Creating Specialised Integrity Checks
Through Partial Evaluation of Meta-Interpreters

Michael Leuschel[*] and Danny De Schreye[†]

Department of Computer Science, K.U. Leuven

Celestijnenlaan 200A, B-3001 Heverlee, Belgium

e-mail: {michael,dannyd}@cs.kuleuven.ac.be

Tel.:++32(0)16-32 7555 Fax:++32(0)16-32 7996

July 10, 1996

Abstract

Integrity constraints are useful for the specification of deductive databases, as well as for inductive and abductive logic programs. Verifying integrity constraints upon updates is a major efficiency bottleneck and specialised methods have been developed to speedup this task. They can however still incur a considerable overhead. In this paper we propose a solution to this problem by using partial evaluation to pre-compile the integrity checking for certain update patterns. The idea being, that a lot of the integrity checking can already be performed given an update pattern without knowing the actual, concrete update.

In order to achieve the pre-compilation, we write the specialised integrity checking as a meta-interpreter in logic programming. This meta-interpreter incorporates the knowledge that the integrity constraints were not violated prior to a given update and uses a technique to lift the ground representation to the non-ground one for resolution. By partially evaluating this meta-interpreter for certain transaction patterns, using a sophisticated partial evaluation technique presented in earlier work, we are able to automatically obtain very efficient specialised update procedures, executing, for instance, substantially faster than the original meta-interpreter.

1 Introduction

Partial evaluation has received considerable attention both in functional programming (see e.g. [16] or [33] and the references therein) and logic programming (e.g. [27,28,36,37,

73]). However, the concerns in these two approaches have strongly differed. In functional programming, self-application and the realisation of the different Futamura projections, has been the focus of a lot of contributions. In logic programming, self-application has received much less attention.[1] Here, the majority of the work has been concerned with direct optimisation of run-time execution, often targeted at removing the overhead caused by meta-interpreters.

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. Firm theoretical foundations for partial deduction have been established by Lloyd and Shepherdson in [52].

However, pure logic programming is rarely considered to be viable for practical "real-life" programming and for instance Prolog incorporates non-declarative extensions (note however the recent efforts towards a declarative successor of Prolog in e.g. [32] or [76]). In [42] we presented a partial evaluation scheme for a practically usable subset of Prolog encompassing built-ins, like `var/1, nonvar/1` and `=../2`,[2] simple side-effects, like `print/1`, and the operational if-then-else construct. In this paper we apply this partial evaluation scheme to a non-trivial and practically useful meta-interpreter for specialised integrity checking in deductive databases.

From a theoretical viewpoint, *integrity constraints* are very useful for specifying deductive databases as well as inductive or abductive logic programs.[3] They ensure that no contradictory data can be introduced and monitor the coherence of a database. From a practical viewpoint however it can be quite expensive to check the integrity of a deductive database after each update. An extensive amount of research addresses the task of improving integrity checking such that it takes advantage of the fact that a database was consistent prior to any particular update and only verifies the relevant parts of a database. Some references to this line of work during the 80's are [9,17,19,53,55,71], with an overview offered in [11]. A clear exposition of the main issues in update propagation, can be found in [38], while [14] compares the efficiency of some major strategies on a range of examples. Finally, recent contributions can be found in, among others, [13,20,40].

Quite often, parts of the database remain static for longer periods of time and the updates are instances of certain given patterns. Therefore some techniques also address precompilation of the integrity constraint specialisation. For instance, the method by Wallace in [80] explicitly generates *specialised update procedures* for certain update patterns. The techniques are however rather ad-hoc and restricted to very specific kinds of updates and to a particular simplification technique. 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 [80].

A *meta-program* is a program which takes another program, the *object-program*, as

---

[1] Some notable exceptions are [29,30,34,62].

[2] But excluding built-ins which manipulate clauses and goals, like `call/1` or `assert/1`.

[3] In the remainder of this paper we will only talk about deductive databases, but the discussions and results remain of course also valid for inductive or abductive logic programs with integrity constraints.

input and manipulates it in some way. A detailed account of meta-programming and its uses can be found in [31]. Some of the applications of meta-programming are: extending the programming language, multi-agent reasoning, debugging, program analysis, program transformation and of course specialised integrity checking. In the latter case the object program is the (relevant) part of a deductive database and the meta program performs specialised integrity checking.

In the late 80's it 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 [10], Bry and Manthey argue that it is not always 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 depending on the application at hand.

However, to the best of our knowledge, the idea based on partially evaluating a meta-interpreter, was never actually implemented and in the second part of this paper we provide the first practical realisation. We will apply the partial evaluation scheme developed in [42] to a particular meta-program performing integrity checking in deductive databases. Our results illustrate the viability of the above approach and indicate that partial evaluation has the potential to create highly specialised update procedures for deductive databases.

The paper is structured as follows. In Section 2 we introduce some basic definitions relative to deductive databases and present a method which performs specialised integrity checking. In Sections 3 and 4 we present a meta-interpreter which implements this method for hierarchical, normal databases and present some insights into object level representation issues. In Section 5 we present a sophisticated partial evaluation technique based on [42], which we then use in Section 6 to generate specialised update procedures and demonstrate their efficiency by an extensive set of experiments. In Section 7 we discuss the steps required to handle recursive databases. Some concluding remarks can be found in Section 8.

An earlier version of this article appeared in [46].

2 Deductive Databases and Integrity Checking

We assume the reader to be familiar with the standard notions of logic programming, like *term*, *atom* or *literal*. Introductions to logic programming can be found in [1] and [51]. We use the convention to represent logical variables by (specially typeset) uppercase letters like x, y. Predicates and functors will be represented by lowercase letters like p, q, f, g.

In this section we present some essential background in deductive databases and integrity checking. We also present a new method for specialised integrity checking, which we will later on implement as a meta-interpreter.

3

Definition 2.1 (Deductive Database)
A clause *is a first-order formula of the form Head ← Body where Head is an atom and Body is a conjunction of literals. A* deductive database *is a set of clauses.*

We do not make any distinction between facts, rules and integrity constraints. A fact is just a clause with an empty body. An integrity constraint is a clause of the form *false ← Body*. As is well-known, more general rules and constraints can be reduced to this format through the transformations proposed in [54]. See also the discussions in [74] about inconsistency indicators.

For the purposes of this paper, it is convenient to consider a database to be *inconsistent*, or violating the integrity constraints, iff *false* is derivable in the database via SLDNF (see e.g. [1] or [51]). Other views of inconsistency exist and some discussions can for instance be found in [11]. Note that we do not require a deductive database to be range-restricted. Range-restriction is not necessary for the approach in this paper. This is because our notion of integrity is based on the SLDNF procedure, which always gives the same answer irrespective of the underlying language (see e.g. [75]). However range-restriction is still useful as it ensures that no SLDNF-refutation will flounder.

Finally we allow SLDNF-derivations to be *incomplete*, i.e. neither leading to success nor failure, but to a goal where no literal has been selected for a further derivation step.

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 [19] and [53]. 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.

We now present a particular method of update propagation in more detail.

Definition 2.2 (Database Update)
A database update, *U, is a triple* $\langle Db^+, Db^=, Db^- \rangle$ *such that* $Db^+, Db^=, Db^-$ *are deductive databases and* $Db^+ \cap Db^= = Db^+ \cap Db^- = Db^= \cap Db^- = \emptyset$.

We say that δ is a SLDNF *derivation after* U *iff δ is an SLDNF derivation for $Db^+ \cup Db^=$.*
Similarly δ is a SLDNF *derivation before* U *if δ is an SLDNF derivation for $Db^- \cup Db^=$.*

Note, that we take the liberty to not always explicitly cite the goal for which an SLDNF-derivation is made. Intuitively, $Db^- \cup Db^=$ represents the database state before the update and $Db^+ \cup Db^=$ represents the database state after the update. In other words Db^- are the clauses removed by the update and Db^+ are the clauses which are added by the update.

We now present a method to characterise the (potential) effect a database update has on the set of deducible atoms. It is loosely based on the calculation of the sets of atoms $pos_{D,D'}, neg_{D,D'}$ by Lloyd, Sonenberg and Topor in [53], which in turn is an extension of the calculation of $atom_{D,D'}$ by Lloyd and Topor in [55]. The main difference being that we calculate $pos(U)$ and $neg(U)$ in one step instead of in two. This approach should be more efficient, while yielding the same result, because in each iteration step the influence of an atom C is independent of the other atoms currently in pos^i and neg^i.

Also, from now on, $mgu^*(A, B)$ represents an idempotent, most general unifier of the set $\{A, B'\}$, where B' is obtained from B by standardising apart. This small technical point was overlooked in [53, 55], meaning that the results therein are incorrect and one should use the mgu^* instead of the plain mgu. A small discussion of this point is provided below in the proof of Lemma 2.7.

Definition 2.3 (Potential Updates)
Given a database update $U = \langle Db^+, Db^=, Db^- \rangle$, we define the set of positive potential updates $pos(U)$ and the set of negative potential updates $neg(U)$ inductively as follows:

$$pos^0(U) = \{A \mid A \leftarrow Body \in Db^+\}$$
$$neg^0(U) = \{A \mid A \leftarrow Body \in Db^-\}$$

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

$$pos(U) = \bigcup_{i \geq 0} pos^i(U)$$
$$neg(U) = \bigcup_{i \geq 0} neg^i(U)$$

Note that the above definition does not test whether an atom $A \in pos(U)$ is a "real" update, i.e. whether A is actually derivable after the update (this is what is called the *phantomness* test) and whether A was indeed not derivable before the update (this is

called the *idleness* test). A similar remark can be made about the atoms in $neg(U)$. This has the advantage that we do not need to access the entire database, and in fact the above definition does not reference the set of facts in $Db^=$ at all, because only clauses with at least one literal in the body are used. This is advantageous in a lot of cases, but also somewhat restricts the usefulness of this method (and the one in [53, 55]) when rules and integrity constraints change more often than facts.

Example 2.4 *Let* $Db^+ = \{man(a) \leftarrow\}$, $Db^- = \emptyset$ *and let the following clauses represent the rules of* $Db^=$:

$$
\begin{array}{l}
mother(\text{X},\text{Y}) \leftarrow parent(\text{X},\text{Y}), woman(\text{X}) \\
father(\text{X},\text{Y}) \leftarrow parent(\text{X},\text{Y}), man(\text{X}) \\
false \leftarrow man(\text{X}), woman(\text{X}) \\
false \leftarrow parent(\text{X},\text{Y}), parent(\text{Y},\text{X})
\end{array}
$$

With $U = \langle Db^+, Db^=, Db^- \rangle$, *we then obtain that* $pos(U) = \{man(a), father(a,\_), false\}$ *and* $neg(U) = \emptyset$.

The following definition uses the sets $pos(U)$ and $neg(U)$ to obtain more specific instances of goals and detect whether the proof tree of a goal is potentially affected by an update.

Definition 2.5 (Θ_U^+, Θ_U^-)
Given a database update U *and a goal* $G = \leftarrow L_1, \ldots, L_n$, *we define:*

$$
\begin{aligned}
\Theta_U^+(G) = \{\theta \mid\ & C \in pos(U),\ mgu^*(L_i, C) = \theta, \\
& L_i \text{ is a positive literal and } 1 \le i \le n \ \} \\
\cup\ \ \{\theta \mid\ & C \in neg(U),\ mgu^*(A_i, C) = \theta, \\
& L_i = \neg A_i \text{ and } 1 \le i \le n \} \\
\Theta_U^-(G) = \{\theta \mid\ & C \in neg(U),\ mgu^*(L_i, C) = \theta, \\
& L_i \text{ is a positive literal and } 1 \le i \le n \ \} \\
\cup\ \ \{\theta \mid\ & C \in pos(U),\ mgu^*(A_i, C) = \theta, \\
& L_i = \neg A_i \text{ and } 1 \le i \le n \}
\end{aligned}
$$

We say that G *is* potentially added *by* U *iff* $\Theta_U^+(G) \neq \emptyset$. *Also,* G *is* potentially deleted *by* U *iff* $\Theta_U^-(G) \neq \emptyset$.

