “a”$ ) = “ $p$ ”( $var(2), var(2)$ ) will fail). The only declarative way out is to use an *explicit unification* algorithm<sup>18</sup>. The top level predicates of one possible such algorithm, taken from [9], are included in appendix A. For instance, *unify*(“ $p$ ”( $var(1), “a”$ ), “ $p$ ”( $var(2), var(2)$ ), *Sub*) would yield an explicit representation of the unifier in *Sub* which can then be applied to other expressions. In contrast to the non-ground representation, the original atoms “ $p$ ”( $var(1), “a”$ ) and “ $p$ ”( $var(2), var(2)$ ) remain accessible in their original form and can thus be used “again” to unify with other atoms. Writing a breadth-first like or a collecting meta-interpreter declaratively poses no problems.

### Renaming Apart

For the non-ground representation we can again use the underlying mechanism for renaming if we store the object-program explicitly in meta-program clauses. For instance, by representing the object program clause  $anc(X, Y) \leftarrow parent(X, Y)$  by *clause*( $anc(X, Y), [parent(X, Y)]$ )  $\leftarrow$

<sup>15</sup> The term “underlying” refers to the system in which the meta-interpreter itself is written.

<sup>16</sup> This is a shorthand for  $eq(p(X, a), p(Y, Y))$  where we define *eq/2* by the single clause  $eq(X, X) \leftarrow$ .

<sup>17</sup> I.e. performing something like *findall/3*.

<sup>18</sup> Note that this is not an option for the non-ground representation because writing an explicit unification algorithm in that context requires non-declarative features, notably *var/1* and *=../2*.

we can resolve an atom  $anc(a, B)$  by calling  $clause(anc(a, B), Body)$  which ensures that clauses are renamed apart before attempting unification with  $anc(a, B)$ <sup>19</sup>.

The disadvantage of this method however is that the object-program is fixed, making it impossible to do “dynamic meta-programming” (see [20]), although this can be remedied by using a mixed meta-interpreter (see [12, 20, 30]).

For the ground representation, it is again easy to declaratively define an explicit standardising apart operator. In the programming language Gödel (see [21]), this predicate is *RenameFormulas/3*.

### Test for variants or instances

In the non-ground representation we cannot test in a declarative way whether two atoms are variants (or an instance) 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* for instance. 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 OLDT or bottom-up meta-interpreters.

Again, 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).

### Summary

The following table summarises the possibilities and limitations of the different styles of writing meta-interpreters.

Behaviour of Meta-interpreter	Non-Ground	Mixed	Ground
Breadth-First/Findall	No	No	Yes
Hypothetical Reasoning	No	Yes	Yes
Loop Checking	No	No	Yes
Underlying Unification	Yes	Yes	No

## D 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 [35] 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).

```

<sup>19</sup>However we cannot generate a renamed apart version of  $anc(a, B)$ . The *copy/2* built-in has to be used for that purpose.

```

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,InNewPos,OutNewPos,InAllPos,OutAllPos) */
/* - Pos are the positive atoms added at the last bottom up step */
/* - InNewPos are the positive atoms added so far in this bottom up step */
/* - InAllPos are all the positive atoms added so far 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 InNewPos,InAllPos. If it has to be added all */
/* subsumed atoms in InNewPos,InAllPos will be removed */
/* The new state is returned in OutNewPos,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).

```