Note that trivially $\Theta_U^+(G) \neq \emptyset$ iff $\Theta_U^+(\leftarrow L_i) \neq \emptyset$ for some literal L_i of G. The method by Lloyd, Sonenberg and Topor in [53] can roughly be seen as simplifying the integrity constraints by calculating $\Theta_U^+(\leftarrow Body_i)$ for each body $Body_i$ of an integrity constraint and then instantiating the integrity constraints using the so obtained set of substitutions.

For the Example 2.4 above we obtain:

$\Theta_U^+(\leftarrow man(\text{x}), woman(\text{x})) = \{\{\text{x}/a\}\}$ and

$\Theta_U^+(\leftarrow parent(\text{x}, \text{Y}), parent(\text{Y}, \text{x})) = \emptyset$

and thus obtain the following set of simplified integrity constraints:

$\{false \leftarrow man(a), woman(a)\}$

Checking this simplified constraint is of course much more efficient than entirely re-checking all the integrity constraints of Example 2.4.

In our method we will use the substitutions Θ_U^+ slightly differently. First though, we characterise the derivations in a database after some update, which were not present before the update.

Definition 2.6 (incremental SLDNF-derivation)
Let $U = \langle Db^+, Db^=, Db^- \rangle$ be a database update and let δ be an SLDNF-derivation after U. A derivation step of δ will be called incremental *iff it resolves a positive literal with a clause from Db^+ or if it selects a ground negative literal $\neg A$ such that $\leftarrow A$ is potentially deleted by U.*
We say that δ is incremental *iff it contains at least one incremental derivation step.*

The treatment of negative literals in the above definition is not optimal. In fact "$\leftarrow A$ is potentially deleted by U" does not guarantee that the same derivation does not exist in the database state prior to an update. However an optimal criterion, due to its complexity, has not been implemented in the current approach.

Lemma 2.7
Let G be a goal and U a database update. If there exists an incremental derivation for G after U, then G is potentially added by U.
Proof:
Let $U = \langle Db^+, Db^=, Db^- \rangle$ and let δ be the incremental derivation for G after U. We define δ' to be the incremental derivation $G_0 = G, G_1, \ldots, G_k$ for G after U, obtained by stopping at the first incremental derivation step of δ.
Base Case: There are two possibilities: either a positive literal $L_i = A_i$ or a negative literal $L_i = \neg A_i$ has been selected inside G_{k-1} at the last (incremental) step. In the first case the goal G_{k-1} has been resolved with a standardised apart[4] clause $A \leftarrow Body \in Db^+$ with $mgu(A_i, A) = \theta$. Thus by Definition 2.3 we have $A \in pos(U)$ and by Definition 2.5 we obtain $\theta \in \Theta_U^+(\leftarrow L)$.

In the second case $\Theta_U^-(\leftarrow A_i) \neq \emptyset$ and by Definition 2.5: $\exists C \in neg(U)$ such that $mgu^*(A_i, C) = \theta$. Hence we know that $\theta \in \Theta_U^+(\leftarrow L)$. In both cases $\Theta_U^+(\leftarrow L) \neq \emptyset$ and it follows that the goal G_{k-1} is potentially added by U.

---

[4]So far we have not provided a formal definition of the notion of "standardising apart". Several ones, correct and incorrect, exist in the literature (see e.g. the discussion in [35] or [22]). Just suppose for the remainder of this proof that fresh variables, not occurring "anywhere else", are used.

Induction Step: We can now prove by induction that the set of goals $\{G_{k-2}, \ldots, G_0\}$ are also potentially added. Let us suppose that $G_m = \leftarrow L_1, \ldots, L_n$, with $1 \le m \le k-1$, is potentially added. We know that for at least one literal L_i we have that $\Theta_U^+(\leftarrow L_i) \neq \emptyset$.

If a negative literal has been selected in the derivation step from G_{m-1} to G_m then G_{m-1} is also potentially added, because all the literals L_i also occur unchanged in G_{m-1}.

If a positive literal L'_j has been selected in the derivation step from G_{m-1} to G_m and resolved with the (standardised apart) clause $A \leftarrow B_1, \ldots, B_q \in Db^=$, with $mgu(L'_j, A) = \theta$, we have: $G_{m-1} = \leftarrow L'_1, \ldots, L'_j, \ldots, L'_r$, $G_m = \leftarrow (L'_1, \ldots, L'_{j-1}, B_1, \ldots, B_q, L'_{j+1}, \ldots, L'_r)\theta$.

There are again two cases. Either there exists a L'_p, with $1 \le p \le r \wedge p \neq j$, such that $\Theta_U^+(\leftarrow L'_p\theta) \neq \emptyset$. In that case we have (because $mgu^*$is used inside Definition 2.5) that $\Theta_U^+(\leftarrow L'_p) \neq \emptyset$ and G_{m-1} is potentially added. Note that this is *not* the case if we use just the *mgu* without standardising apart. As already pointed out this has been overlooked in [53,55]. Take for instance $L'_p = p(a, \textsc{y}, b, \textsc{x})$ and $\theta = \{\textsc{x}/\textsc{y}, \textsc{y}/\textsc{x}\}$. Then $L'_p\theta$ unifies with $p(\textsc{x}, a, \textsc{y}, b) \in pos(U)$ and the more general L'_p does not!

In the other case there only exists a B_p, with $1 \le p \le q$, such that $\Theta_U^+(\leftarrow B_p\theta) \neq \emptyset$. If B_p is a positive literal, we know by Definition 2.5 that: $\exists C \in pos(U)$ with $mgu^*(B_p\theta, C) = \sigma$. From the fact that C is standardised apart before unifying with $B_p\theta$ we know that for some θ': $mgu^*(B_p, C) = \theta'$ (again this is *not* the case if we use the *mgu*). Hence by Definition 2.3 we can conclude that a variant of $A\theta'$ is an element of $pos(U)$. It only remains to be proven that $mgu^*(L'_j, A\theta')$ exists. We know that θ' is the most general unifier of B_p and a standardised apart version C^* of C and we also know that $\theta\sigma$ is a unifier of B_p and C^* (because the variables of θ and C^* are disjoint). Hence θ' is more general than $\theta\sigma$, i.e. for some γ we have $\theta'\gamma = \theta\sigma$. From this we can deduce that $A\theta'\gamma = A\theta\sigma$. Let us now take the standardised apart version A^* of $A\theta'$ that will be used in the calculation of $mgu^*(L'_j, A\theta')$. We know that for some γ' we have that $A^*\gamma' = A\theta\sigma$. We also know (by standardising apart) that the domain of γ' has no variables in common with the domain of $\theta\sigma$ and hence $\gamma \cup \theta\sigma$ is a well defined substitution. And indeed, $\gamma \cup \theta\sigma$ is a unifier for $\{L'_j, A^*\}$ and hence a most general unifier must exist. We can thus conclude that $\Theta_U^+(\leftarrow L'_j) \neq \emptyset$ and that G_{m-1} is potentially added.

The proof is almost identical for the case that B_p is a negative literal. In summary all the goals $\{G_{k-1}, \ldots, G_0\}$ are potentially added and thus also $G = G_0$. $\qquad\square$

Definition 2.8 (relevant SLDNF-derivation)
Let δ be a (possibly incomplete) SLDNF-derivation after $U = \langle Db^+, Db^=, Db^- \rangle$ and let $G_0, G_1, \ldots$ be the sequence of goals of δ. We say that δ is a relevant *derivation after U iff for each G_i we either have that G_i is potentially added by U or δ_i is incremental after U, where δ_i is the sub-derivation leading from G_0 to G_i.*

A refutation being a particular derivation we can specialise the concept and define *relevant refutations*. The following theorem will form the basis of our method for performing specialised integrity checking.

Theorem 2.9 (Incremental Integrity Checking)
Let $U = \langle Db^+, Db^=, Db^- \rangle$ be a database update such that there is no SLDNF refutation

before U for the goal $\leftarrow false$.
Then $\leftarrow false$ has a SLDNF-refutation after U iff $\leftarrow false$ has a relevant refutation after U.

Proof:

$\Leftarrow$: If $\leftarrow false$ has a relevant refutation then it trivially has a refutation, namely the relevant one.

$\Rightarrow$: The refutation must be incremental, because otherwise the derivation is also valid for $Db^= \cup Db^-$ and we have a contradiction. Let $G_0 = \leftarrow false, G_1, \ldots, G_k = \square$ be the incremental refutation. For each G_i, we either have that G_i occurs after the first incremental derivation step and hence the sub-derivation δ_i, leading from G_0 to G_i, is incremental. If on the other hand G_i is situated before the first incremental derivation step, we can use Lemma 2.7 to infer that G_i is potentially added. Thus the derivation conforms to Definition 2.8 and is relevant. $\square$

In other words, if we know that the integrity constraints of a deductive database were not violated before an update, then we only have to search for a *relevant* refutation of $\leftarrow false$ in order to check the integrity constraints after the update.

The method can best be illustrated by re-examining Example 2.4. The goals in the SLD-tree in Figure 1 are annotated with their corresponding sets of substitutions Θ_U^+. The SLD-derivation leading to $\leftarrow parent(x, y), parent(y, x)$ is not relevant and can therefore be pruned. Similarly all derivations descending from the goal $\leftarrow man(x), woman(x)$ which do not use $Db^+ = \{man(a) \leftarrow\}$ are not relevant either and can also be pruned. However the derivation leading to $\leftarrow woman(a)$ is incremental and is relevant even though $\leftarrow woman(a)$ is not potentially added.
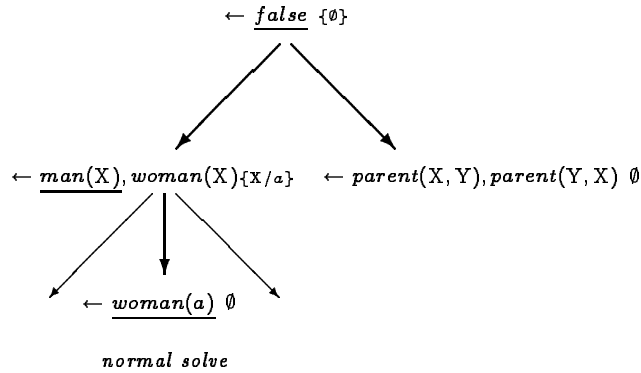

Figure 1: SLDNF-tree for example 2.4

Note that the above method can be seen as an extension of the method in [53], because Θ_U^+ is not only used to simplify the integrity constraints at the topmost level (i.e. affecting the bodies of integrity constraints), but is used throughout the testing of the integrity

constraints to prune non-relevant branches. An example where this aspect is important will be presented in Section 6.

Note however that the method in [53] not only removes integrity constraints but also instantiates them, possibly generating several specialised integrity constraints for a single unspecialised one. This instantiation often considerably reduces the number of matching facts and is therefore often vital for improving the efficiency of the integrity checks. The Definition 2.8 of relevant derivations does not use Θ_U^+ to instantiate intermediate goals. The reasons for this are purely practical, namely, to keep the meta-interpreter as simple as possible for effective partial evaluation. Definition 2.8 could actually be easily adapted to use Θ_U^+ for instantiating goals and Theorem 2.9 would still be valid. But, surprisingly, the instantiations will often be performed by the partial evaluation method itself and the results in Section 6 illustrate this. We will further elaborate on these aspects in Section 4.

3 The Meta-interpreter

3.1 Specialised Update Procedures

Specialised integrity checking, like the method of the previous section, can be implemented through a meta-interpreter, manipulating updates and databases 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 strategy can be incorporated into the meta-program.

Furthermore, by partial evaluation of this meta-interpreter, we may (in principle) be able to pre-compile the integrity checking for certain update patterns. Let us re-examine Example 2.4. For the concrete update of Example 2.4, with $Db^+ = \{man(a) \leftarrow\}$, a meta-interpreter implementing the method of the previous section would try to find a refutation for $\leftarrow false$ in the manner outlined in Figure 1. By specialising this meta-interpreter for an update pattern $Db^+ = \{man(\mathcal{A}) \leftarrow\}$, $Db^- = \emptyset$, where $\mathcal{A}$ is not yet known, one might (hopefully) obtain a *specialised update procedure*, efficiently checking integrity, essentially as follows:

$$inconsistent(add(man(A))) \leftarrow evaluate(woman(A))$$

Given the actual value for $\mathcal{A}$, this procedure will basically check consistency in a similar manner to the unspecialised meta-interpreter, but will do this much more efficiently, because the propagation, simplification and evaluation process is already *pre-compiled*. For instance, the derivation in Figure 1 leading to $\leftarrow parent(\text{x}, \text{y})$, $parent(\text{y}, \text{x})$ has already been pruned at specialisation time. Similarly all derivations descending from the goal $\leftarrow man(\text{x})$, $woman(\text{x})$, which do not use Db^+, have also already been pruned at specialisation time. Finally, the specialised update procedure no longer has to calculate $pos(U)$ and $neg(U)$ for the concrete update U. All of this can lead to very high efficiency gains.

Both [80] and [74] 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

10

and partial deduction (the main concern of [74] is to show why using inconsistency indicators instead of integrity constraints is relevant for efficiency and a good idea in general).

In this section we will present a meta-interpreter for specialised integrity checking which is based on Theorem 2.9. This meta-interpreter will act on object level expressions (terms, atoms, goals, clauses, ...) which represent the deductive database under consideration. So, before presenting the meta-interpreter in more detail, it is advisable to discuss the issue of representing these object level expressions at the meta-level, i.e. inside the meta-interpreter.

3.2   The Ground, Non-Ground and Mixed Representations

In logic programming, there are basically two opposing schools of thought on how an object level expression, say the atom $p(\mathrm{x}, a)$, should be represented at the meta-level. The first school would use the term $p(\mathrm{x}, a)$ as the representation, while the second one would use something like the term $struct(p, [var(1), struct(a, [])])$. The first term is a *non-ground* representation which represents an object-level variable as a meta-level variable. The second one is a *ground* representation which represents an object-level variable as a ground term. Some examples of the ground representation that will be used throughout this paper are presented in Figure 2.

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

Figure 2: A ground representation

The ground representation has the advantage that it can be treated purely declaratively, while for many applications the non-ground representation requires the use of extra-logical built-ins. The non-ground representation also has semantical problems (although they were solved to some extent in [18, 58, 59]). The main 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. The current 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 the reader to [31], [6, 32], the conclusion of [58] or the extended version of [48]. Some further comments concerning these issues will also be made in Section 7.

Sometimes however it is possible to combine both approaches into one. This was first exemplified by Gallagher in [26, 27], where an interpreter for the ground representation is

presented which lifts the ground representation to the non-ground one for resolution. We will call this approach the *mixed* representation. A similar technique was used in the self-applicable partial evaluator Logimix [33,62]. Hill and Gallagher [31] also provide a recent account of this style of writing meta-interpreters with its uses and limitations. With that technique we can use the versatility of the ground representation for representing object level expressions, while not suffering an enormous speed decrease. Furthermore, as demonstrated by Gallagher in [26] and in the results of our experiments, partial evaluation can in this way sometimes completely remove the overhead of the ground representation. Performing a similar feat on a meta-interpreter using the ground representation and explicit unification is much harder and has, to the best of our knowledge, not been accomplished yet (for some promising attempts see the partial evaluator SAGE [7, 29, 30], or the new scheme for the ground representation in [48]).

Instead of using the mixed representation, one may also think of simply using the non-ground representation to write our meta-interpreter. There are actually several reasons why this is not such a good idea. One disadvantage of the non-ground representation is that it is more difficult to specify partial knowledge for partial evaluation. Suppose for instance that we know that a given atom (for instance the head of a fact that will be added to a deductive database) will be of the form $man(\mathcal{T})$, where $\mathcal{T}$ is a constant, but we don't know yet at partial evaluation time which particular constant $\mathcal{T}$ stands for. In the ground representation the this can be expressed by writing the atom as $struct(man, [struct(c, [])])$. However in the non-ground representation we have to write this as $man(x)$, which is unfortunately less precise, as the variable x now no longer represents only constants but stands for any term.[5]

There is unfortunately another caveat to using the non-ground representation: either the object-program has to be stored explicitly using meta-program clauses, instead of using a term-representation of the object program, or non-logical built-ins like *copy*/2 have to be used to perform the standardising apart. Figure 3 illustrates these two possibilities. Note that without the *copy* in Figure 3, the second meta-interpreter would incorrectly fail for the given query. For our application this means that, on the one hand, using the non-logical copying approach unduly complicates the specialisation task while at the same time leading to a serious efficiency bottleneck. On the other hand, using the clause representation, implies that representing updates to a database becomes much more cumbersome. Basically we also have to encode the updates explicitly as meta-program clauses, thereby making dynamic meta-programming (see e.g. [31]) impossible.

So, the most promising option for our application is to use the mixed representation. As such, our meta-interpreter contains a predicate *make_non_ground*/2, which lifts a ground term to a non-ground one. For instance the query

$\leftarrow make\_non\_ground(struct(f, [var(1), var(2), var(1)]), x)$

succeeds with a computed answer similar to

---

[5]A possible way out is to use the $= ../2$ built-in and represent the atom by $man(X), X = ..[c]$. This requires that the partial evaluator provides non-trivial support for the built-in $= ../2$ (to ensure for instance that the information about X, provided by $X = ..[c]$, is properly used and propagated).

1. Using a Clause Representation	2. Using a Term Representation
$solve([]) \leftarrow$	$solve(\text{P}, []) \leftarrow$
$solve([\text{H}|\text{T}]) \leftarrow$	$solve(\text{P}, [\text{H}|\text{T}]) \leftarrow$
$\quad clause(\text{H}, \text{B})$	$\quad member(\text{Cl}, \text{P}), copy(\text{Cl}, cl(\text{H}, \text{B}))$
$\quad solve(\text{B}), solve(\text{T})$	$\quad solve(\text{P}, \text{B}), solve(\text{P}, \text{T})$
$clause(p(X), []) \leftarrow$	
$\leftarrow solve([p(a), p(b)])$	$\leftarrow solve([cl(p(X), [])], [p(a), p(b)])$

Figure 3: Two non-ground meta-interpreters with $\{p(\text{x}) \leftarrow\}$ as object program

$$
\begin{aligned}
&make\_non\_ground(\text{GrTerm}, \text{NgTerm}) \leftarrow \\
&\quad mng(\text{GrTerm}, \text{NgTerm}, [], \_\text{Sub}) \\
\\
&mng(var(\text{N}), \text{X}, [], [sub(\text{N}, \text{X})]) \leftarrow \\
&mng(var(\text{N}), \text{X}, [sub(\text{M}, \text{Y})|T], [sub(\text{M}, \text{Y})|\text{T}_1]) \leftarrow \\
&\quad (\text{N} = \text{M} \rightarrow (\text{T}_1 = \text{T}, \text{X} = \text{Y}) \; ; \; mng(var(\text{N}), \text{X}, \text{T}, \text{T}_1)) \\
&mng(struct(\text{F}, \text{GrArgs}), struct(\text{F}, \text{NgArgs}), \text{InSub}, \text{OutSub}) \leftarrow \\
&\quad l\_mng(\text{GrArgs}, \text{NgArgs}, \text{InSub}, \text{OutSub}) \\
\\
&l\_mng([], [], \text{Sub}, \text{Sub}) \leftarrow \\
&l\_mng([\text{GrH}|\text{GrT}], [\text{NgH}|\text{NgT}], \text{InSub}, \text{OutSub}) \leftarrow \\
&\quad mng(\text{GrH}, \text{NgH}, \text{InSub}, \text{InSub}_1), \\
&\quad l\_mng(\text{GrT}, \text{NgT}, \text{InSub}_1, \text{OutSub})
\end{aligned}
$$

Figure 4: Lifting the ground representation

$\{\text{x}/struct(f, [\_49, \_57, \_49])\}$.

The variables $\_49$ and $\_57$ are fresh variables. Their actual names may vary and are not important. The code for this predicate is presented in Figure 4 and a simple meta-interpreter based on it can be found in Figure 5. In contrast to the meta-interpreter presented in [26], this meta-interpreter can be executed without specialisation. Note that we use an if-then-else construct, written as (if -> then ; else). For the time being we assume this construct to be declarative, i.e. it is just a logical connective as in Gödel [32]. But note that even the Prolog if-then-else will behave in a "declarative" way, because the first argument to $make\_non\_ground$ will always be ground.

We can now use Theorem 2.9 to extend the interpreter in Figure 5 for specialised integrity checking. Based on Theorem 2.9, we know that we can stop resolving a goal G when it is not potentially added, unless we have performed an *incremental* resolution step earlier in the derivation.

$$
\begin{aligned}
&solve(\text{Prog}, [\,]) \leftarrow \\
&solve(\text{Prog}, [\text{H}|\text{T}]) \leftarrow \\
&\qquad non\_ground\_member(struct(clause, [\text{H}|\text{Body}]), \text{Prog}), \\
&\qquad solve(\text{Prog}, \text{Body}), \\
&\qquad solve(\text{Prog}, \text{T}) \\[1em]
&non\_ground\_member(\text{NonGrTerm}, [\text{GrH}|\text{GrT}]) \leftarrow \\
&\qquad make\_non\_ground(\text{GrH}, \text{NonGrTerm}) \\
&non\_ground\_member(\text{NonGrTerm}, [\text{GrH}|\text{GrT}]) \leftarrow \\
&\qquad non\_ground\_member(\text{NonGrTerm}, \text{GrT})
\end{aligned}
$$

Figure 5: An interpreter for the ground representation

$$
\begin{aligned}
&incremental\_solve(\text{Updt}, \text{Goal}) \leftarrow \\
&\qquad potentially\_added(\text{Updt}, \text{Goal}), \\
&\qquad resolve(\text{Updt}, \text{Goal}) \\[1em]
&resolve(\text{Updt}, \text{Goal}) \leftarrow \\
&\qquad resolve\_unincrementally(\text{Updt}, \text{Goal}, \text{NewGoal}), \\
&\qquad incremental\_solve(\text{Updt}, \text{NewGoal}) \\
&resolve(\text{Updt}, \text{Goal}) \leftarrow \\
&\qquad resolve\_incrementally(\text{Updt}, \text{Goal}, \text{NewGoal}), \\
&\qquad \text{Updt} = \text{``}\langle Db^+, Db^=, Db^- \rangle\text{''}, \\
&\qquad solve(\text{``}Db^= \cup Db^+\text{''}, \text{NewGoal})
\end{aligned}
$$

Figure 6: Skeleton of the integrity checker

The skeleton of our meta-interpreter in Figure 6 implements this idea (the full Prolog code can be found in Appendix A). Note that the argument Updt contains the ground representation of the update $\langle Db^+, Db^=, Db^- \rangle$. Also, from now on, we use "$\mathcal{T}$" to denote the ground representation of a term $\mathcal{T}$.

The predicate *resolve_incrementally/3* performs incremental resolution steps (according to Definition 2.6) and *resolve_unincrementally/3* performs non-incremental ones. The predicate *potentially_added/3* tests whether a goal is potentially added by an update according to Definition 2.5. Specialised integrity checking now consists in calling

$$\leftarrow incremental\_solve \left(\text{``}\langle Db^+, Db^=, Db^- \rangle\text{''}, \leftarrow false \right)$$

The query will succeed if the integrity of the database has been violated by the update. In Subsection 3.3 we will examine how this meta-interpreter can be unfolded by a partial evaluator and in Section 4 we will study the implementation of the predicate *potentially_added/2*.

14

3.3 Unfolding the Meta-interpreter

In this subsection we will examine how the meta-interpreter of Figure 6 can be unfolded by a partial evaluator. The discussions are also valid for the simpler meta-interpreter of Figure 5.

Unfolding a meta-interpreter in a satisfactory way is a non-trivial issue and has been the topic of a lot of contributions [4,29,39,56,57,63,65,78]. For the fully general case, this problem has not been solved yet. However, using a non-ground representation for goals in the meta-interpreter greatly simplifies the control of unfolding. In fact, a simple variant test[6] inside the partial evaluator can quite often be sufficient to guarantee termination when unfolding such a meta-interpreter. This is illustrated in Figure 7, where intermediate goals have been removed for clarity and where Prog represents an object program inside of which the predicate $p/1$ is recursive via $q/1$. The meta-interpreter unfolded in the left column uses a ground representation for resolution and a variant test of the partial evaluator will not detect a loop. The partial evaluator will have to abstract away the constants 1 and 3 in order to terminate and generate specialised code.[7] However if we unfold the meta-interpreter of Figure 5, the variant test is sufficient to detect the loop and no abstraction is needed to generate efficient specialised code. This point is completely independent of the internal representation the partial evaluator uses, i.e. of the fact whether the partial evaluator itself uses a ground or a non-ground representation — it might even be written in another programming language.

A ground solve	*Non-ground solve of Figure 5*
$solve(\mathrm{Prog}, [struct(p, [var(1)])])$	$solve(\mathrm{Prog}, [p(\_1)])$
$\downarrow$	$\downarrow$
$solve(\mathrm{Prog}, [struct(q, [var(3)])])$	$solve(\mathrm{Prog}, [q(\_3)])$
$\downarrow$	$\downarrow$
$solve(\mathrm{Prog}, [struct(p, [var(3)])])$	$solve(\mathrm{Prog}, [p(\_3)])$

Figure 7: Unfolding meta-interpreters

The partial evaluator which was used in the experiments of this paper actually uses the variant test combined with annotations of the program to be specialised and the use of the mixed representation simplified the control of unfolding. We will give more details on the particular unfolding strategy of the partial evaluator in Section 5.4.

Let us suppose that the issue of unfolding our meta-interpreter is solved. Then there remains only one issue for specialisation: namely how to implement *potentially_added* such

---

[6]Meaning that the partial evaluator stops unfolding when it comes upon a variant of a selected atom which it has already encountered higher up in the proof tree.

[7]Note that reformulating the variant test so that it takes the ground representation of the meta-interpreter into account solves the problem only partially, because residual clauses generated for $solve(\mathrm{Prog}, struct(p, [var(1)]))$ cannot be used for $solve(\mathrm{Prog}, struct(p, [var(3)]))$, i.e. abstraction is mandatory. However for more involved object programs, like the reverse with accumulating parameter, the variant test is not sufficient anyway and abstraction is mandatory for both approaches anyhow.

that it can be effectively partially evaluated. This turns out to be non-trivial as well and in the next section we propose a solution for hierarchical databases.

4 Implementing *potentially_added*

The rules of Definition 2.3, which are at the basis of the predicate *potentially_added*, can be directly transformed into a simple logic program which detects in a naive top-down way whether a goal is potentially added or not. Making abstraction of the particular representation of clauses and programs, we might write *potentially_added* like this:

$$potentially\_added(A, \langle DB^+, DB^=, DB^- \rangle) \leftarrow$$
$$A \leftarrow \ \in DB^+$$
$$potentially\_added(A, \langle DB^+, DB^=, DB^- \rangle) \leftarrow$$
$$A \leftarrow \ldots, A', \ldots \in DB^=,$$
$$potentially\_added(A', \langle DB^+, DB^=, DB^- \rangle)$$

Such an approach terminates for hierarchical databases and is very easy to partially evaluate. It will however lead to a predicate which has multiple (and maybe identical and/or covered[8]) solutions and which might instantiate the (non-ground) goal under consideration. This means that, to ensure completeness, we would either have to backtrack and try out a lot of useless instantiations[9], or collect all solutions and perform expensive subsumption tests to keep only the most general ones. The latter approach would have to make use of a `findall` primitive as well as a non-declarative instance test, both of which are very hard to partially evaluate satisfactorily; e.g. effective partial evaluation of `findall` has to the best of our knowledge not been accomplished yet. Let us illustrate this problem through an example.

Example 4.1 *Let the following clauses be the rules of $Db^=$:*

$$\boxed{\begin{array}{l} mother(\mathrm{X}, \mathrm{Y}) \leftarrow \ parent(\mathrm{X}, \mathrm{Y}), woman(\mathrm{X}) \\ father(\mathrm{X}, \mathrm{Y}) \leftarrow \ parent(\mathrm{X}, \mathrm{Y}), man(\mathrm{X}) \\ false \leftarrow \ mother(\mathrm{X}, \mathrm{Y}), father(\mathrm{X}, \mathrm{Z}) \end{array}}$$

Let $Db^- = \emptyset$ and $Db^+ = \{parent(a, b) \leftarrow, man(a) \leftarrow\}$ and as usual $U = \langle Db^+, Db^=, Db^- \rangle$. A naive top-down implementation will succeed 3 times for the query

$$\leftarrow potentially\_added(\leftarrow false, \text{``}U\text{''})$$

and twice for the query

---

[8]A computed answer θ of a goal G is called *covered* if there exists another computed answer θ' of G such that $G\theta$ is a strict instance of $G\theta'$.

[9]It would also mean that we would have to extend Theorem 2.9 to allow for instantiation, but this is not a major problem.

$$\leftarrow potentially\_added(\leftarrow father(\text{x}, \text{y}), ``\text{u}")$$

with computed answers $\{\text{x}/a\}$ and $\{\text{x}/a, \text{y}/b\}$. Note that the solution $\{\text{x}/a, \text{y}/b\}$ is "covered" by $\{\text{x}/a\}$ (which means that, if floundering is not possible, it is useless to instantiate the query by applying $\{\text{x}/a, \text{y}/b\}$).

The above example shows that using a naive top-down implementation inside the integrity checker of Figure 6 is highly inefficient, because a lot of redundant checking will occur. The solution to this problem is to wrap calls to the predicate *potentially_added/1* into a *verify*(.) primitive, which succeeds once with the empty computed answer if its argument succeeds (in any way) and fails otherwise. This solves the problem of duplicate and covered solutions.

For instance for Example 4.1 above, both

$$\leftarrow verify(potentially\_added(\leftarrow false, ``\text{u}"))$$
$$\leftarrow verify(potentially\_added(\leftarrow father(\text{x}, \text{y}), ``\text{u}"))$$

will succeed just once with the empty computed answer and no backtracking is required, because no instantiations are made. The *verify*(.) primitive can be implemented with the Prolog if-then-else construct in the following way:

```
((Goal->fail;true)->fail;true).
```

Although this usage of the if-then-else is no longer declarative, we can still apply the partial evaluation method of [42] which incorporates extensive support for the if-then-else. This partial evaluation method will be presented in the next section.

The disadvantage of using *verify* is of course that no instantiations are performed (which in general cut down the search space dramatically). However, as mentioned before, these instantiations can often be performed by the partial evaluation method through pruning and safe left- and right-propagation of bindings. We will come back to this point in the next section.

5   Partial Evaluation of RLP

"Real-life Logic Programming" or simply RLP, is a practically usable subset of Prolog encompassing simple built-ins, simple side-effects and the if-then-else. For the application in this paper, RLP turned out to be ideal. On the one hand, the extra power of if-then-else (but nothing more) was required to implement a meta-interpreter for specialised integrity checking in deductive databases. On the other hand, the restriction to RLP allowed us to produce more efficient specialised programs than could be obtained by current partial evaluation systems for full Prolog.

5.1 Definition of RLP

We first define the RLP language. The syntax of RLP is based on the syntax of definite logic programs with the following extensions and modifications.

The definition of *terms* remains the same. The set of *predicates* P is partitioned into the set of "normal" predicates P_{cl} defined through clauses and the set of built-in predicates P_{bi}. A *normal atom* is an atom which is constructed using a predicate symbol $\in P_{cl}$. Similarly a *built-in atom* is constructed using a predicate symbol $\in P_{bi}$.

A *RLP-atom* is either a normal atom, a built-in atom or it is an expression of the form $(if \rightarrow then; else)$ where $if, then$ and $else$ are conjunctions of RLP-atoms. We will denote by *Goals* the set of all conjunctions of RLP-atoms.

A *RLP-clause* is an expression of the form $Head \leftarrow Body$ where $Head$ is a normal atom and $Body$ is a conjunction of RLP-atoms.

As can be seen from the above, RLP does not incorporate the negation nor the cut, but uses the if-then-else construct instead. This construct behaves just like the Prolog version of the if-then-else, which contains a local cut and whose behaviour is as follows:

1. If the test-part succeeds then a local cut is executed and the then-part is entered.

2. If the test-part fails finitely then the else-part is entered.

3. If the test-part "loops" (i.e. fails infinitely) then the whole construct loops.

Most uses of the cut can actually be mapped to if-then-else constructs and the if-then-else can also be used to implement the negation. Also, for a lot of practical programming tasks, the if-then-else can be used to write much more efficient code when compared to pure logic programs.

Using the if-then-else construct, or some of the Prolog built-ins for that matter, has important consequences on the semantic level. Because the if-then-else contains a local cut, it is sensitive to the sequence of computed answers of the test-part. This means that the computation rule and the search rule have to be fixed in order to give a clear meaning to the if-then-else. From now on we will presuppose the Prolog left-to-right computation rule and the lexical search rule. SLD-trees and derivations following this convention will be called LD-trees and derivations.

The two RLP-programs hereafter illustrate the importance of the order of the solutions for the if-then-else:

Program P_1	Program P_2
$q(\mathrm{x}) \leftarrow (p(\mathrm{x}) \rightarrow r(\mathrm{x}); fail)$	$q(\mathrm{x}) \leftarrow (p(\mathrm{x}) \rightarrow r(\mathrm{x}); fail)$
$p(a) \leftarrow$	$p(c) \leftarrow$
$p(c) \leftarrow$	$p(a) \leftarrow$
$r(c) \leftarrow$	$r(c) \leftarrow$

Using the Prolog computation and search rules, the query $\leftarrow q(\mathrm{x})$ will fail for program P_1, whereas it will succeed for P_2. All we have done is change the order of the computed

answers for the predicate $p/1$. This implies that a partial evaluator which handles the if-then-else has to preserve the sequence of computed answers of all goals prone to be used inside an if-then-else test-part. This for instance is not guaranteed by the partial deduction framework of [52], which only preserves the computed answers but not their sequence.

So the added power of the if-then-else has its price in terms of a more complex partial evaluation procedure. In turn however, the if-then-else, is much better suited for partial evaluation than the "full blown" cut. For instance, a simple denotational semantics, along the line of the semantics described in [2] and [69], can be given to RLP and the unfolding techniques are also much simpler (see [42] and the example below). Using the if-then-else instead of the cut was already advocated by O'Keefe in [64] and performed by Takeuchi and Furukawa in [79].

5.2   Specialising RLP

In order to preserve the sequence of computed answers, we have to address the problem of *left-propagation* of bindings. For instance, unfolding a non-leftmost atom in a clause might instantiate the head of the clause or the atoms to the left of it. In the context of extra-logical built-ins this can change the program's behaviour. But even without built-ins, this left-propagation of bindings can change the order of solutions which, as we have seen above, can lead to incorrect transformations for programs containing the if-then-else. In the example below, P_4 is obtained from P_3 by unfolding the non-leftmost atom $q(\mathrm{Y})$, thereby changing the sequence of computed answers.

Program P_3	Program P_4
$p(\mathrm{X},\mathrm{Y}) \leftarrow q(\mathrm{X}), \underline{q(\mathrm{Y})}$	$p(\mathrm{X},a) \leftarrow q(\mathrm{X})$
$q(a) \leftarrow$	$p(\mathrm{X},b) \leftarrow q(\mathrm{X})$
$q(b) \leftarrow$	$q(a) \leftarrow$
	$q(b) \leftarrow$
Sequence of computed answers for $\leftarrow p(\mathrm{X},\mathrm{Y})$	
$\langle p(a,a), p(a,b), p(b,a), p(b,b) \rangle$	$\langle p(a,a), p(b,a), p(a,b), p(b,b) \rangle$

This problem of left-propagation of bindings has been solved in various ways in the partial evaluation literature [66,67,72,73], as well as overlooked in some contributions (e.g. [24]). In the context of unfold/fold transformations of pure logic programs, preservation of the order of solutions, as well as left-termination, is handled e.g. in [5,69].

The solution that has been used in [42] for RLP, is to strictly enforce the Prolog left-to-right selection rule. However, sometimes one does not want to select the leftmost atom, for instance because it is a built-in which is not sufficiently instantiated, or simply to ensure termination of the partial evaluation process. To cope with this this problem, the concept of LD-derivations and LD-trees has been extended to LDR-derivations and LDR-trees, which in addition to containing left-most resolution steps also contain *residualisation* steps, which remove the left-most atom from the goal and hide it from the left-propagations of bindings. Generating the residual code from LDR-trees is discussed in [42].

19

Unfolding inside the if-then-else is also handled in a rather straightforward manner. This is in big contrast to programs which contain the full blown cut. The reason is that the full cut can have an effect on all subsequent clauses defining the predicate under consideration. By unfolding, the scope of the cut can be changed, thereby altering the behaviour of the program. See [12], [72, 73] or [66, 67] for the elaborate techniques that are required to solve this problem. The cut inside the if-then-else however is local and does not affect the reachability and meaning of other clauses. It is therefore much easier to handle by a partial evaluator. The following example illustrates this. Unfolding $q(x)$ in the program with the if-then-else poses no problems and leads to a correct specialised program. However unfolding the same atom in the program written with the cut leads to an incorrect specialised program for which e.g. $q(a)$ is no longer a consequence.

Program P_5	Program P_6
$p(x) \leftarrow (\underline{q(x)} \rightarrow fail; true)$	$p(x) \leftarrow \underline{q(x)}, !, fail$
	$p(x) \leftarrow$
$q(x) \leftarrow (x = a \rightarrow fail; true)$	$q(x) \leftarrow X = a, !, fail$
	$q(x) \leftarrow$
Unfolded Programs	
$p(x) \leftarrow ((x = a \rightarrow fail; true)$	$p(x) \leftarrow X = a, !, fail, !, fail$
$\rightarrow fail$	$p(x) \leftarrow !, fail$
$; true)$	$p(x) \leftarrow$

The exact details on how to unfold the if-then-else can be found in [42]. In [42] we also showed that freeness and sharing information, which so far have been of little interest in (pure) partial deduction, can be important to produce efficient specialised programs by removing useless bindings as well as useless if-then-else tests. The latter often occur when predicates with output arguments are used inside the test-part of an if-then-else.

A further improvement lies in generating multiple versions of a predicate call according to varying freeness information of the arguments. In our implementation we have accomplished this by integrating the freeness and sharing analysis into the partial evaluation process and using more refined notions of "instance" and "variant". Our system might thus generate two versions for a given goal $\leftarrow p(x)$: one version where x is guaranteed to be free and one where it is not. Under some circumstances this can substantially improve the quality of the generated code.

In summary, partial evaluation of RLP, can be situated somewhere in the middle between partial deduction of *pure* logic programs (see [8, 28, 52, 60, 61] and systems like SP [26, 27], SAGE [29, 30] or more recently ECCE [43, 45, 49]) and partial evaluation of *full* Prolog (e.g. MIXTUS [72, 73] or PADDY [66–68]).

A partial evaluation system (LEUPEL), which includes all the techniques sketched in this section, has been developed. The implementation originally grew out of [41]. In Section 6 we will apply this system to obtain specialised update procedures. In the next two subsections we present two more aspects of that system, which are relevant for our application.

5.3 Safe Left-Propagation of Bindings

At the end of Section 4 we pointed out that a disadvantage of using *verify* is that no instantiations are performed. Fortunately these instantiations can often be performed by the partial evaluation method through pruning and safe left- and right-propagation of bindings. Take for instance a look at the specialised update procedure presented in Figure 10 of Section 6 and generated for the update $Db^+ = \{man(a) \leftarrow\}$, $Db^- = \emptyset$. This update procedure tests directly whether $woman(\mathcal{A})$ is a fact, whereas the original meta-interpreter of Figure 6 would test whether there are facts matching $woman(\mathrm{x})$ and only afterwards prune all irrelevant branches. This instantiation performed by the partial evaluator is in fact the reason for the extremely high speedup figures presented in the results of Section 6. In a sense, part of the specialised integrity checking is performed by the meta-interpreter and part is performed by the partial evaluator.

The above optimisation was obtained by unfolding and pruning all irrelevant branches. In some cases we can also improve the specialised update procedures by performing a *safe left-propagation* of bindings. As we have seen in the previous subsection, left-propagation of bindings is in general unsafe. There are however some circumstances where bindings can be left-propagated without affecting the correctness of the specialised program. The following example illustrates such a safe left-propagation, as well as its benefits for efficiency.

Example 5.1 *Take the following clause, which might be part of a specialised update procedure.*

$$incremental\_solve\_1(\mathrm{A}) \leftarrow parent(\mathrm{X},\mathrm{Y}), (a\_test \rightarrow \mathrm{X} = \mathrm{A}, \mathrm{Y} = b; \mathrm{X} = \mathrm{A}, \mathrm{Y} = c)$$

Suppose that a_test does not generate side-effects and that the c.a.s. of parent/2 are always grounding substitutions. The former always holds in our case, because our incremental_solve meta-interpreter of Appendix A is written completely without side-effects. The latter is always guaranteed for range restricted database predicates (see e.g. [11]). In that case the binding $\mathrm{X} = \mathrm{A}$ *can be left-propagated in the following way:*

$$incremental\_solve\_1(\mathrm{A}) \leftarrow \underline{\mathrm{X} = \mathrm{A}}, parent(\mathrm{X},\mathrm{Y}), (a\_test \rightarrow \mathrm{Y} = b; \mathrm{Y} = c)$$

This clause will generate the same sequence of computed answers than the original clause, but will do so much more efficiently. Usually, there will be lots of parent/2 facts and incremental_solve_1 will be called with A *instantiated. Therefore, the second clause will be much more efficient — the call to parent will just succeed once for every child of* A *instead of succeeding for the entire parent relation.*

The *leupel* partial evaluator contains a post-processing phase which performs safe left-propagation of common bindings, like $\mathrm{X} = \mathrm{A}$ above. However, the analysis performed by the partial evaluator is not yet optimal. By implementing a more precise analysis of the if-then-else structure the speedup figures could still be improved. Also, a more aggressive propagation of bindings could be envisaged. Currently only the parts of the bindings which are common to *all* alternatives (like $\mathrm{X} = \mathrm{A}$ above) are propagated, meaning

that no additional choice-points are generated. This conservative approach can never lead to deterioration, but it is also not always optimal. In the context of large deductive databases, it is usually beneficial to also left-propagate the non-common bindings. For instance, for Example 5.1 above, we might also left-propagate the bindings concerning X, thereby producing the following clauses:

$$incremental\_solve\_1(\text{A}) \leftarrow \underline{\text{X} = \text{A}, \text{Y} = b}, parent(\text{X}, \text{Y}), (a\_test \rightarrow true; fail)$$
$$incremental\_solve\_1(\text{A}) \leftarrow \underline{\text{X} = \text{A}, \text{Y} = c}, parent(\text{X}, \text{Y}), (a\_test \rightarrow fail; true)$$

In general this specialised update procedure will be even more efficient. Again, this indicates that the results in Section 6 can be even further improved.

5.4 Control of Unfolding

In this subsection we describe some further details about the unfolding strategy used by the LEUPEL system, and indicate how we are able to handle the task of specialising the *incremental_solve* meta-interpreter of Appendix A.

The partial evaluation process performed by LEUPEL has been decomposed into three phases:

1. the *annotation phase*, which annotates the program to be specialised by giving indications of how the predicate calls should be unfolded,

2. the *specialisation phase*, which performs the unfolding guided by the annotation of the first phase,

3. the *post-processing phase*, which performs optimisation on the generated partial deductions (i.e. removes useless bindings, performs safe left-propagation of bindings, simplifies the residual code) and generates the residual program.

Such a decomposition has already proven to be useful for self-application in the world of functional programming (see e.g. [33]) as well as for logic programming ([62], [29]).

Unfortunately, the annotation phase of LEUPEL is not yet automatic and must usually be performed by hand. On the positive side, this gives the knowledgeable user very precise control over the unfolding, especially since some quite sophisticated annotations are provided for. More precisely, the user can give (on-line) conditions on

1. when an atom should be *evaluated* (E) without further testing,

2. when it should be *unfolded once* (U)
   (unless a loop is detected at partial evaluation time),

3. when it should be *residualised* (R)
   (i.e. the atom should be left untouched).

For instance the user can specify that the atom $var(x)$ should be fully evaluated only if its argument is ground or if its argument is guaranteed to be free (at evaluation time) and that it should be residualised otherwise. Our partial evaluation method can thus be seen as being "semi on-line" in the sense that some unfolding decisions are made off-line while others are still made on-line. This idea has recently been taken up for functional programming in [77]. Also note that the filters in the partial evaluator Schism for applicative languages (see [15]) also allow for conditions on when to unfold (U) and when to residualise (R). They are however used in a purely off-line fashion.

In our case, the approach of hand annotating the meta-interpreter is very sensible. Indeed, given proper care, the same annotated meta-program can be used for *any* kind of update pattern. Therefore, investing time in annotating the meta-interpreter of Appendix A — which only has to be done *once* — gives high benefits for all consecutive applications and the second and third phases of LEUPEL will then be able to derive specialised update procedures **fully automatically**, as exemplified by the prototype [44].

6   Experiments and Results

6.1   An Example

Before showing the results of our method, let us first illustrate in what sense it *improves* upon the method of Lloyd *et al* in [53].

Example 6.1 *Let the rules in Figure 8 form the intensional part of $Db^=$ and let $Db^+ = \{man(a) \leftarrow\}$, $Db^- = \emptyset$. Then, independently of the facts in $Db^=$, we have that:*

$$pos(U) = \{man(a), father(a, \_), married\_to(a, \_),$$
$$married\_man(a), unmarried(a), false\}$$
$$neg(U) = \{unmarried(a), false\}$$

The method of Lloyd et al *[53] will then generate the following simplified integrity constraints:*

$$false \leftarrow man(a), woman(a)$$
$$false \leftarrow parent(a, \text{Y}), unmarried(a)$$

Given the available information, this simplification of the integrity constraints is not optimal. Suppose that some fact matching $parent(a, \text{Y})$ exists in the database. Evaluating the second simplified integrity constraints above, then leads to the incomplete SLDNF-tree depicted in Figure 9 and subsequently to the evaluation of the goal:

$$\leftarrow woman(a), \neg married\_woman(a)$$

This goal is not potentially added *and the derivation leading to the goal is not* incremental. *Hence, by Theorem 2.9, this derivation can be pruned and will never lead to a*

successful refutation, given the fact that the database was consistent before the update. The incremental_solve meta-interpreter of Appendix A improves upon this and does not try to evaluate the goal: ← woman(a), ¬married_woman(a).

$$
\begin{aligned}
&mother(\text{X}, \text{Y}) \leftarrow parent(\text{X}, \text{Y}), woman(\text{X}) \\
&father(\text{X}, \text{Y}) \leftarrow parent(\text{X}, \text{Y}), man(\text{X}) \\
&grandparent(\text{X}, \text{Z}) \leftarrow parent(\text{X}, \text{Y}), parent(\text{Y}, \text{Z}), \\
&married\_to(\text{X}, \text{Y}) \leftarrow \\
&\qquad parent(\text{X}, \text{Z}), parent(\text{Y}, \text{Z}), \\
&\qquad man(\text{X}), woman(\text{Y}) \\
&married\_man(\text{X}) \leftarrow married\_to(\text{X}, \text{Y}) \\
&married\_woman(\text{X}) \leftarrow married\_to(\text{Y}, \text{X}) \\
&unmarried(\text{X}) \leftarrow man(\text{X}), \neg married\_man(\text{X}) \\
&unmarried(\text{X}) \leftarrow woman(\text{X}), \neg married\_woman(\text{X}) \\
\\
&false \leftarrow man(\text{X}), woman(\text{X}) \\
&false \leftarrow parent(\text{X}, \text{Y}), parent(\text{Y}, \text{X}) \\
&false \leftarrow \quad parent(\text{X}, \text{Y}), unmarried(\text{X})
\end{aligned}
$$

Figure 8: Intensional part of $Db^=$

Actually, through partial evaluation of the *incremental_solve* meta-interpreter, this useless branch is already pruned at specialisation time. For instance, when generating a specialised update procedure for the update pattern $Db^+ = \{man(\mathcal{A}) \leftarrow\}$, $Db^- = \emptyset$, we obtain the update procedure presented in Figure 10.[10] This update procedure is very satisfactory and is in a certain sense optimal. The only way to improve it, would be to add the information that the predicates in the intensional and the extensional database are disjoint. For most applications this is the case, but it is not required by the current method. This explains the seemingly redundant test in Figure 10, checking whether there is a fact *married_to* in the database. Benchmarks concerning this example will be presented in Section 6.2.

6.2 Comparison with Other Partial Evaluators

In this subsection, we perform some experiments with the database of Example 6.1. The goal of these experiments is to compare the partial evaluation technique presented in Section 5 with existing partial evaluators for full Prolog and give a first impression of the potential of our approach. In Subsection 6.3, we will do a more extensive study on a more

---

[10]The figure actually contains a slightly sugared and simplified version of the resulting update procedure. There is no problem whatsoever, apart from finding the time for coding, to directly produce the sugared and simplified version. Also, all the benchmarks were executed on un-sugared and un-simplified versions.
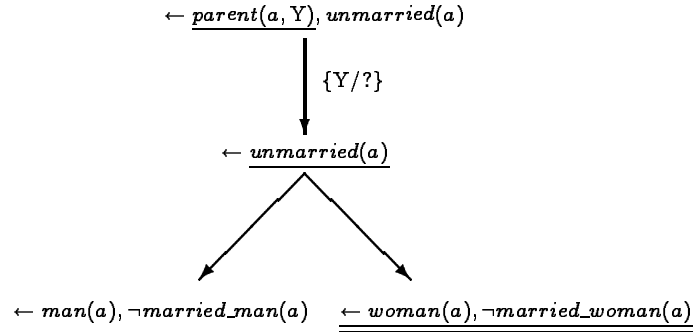
$$\leftarrow \underline{parent(a, \text{Y})}, unmarried(a)$$

$$\{\text{Y}/?\}$$

$$\leftarrow \underline{unmarried(a)}$$

$$\leftarrow man(a), \neg married\_man(a) \qquad \leftarrow woman(a), \neg married\_woman(a)$$

Figure 9: SLDNF-tree for example 6.1

```
incremental_solve_1(X1) :-
    fact(woman,.(struct(X1,[]),[])).
incremental_solve_1(X1) :-
    fact(parent,.(struct(X1,[]),.(X2,[]))),
    (fact(married_to,.(struct(X1,[]),.(X3,[])))
     -> fail
    ;
     ((fact(parent,.(struct(X1,[]),.(X4,[]))),
       fact(parent,.(X3,.(X4,[]))),
       fact(woman,.(X3,[])) )
       -> fail
       ; true
      )
    ).
```

Figure 10: Specialised update procedure for adding $man(\mathcal{A})$

complicated database, with more elaborate transactions, and show the benefits compared to the Lloyd *et al.* method [53].

The times for the benchmark are expressed in seconds and were obtained by calling the *time*/2 predicate of Prolog by BIM, which incorporates the time needed for garbage collection, see [70]. We used sets of 400 updates and a fact database consisting of 108 facts and 216 facts respectively. The rule part of the database is presented in Figure 8. Also note that, in trying to be as realistic as possible, the fact part of the database has been simulated by Prolog facts. The tests were executed on a **Sun Sparc Classic** running under Solaris 2.3.

The following different integrity checking methods were benchmarked:

1. **solve:** This is the naive meta-interpreter of Figure 5. It does not use the fact that the database was consistent before the update and simply tries to find a refutation for $\leftarrow false$.

2. **ic-solve:** This is the *incremental_solve* meta-interpreter performing specialised in-

tegrity checking, as described in Section 3. The skeleton of the meta-interpreter can be found in Figure 6, the full code is in Appendix A.

3. **ic-leupel:** These are the specialised update procedures obtained by specialising *ic-solve* with the partial evaluation system LEUPEL described in Section 5. A prototype, based on LEUPEL, performing these specialisations fully automatically, is publicly available in [44]. This prototype can also be used to get the timings for *solve* and *ic-solve* above as well as *ic-leupel⁻* below.

4. **ic-leupel⁻:** These are also specialised update procedures obtained by LEUPEL, but this time with the safe left-propagation of bindings (see Section 5.3) disabled.

5. **ic-mixtus:** These specialised update procedures were obtained by specialising *ic-solve* using the automatic partial evaluator MIXTUS described in [72, 73]. Version 0.3.3 of MIXTUS, with the default parameter settings, was used in the experiments.

6. **ic-paddy:** These specialised update procedures were obtained by specialising *ic-solve* using the automatic partial evaluator PADDY presented in [66–68]. The resulting specialised procedures had to be slightly converted for Prolog by BIM: `get_cut/1` had to be transformed into `mark/1` and `cut_to/1` into `cut/1`. We also had to increase the "term_depth" parameter of PADDY from its default value. With the default value, PADDY actually slowed down the *ic-solve* meta-interpreter by about 30 %.

The first experiment we present consists in generating an update procedure for the update pattern:

$$Db^+ = \{man(\mathcal{A}) \leftarrow\},\ Db^- = \emptyset,$$

where $\mathcal{A}$ is unknown at partial evaluation time. The result of the partial evaluation obtained by LEUPEL can be seen in Figure 10 and the timings are summarised in Table 1. The first row of figures contains the absolute and relative times required to check the integrity for a database with 108 facts. The second row contains the corresponding figures for a database with 216 facts.

solve	*ic-solve*	*ic-leupel*	*ic-leupel⁻*	*ic-mixtus*	*ic-paddy*
108 facts					
42.93 s	6.81 s	0.075 s	0.18 s	0.34 s	0.27 s
572.4	90.8	1	2.40	4.53	3.60
216 facts					
267.9 s	18.5 s	0.155 s	0.425 s	0.77 s	0.62 s
1728.3	119.3	1	2.74	4.96	4.00

Table 1: Results for $Db^+ = \{man(\mathcal{A}) \leftarrow\},\ Db^- = \emptyset$

26

The time needed to obtain the *ic-leupel* specialised update procedure was 78.19 s. Note that the current implementation of LEUPEL has a very slow post-processor, displays tracing information and uses the ground representation. Therefore, it is almost certainly possible to reduce the time needed for partial evaluation by at least one order of magnitude. Still, even using the current implementation, the time invested into partial evaluation should pay off rather quickly for larger databases. Also, LEUPEL seemed to be faster than MIXTUS and almost half as fast as PADDY.[11]

In another experiment we generated a specialised update procedure for the following update pattern:

$$Db^+ = \{parent(\mathcal{A}, \mathcal{B}) \leftarrow\},\ Db^- = \emptyset,$$

where $\mathcal{A}$ and $\mathcal{B}$ are unknown at partial evaluation time. This update pattern offers less opportunities for specialisation than the previous one. The speedup figures are still satisfactory but less spectacular. The results are summarised in the following Table 2.

solve	ic-solve	ic-leupel	ic-leupel⁻	ic-mixtus	ic-paddy
108 facts					
43.95 s	7.75 s	0.24 s	0.355 s	0.53 s	0.45 s
183.1	32.3	1	1.48	2.21	1.88
216 facts					
273.1 s	21.9 s	0.915 s	1.16 s	1.67 s	1.435 s
298	23.9	1	1.26	1.82	1.57

Table 2: Results for $Db^+ = \{parent(\mathcal{A}, \mathcal{B}) \leftarrow\},\ Db^- = \emptyset$

In summary, the speedups obtained with the LEUPEL system are very encouraging. The specialised update procedures execute up to 2 orders of magnitude faster than the intelligent incremental integrity checker *ic-solve* and up to 3 orders of magnitude faster than the non-incremental *solve*. The latter speedup can of course be made to grow to almost any figure by using larger databases. Note that, according to our experience, specialising the *solve* meta-interpreter of Figure 5 usually yields speedups reaching at most 1 order of magnitude.

Also, the specialised update procedures obtained by using the LEUPEL system performed between 1.6 and 5 times faster than the ones obtained by MIXTUS or PADDY. Finally, note that the safe left-propagation of bindings described in Section 5.3 has a definite, beneficial effect on the efficiency of the specialised update procedures.

6.3 A More Comprehensive Study

In this subsection, we perform a more elaborate study of the specialised update procedures generated by LEUPEL and compare their efficiency with the one of the well established

---

[11]Exact comparisons where not made because MIXTUS runs under Sicstus Prolog, PADDY under Eclipse and LEUPEL under Prolog by BIM.

technique by Lloyd *et al* [53], which often performs very well in practice. To that end we will use a more sophisticated database and more complicated transactions. The rules and integrity constraints $Db^=$ of the database are taken from the most complicated example in [74] and can be found in Appendix C.

For the benchmarks of this subsection, *solve*, *ic-solve* and *ic-leupel* are the same as in the Subsection 6.2. In addition we also have the integrity checking method *ic-lst*, which is an implementation of the method of Lloyd *et al* [53] and whose code can be found in Appendix B.[12]

For the more elaborate benchmarks, we used 5 different update patterns. The results are summarised in the Tables 3, 4, 5, 6 and 7. The particular update pattern, for which the specialised update procedures were generated, figures in the table description. Different concrete updates, all instances of the given update pattern, were used to measure the efficiency of the methods. The second column of each table contains the integrity constraints violated by each concrete update. For each particular concrete update, the first row contains the absolute times for 100 updates and, in order to show the speedups, the second row contains the relative time wrt to *ic-leupel*. Each table is divided into sub-tables for databases of different sizes.

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
109 facts					
1	{2,2}	45.80 s	13.77 s	0.05 s	5.52 s
		916	275	1	110
2	{8}	46.10 s	13.70 s	0.06 s	5.49 s
		768	228	1	92
3	{}	47.00 s	13.42 s	0.05 s	5.45 s
		940	268	1	109
218 facts					
1	{2,2}	108.10 s	23.07 s	0.10 s	5.54 s
		1081	231	1	55
2	{8}	108.20 s	23.62 s	0.13 s	5.59 s
		832	182	1	43
3	{}	108.00 s	22.92 s	0.09 s	5.49 s
		1200	255	1	61

Table 3: Results for $Db^+ = \{father(\mathcal{X}, \mathcal{Y}) \leftarrow\}$, $Db^- = \emptyset$

As can be seen from the benchmark tables, the update procedures generated by LEUPEL perform extremely well. In Table 6, LEUPEL detected that there is no way this update can violate the integrity constraints — hence the "infinite" speedup. For 218 facts, the speedups

---

[12]We also tried a "dirty" implementation using assert and retracts to store the potential updates. But to our surprise this solution ran slower than the one shown in Appendix B, which stores the potential updates in a list.

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
109 facts					
1	{6}	46.20 s 330	65.49 s 468	0.14 s 1	11.76 s 84
2	{a1, a1, 10}	46.60 s 291	66.47 s 415	0.16 s 1	10.96 s 69
3	{}	46.40 s 357	65.27 s 502	0.13 s 1	9.95 s 77
218 facts					
1	{6}	107.50 s 371	132.52 s 457	0.29 s 1	11.91 s 41
2	{a1, a1, 10}	108.50 s 362	134.37 s 448	0.30 s 1	11.19 s 37
3	{}	109.00 s 376	135.06 s 466	0.29 s 1	9.99 s 34

Table 4: Results for $Db^+ = \{civil\_status(\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}) \leftarrow\}$, $Db^- = \emptyset$

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
109 facts					
1	{}	47.50 s 250	73.82 s 389	0.19 s 1	17.79 s 94
218 facts					
2	{}	109.20 s 295	153.28 s 414	0.37 s 1	17.93 s 48

Table 5: Results for $Db^+ = \{father(\mathcal{F}, \mathcal{X}), civil\_status(\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}) \leftarrow\}$, $Db^- = \emptyset$

in the other tables range from 295 to 1200 over *solve*, from 77 to 466 over *ic-solve* and from 34 to 68 over $lst - solve$. These speedups are very encouraging and lead us to conjecture that the approach presented in this paper can be very useful in practice and lead to drastic efficiency improvements.

Of course, the larger the database becomes, the more time will be needed on the actual evaluation of the simplified constraints and not on the simplification. That is why the relative difference between *ic-lst* and *ic-leupel* diminishes with a growing database. In the worst case, namely the figures for 872 facts in Table 7, *ic-leupel* "only" runs 6 times faster than *ic-lst*. But for all examples tested so far, using an evaluation mechanism which is slower than in "real" database systems and therefore exaggerates the effect of the size of the database on the benchmark figures, the difference remains big.

We also measured heap consumption of *ic-leupel*, which used from 61 to 307 times less

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
109 facts					
1	{}	45.80 s "∞"	4.03 s "∞"	0.00 s 1	5.90 s "∞"
2	{}	45.60 s "∞"	4.02 s "∞"	0.00 s 1	5.94 s "∞"
218 facts					
1	{}	109.80 s "∞"	4.12 s "∞"	0.00 s 1	5.90 s "∞"
2	{}	108.50 s "∞"	4.04 s "∞"	0.00 s 1	5.84 s "∞"

Table 6: Results for $Db^+ = \emptyset$, $Db^- = \{father(\mathcal{X}, \mathcal{Y}) \leftarrow\}$

heap space than *ic-solve*. Finally, in a small experiment we also tried to specialise *ic-lst* for the update patterns using MIXTUS, but without much success. Speedups were of the order of 10%.

7 Moving to Recursive Databases

7.1 The RLP Approach

As we have seen in the previous section, we were able to produce highly efficient update procedures for hierarchical databases by partial evaluation of meta-interpreters. The question is whether these results can be extended in a straightforward way to recursive, stratified databases, maybe by (slightly) adapting the meta-interpreter.

In theory the answer should be yes. The "only" added complication is that in a top-down method, a loop check has to be incorporated into *potentially_added*, to ensure termination. This loop check must act on the non-ground representation of the goal and can, for Datalog programs, be based on keeping a history of goals and using the instance or variant check to detect loops. Unfortunately, we found out, through first initial experiments, that such a loop check is very difficult to partially evaluate satisfactorily: often partial evaluation leads to an explosion of alternatives in the residual code. Furthermore, only a fraction of these alternatives is actually reachable, the reasons being as follows.

Firstly, good specialisation of such a loop check requires intricate specialisation of built-ins using freeness information. For example, one has to detect that, if x is guaranteed to be free, then $p(a)$ is always an instance of $p(x)$. Although being far from obvious, this specialisation might still be obtained by the LEUPEL system.

There are however further difficulties that have to be overcome. For instance, the partial evaluator must be able to detect that something like $p(f(\mathcal{A}))$ is always an instance of $p(\mathcal{A})$, no matter what $\mathcal{A}$ stands for. Such reasoning requires analysing infinitely many

Update	ICs viol	solve	ic-solve	ic-leupel	ic-lst
109 facts				
1	{8,8,9a}	45.90 s 242	17.95 s 94	0.19 s 1	15.58 s 82
2	{8,8,9a}	47.20 s 295	14.79 s 92	0.16 s 1	11.13 s 70
3	{}	46.40 s 580	14.45 s 181	0.08 s 1	11.01 s 138
4	{}	45.90 s 417	17.57 s 160	0.11 s 1	15.37 s 140
218 facts				
1	{8,8,9a}	107.20 s 306	30.03 s 86	0.35 s 1	15.44 s 44
2	{8,8,9a}	107.10 s 357	23.12 s 77	0.30 s 1	11.13 s 37
3	{}	107.90 s 674	22.62 s 141	0.16 s 1	10.95 s 68
4	{}	106.90 s 445	29.30 s 122	0.24 s 1	15.25 s 64
436 facts				
1	{8,8,9a}	285.30 s 297	53.76 s 56	0.96 s 1	15.76 s 16
872 facts				
1	{8,8,9a}	854.80 s 297	100.89 s 35	2.88 s 1	17.01 s 6

Table 7: Results for $Db^+ = \emptyset$, $Db^- = \{civil\_status(\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}) \leftarrow\}$

computations, something which standard partial evaluation cannot do. This explains that partial evaluation of a meta-interpreter containing a loop check will often lead to a code explosion with a lot of unreachable constructs.

In order to overcome this problem, stronger partial evaluation methods are needed. Very recently, an approach has been proposed in [50] which combines partial deduction with bottom-up abstract interpretation capabilities. This approach is often able to extract information from infinitely many computations. So, combining LEUPEL and its freeness analysis with the approach of [50] might result in a system which can obtain specialised update procedures for recursive, stratified databases. Further work will be needed to establish this.

7.2 The Declarative Approach

Another approach has been pursued in [47, 48]. There it was attempted to use *pure* logic programs and partial deduction in the hope of overcoming the above mentioned problems.

In the course of this work, it turned out that any loop check for the non-ground representation or the mixed representation is non-declarative by nature. 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-ins, 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. In fact, the exact same reasoning holds for the predicate $instance/2$. Finally, because the goals of the mixed representation are also in non-ground form, loop checking cannot be added declaratively to the mixed representation as well.

This insight has prompted us in [47, 48] to use the ground representation for the meta-interpreter performing specialised integrity checking. But then, contrary to what one might expect, partial deduction was unable to specialise this meta-interpreter in an interesting way and no specialised update procedures could be obtained.

The crucial problem in [48] boiled down to a lack of information propagation at the object level. Indeed, the ground representation has to make use of an explicit unification algorithm. Take for instance a meta-interpreter which implements specialised integrity checking as outlined in Section 2. To calculate the set of positive potential updates $pos(U)$, the meta-interpreter will select an atom $C \in pos^i(U)$, unify it with an atom B in the body of a clause $A \leftarrow \ldots, B, \ldots \in Db^=$ and then apply the unifier to the head A to obtain an induced, potential update of $pos^{i+1}(U)$. At partial deduction time, the atoms A, B and C are in general not fully known. 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 no substantial compilation will occur and it will be impossible to obtain efficient specialised update procedures.

To solve this problem, again an infinite number of different computations have to be analysed — something which standard partial deduction cannot do. The problem was solved in [48] via a new implementation of the ground representation combined with a custom specialisation technique. Some promising results were obtained, but the results are still far away from the good results obtained for hierarchical databases in this paper. One reason was that the problems mentioned above for the instance check using the non-ground representation also persist to some extent with the ground representation. Also, as we already discussed in Section 5.4, it is more difficult to properly unfold meta-interpreters which use the ground representation instead of the non-ground or the mixed representation. So, also for the ground representation, further work, using maybe the techniques developed in [50], will be required to handle recursive databases in a satisfactory way.

8 Conclusion

We presented the idea of obtaining specialised update procedures for deductive databases in a principled way: namely by writing a meta-interpreter for specialised integrity checking and then partially evaluating this meta-interpreter for certain update patterns. The goal was to obtain specialised update procedures which perform the integrity checking much more efficiently than the generic integrity checking methods.

In this paper we have first described this approach in general and then presented a new integrity checking method well suited for partial evaluation. We then discussed several implementation details of this integrity checking method and gained insights into issues concerning the ground versus the non-ground representation. We notably argued for the use of a "mixed" representation, in which the object program is represented using the ground representation, but where the goals are lifted to the non-ground representation for resolution. This approach has the advantage of using the flexibility of the ground representation for representing knowledge about the object program (in our case the deductive database along with the updates), while using the efficiency of the non-ground representation for resolution. The mixed representation is also much better suited for partial evaluation than the full ground representation.

In a first approach, we have restricted ourselves to normal, hierarchical databases. Also, for efficiency purposes, the meta-interpreter for specialised integrity checking had to make use of a *verify* construct. In essence, the *verify* construct is used to test whether a given goal, encountered while checking the integrity of a database upon an update, might be influenced by that update. If this is not the case the integrity checker will stop the derivation of the goal.

This *verify* primitive can be implemented via the if-then-else construct. We have thus presented an extension of pure Prolog, called RLP, which incorporates the if-then-else and we have presented how this language can be specialised. We have drawn upon the techniques in [42] and presented the partial evaluator LEUPEL and a prototype [44] based upon it, which can generate specialised update procedure fully automatically.

This prototype has been used to conduct extensive experiments, the results of which were very encouraging. Speedups reached and exceeded 2 orders of magnitude when specialising the integrity checker for a given set of integrity constraints and a given set of rules. These high speedups are also due to the fact that the partial evaluator performs part of the integrity checking. We also compared the specialised update procedures with the well known approach in [53], and the results show that big performance improvements, also reaching and exceeding 2 orders of magnitude, can be obtained.

To summarise, it seems that partial evaluation is capable of automatically generating highly specialised update procedures for hierarchical databases with negation.

Future Directions

In the current paper we have discussed how to extend the results to recursive, stratified databases and we have indicated that further work, using maybe the techniques developed

in [50], will be needed to handle recursion in a fully satisfactory manner.

One might also apply the techniques of this paper to other meta-interpreters, which have a more flexible way of specifying static and dynamic parts of the database and are less entrenched in the concept that facts change more often than rules and integrity constraints.

Another important point is the efficiency of generating the specialised update procedures, as opposed to running them. For the examples presented in this paper, the update procedures have to be re-generated when the rules or the integrity constraints change. A technique, based on work by Benkerimi and Shepherdson [3], could be used to incrementally adapt the specialised update procedure whenever the rules or integrity constraints change.[13] Another approach might be based on using a *self-applicable* partial evaluation system in order to obtain efficient update procedure compilers by self-application.

On the level of practical applications, one might try to apply the methods of this paper to abductive and inductive logic programs. For instance, we conjecture that solvers for abduction, like the SLDNFA procedure [21], can greatly benefit in terms of efficiency, by generating specialised integrity checking procedures for each abducible predicate.

Finally, it might also be investigated whether partial evaluation *alone* is able to derive specialised integrity checks. In other words, is it possible to obtain specialised integrity checks by partially evaluating a simple solve meta-interpreter, like the one of Figure 5. In that case, self-applicable partial evaluation could be used to obtain specialised update procedures by performing the second Futamura projection [23, 25] and update procedure compilers by performing the third Futamura projection. Combining partial deduction with abstract interpretation, extending e.g. [50] for richer abstract domains, might provide a way of achieving this goal.

Acknowledgements

References

[1] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.

---

[13]Thanks to Bern Martens for pointing this out.

[2] M. Baudinet. Proving termination of Prolog programs: A semantic approach. *The Journal of Logic Programming*, 14(1 & 2):1–29, 1992.

[3] K. Benkerimi and J. C. Shepherdson. Partial deduction of updateable definite logic programs. *The Journal of Logic Programming*, 18(1):1–27, January 1994.

[4] R. Bol. Loop checking in partial deduction. *The Journal of Logic Programming*, 16(1&2):25–46, 1993.

[5] A. Bossi, N. Cocco, and S. Etalle. Transformation of left terminating programs: The reordering problem. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, Lecture Notes in Computer Science 1048, pages 33–45, Utrecht, Netherlands, September 1995. Springer-Verlag.

[6] A. Bowers. Representing Gödel object programs in Gödel. Technical Report CSTR-92-31, University of Bristol, November 1992.

[7] 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.

[8] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.

[9] F. Bry, , H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In J. Schmidt, S. Ceri, and M. Missikoff, editors, *Proceedings of the International Conference on Extending Database Technology*, Lecture Notes in Computer Science, pages 488–505, Venice, Italy, 1988. Springer-Verlag.

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

[11] 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.

[12] M. Bugliesi and F. Russo. Partial evaluation in Prolog: Some improvements about cut. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, pages 645–660. MIT Press, 1989.

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

[14] 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.

[15] C. Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154. ACM Press, 1993.

[16] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of POPL'93*, Charleston, South Carolina, January 1993. ACM Press.

[17] S. Das and M. Williams. A path finding method for constraint checking in deductive databases. *Data & Knowledge Engineering*, 4:223–244, 1989.

[18] D. De Schreye and B. Martens. A sensible least Herbrand semantics for untyped vanilla meta-programming. In A. Pettorossi, editor, *Proceedings Meta'92*, Lecture Notes in Computer Science 649, pages 192–204. Springer Verlag, 1992.

[19] 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.

[20] 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.

[21] M. Denecker and D. De Schreye. SLDNFA; an abductive procedure for normal abductive programs. In K. Apt, editor, *Proceedings of the International Joint Conference and Symposium on Logic Programming, Washington*, 1992.

[22] K. Doets. Levationis laus. *Journal of Logic and Computation*, 3(5):487–516, 1993.

[23] A. P. Ershov. On Futamura projections. *BIT (Japan)*, 12(14):4–5, 1982. In Japanese.

[24] H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2 & 3):91–118, 1988.

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

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

[27] 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.

[28] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS, Schloß Dagstuhl, 1996. To Appear. Extended version as Technical Report CW 221, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[35] H.-P. Ko and M. E. Nadel. Substitution and refutation revisited. In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 679–692. MIT Press, 1991.

[36] J. Komorowksi. *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1981. Linköping Studies in Science and Technology Dissertations 69.

[37] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, pages 49–69. Springer-Verlag, LNCS 649, 1992.

[38] 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.

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

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

[41] M. Leuschel. Self-applicable partial evaluation in Prolog. Master's thesis, K.U. Leuven, 1993.

[42] 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.

[43] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR '95*, Lecture Notes in Computer Science 1048, pages 1–16, Utrecht, Netherlands, September 1995. Springer-Verlag.

[44] M. Leuschel. Prototype partial evaluation system to obtain specialised integrity checks by specialising meta-interpreters. Prototype Compulog II, D 8.3.3, Departement Computerwetenschappen, K.U. Leuven, Belgium, September 1995. Obtainable at `ftp://ftp.cs.kuleuven.ac.be/pub/compulog/ICLeupel/`.

[45] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.cs.kuleuven.ac.be/~lpai`, 1996.

[46] 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.

[47] 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.

[48] 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*, pages 495–509, Portland, USA, December 1995. MIT Press. Extended version as Technical Report CW 210, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[49] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS, Schloß Dagstuhl,

1996. To Appear. Extended version as Technical Report CW 220, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[50] M. Leuschel and D. Schreye. Logic program specialisation: How to be more specific. In *Proceedings of PLILP'96*, LNCS, Aachen, Germany, September 1996. To Appear. Extended version as Technical Report CW 232, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

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

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

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

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

[55] J. W. Lloyd and R. W. Topor. A basis for deductive database systems. *The Journal of Logic Programming*, 2:93–109, 1985.

[56] 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.

[57] B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, K.U. Leuven, February 1994.

[58] 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.

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

[60] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996. To Appear.

[61] B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122(1–2):97–117, 1994.

[62] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 214–227. Springer-Verlag, 1992.

[63] G. Neumann. A simple transformation from Prolog-written metalevel interpreters into compilers and its implementation. In A. Voronkov, editor, Logic Programming. *Proceedings of the First and Second Russian Conference on Logic Programming*, Lecture Notes in Computer Science 592, pages 349–360. Springer-Verlag, 1991.

[64] R. O'Keefe. On the treatment of cuts in Prolog source-level tools. In *Proceedings of the Symposium on Logic Programming*, pages 68–72. IEEE, 1985.

[65] S. Owen. Issues in the partial evaluation of meta-interpreters. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 319–339. MIT Press, 1989.

[66] S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

[67] S. Prestwich. An unfold rule for full Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, Workshops in Computing, University of Manchester, 1992. Springer-Verlag.

[68] S. Prestwich. Online partial deduction of large programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 111–118. ACM Press, 1993.

[69] M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics based Program Manipulation, PEPM'91*, Sigplan Notices, Vol. 26, N. 9, pages 274–284, Yale University, New Haven, U.S.A., 1991.

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

[71] F. Sadri and R. Kowalski. A theorem-proving approach to database integrity. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 9, pages 313–362. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1988.

[72] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, Swedish Institute of Computer Science, Mar. 1991.

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

[74] R. Seljée. A new method for integrity constraint checking in deductive databases. *Data & Knowledge Engineering*, 15:63–102, 1995.

[75] J. C. Shepherdson. Language and equality theory in logic programming. Technical Report PM-91-02, University of Bristol, 1991.

[76] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *The Journal of Logic Programming*, 1996. To Appear.

[77] M. Sperber. How to have your cake and eat it too: Self-applicable online partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS, Schloß Dagstuhl, 1996. To Appear.

[78] L. Sterling and R. D. Beer. Metainterpreters for expert system construction. *The Journal of Logic Programming*, 6(1 & 2):163–178, 1989.

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

[80] 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 Specialised Integrity Checking

```
/* ------------------------------------ */
/* normal_solve(GrFacts,GrRules,NgGoal) */
/* ------------------------------------ */

/* This normal_solve makes no assumptions about the Facts and the Rules.
   For instance the predicates defined in Rules can also be present
   in Facts and vice versa */

normal_solve(GrXtraFacts,GrDelFacts,GrRules,[]).
normal_solve(GrXtraFacts,GrDelFacts,GrRules,[not(NgG)|NgT]) :-
    (normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos(NgG)])
     -> fail
     ;  (normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT))
    ).
normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos(NgH)|NgT]) :-
    db_fact_lookup(NgH),
    not(non_ground_member(NgH,GrDelFacts)),
    normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT).
normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos(NgH)|NgT]) :-
    non_ground_member(NgH,GrXtraFacts),
    normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT).
normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos(NgH)|NgT]) :-
    non_ground_member(term(clause,[pos(NgH)|NgBody]),GrRules),
    normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgBody),
    normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT).
```

```
/* --------------------------- */
/*    INCREMENTAL IC CHECKER    */
/* --------------------------- */

incremental_solve(GoalList,DB) :-
    verify_one_potentially_added(GoalList,DB),
    inc_resolve(GoalList,DB).

inc_resolve([pos(NgH)|NgT],DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    db_fact_lookup(NgH),
    not(non_ground_member(NgH,DeletedFacts)),
    incremental_solve(NgT,DB).
inc_resolve([pos(NgH)|NgT],DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(NgH,AddedFacts),
    /* print(found_added_fact(NgH)),nl, */
    append(AddedRules,ValidOldRules,NewRules),
    normal_solve(AddedFacts,DeletedFacts,NewRules,NgT).
inc_resolve([pos(NgH)|NgT],DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(NgH)|NgBody]),ValidOldRules),
    append(NgBody,NgT,NewGoal),
    incremental_solve(NewGoal,DB).
inc_resolve([pos(NgH)|NgT],DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(NgH)|NgBody]),AddedRules),
    append(AddedRules,ValidOldRules,NewRules),
    normal_solve(AddedFacts,DeletedFacts,NewRules,NgBody),
    normal_solve(AddedFacts,DeletedFacts,NewRules,NgT).
inc_resolve([not(NgH)|NgT],DB) :-
    DB = db(AddedFacts,DeletedFacts,
                ValidOldRules,AddedRules,DeletedRules),
    append(AddedRules,ValidOldRules,NewRules),
    (normal_solve(AddedFacts,DeletedFacts,NewRules,[pos(NgH)])
     -> (fail)
     ;  (verify_potentially_added(not(NgH),DB)
    -> (normal_solve(AddedFacts,DeletedFacts,NewRules,NgT))
    ;  (incremental_solve(NgT,DB))
        )
    ).

verify_one_potentially_added(GoalList,DB) :-
    ( (one_potentially_added(GoalList,DB) -> fail ; true)
    -> fail
    ;  true
    ).
```

```prolog
one_potentially_added(GoalList,DB) :-
    member(Literal,GoalList),
    potentially_added(Literal,DB).


/* --------------------------------------------------- */
/* Determining the literals that are potentially added */
/* --------------------------------------------------- */


/* verify if a literal is potentially added -
   without making any bindings and succeeding only once */
verify_potentially_added(Literal,DB) :-
    ( (potentially_added(Literal,DB) -> fail ; true)
    -> fail
    ;  true
    ).


potentially_added(neg(Atom),DB) :-
    potentially_deleted(pos(Atom),DB).
potentially_added(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(Atom,AddedFacts).
potentially_added(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(Atom)|NgBody]),AddedRules).
potentially_added(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(Atom)|NgBody]),ValidOldRules),
    member(BodyLiteral,NgBody),
    potentially_added(BodyLiteral,DB).


potentially_deleted(neg(Atom),DB) :-
    potentially_added(pos(Atom),DB).
potentially_deleted(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(Atom,DeletedFacts).
potentially_deleted(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(Atom)|NgBody]),DeletedRules).
potentially_deleted(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(Atom)|NgBody]),ValidOldRules),
    member(BodyLiteral,NgBody),
    potentially_deleted(BodyLiteral,DB).


/* ------------------------------------ */
```

```
/* non_ground_member(NgExpr,GrListOfExpr) */
/* ----------------------------------- */

non_ground_member(NgX,[GrH|GrT]) :-
    make_non_ground(GrH,NgX).
non_ground_member(NgX,[GrH|GrT]) :-
    non_ground_member(NgX,GrT).


/* ------------------------------------------------------ */
/*    make_non_ground(GroundRepOfExpr,NonGroundRepOfExpr)    */
/* ------------------------------------------------------ */
/* ex. ?-make_non_ground(pos(term(f,[var(1),var(2),var(1)])),X). */

make_non_ground(G,NG) :-
    mng(G,NG,[],Sub).

mng(var(N),X,[],[sub(N,X)]).
mng(var(N),X,[sub(M,Y)|T],[sub(M,Y)|T1]) :-
    ((N=M)
    -> (T1=T, X=Y)
    ;  (mng(var(N),X,T,T1))
    ).
mng(term(F,Args),term(F,IArgs),InSub,OutSub) :-
    l_mng(Args,IArgs,InSub,OutSub).
mng(not(G),not(NG),InSub,OutSub) :-
    mng(G,NG,InSub,OutSub).
mng(pos(G),pos(NG),InSub,OutSub) :-
    mng(G,NG,InSub,OutSub).

l_mng([],[],Sub,Sub).
l_mng([H|T],[IH|IT],InSub,OutSub) :-
    mng(H,IH,InSub,IntSub),
    l_mng(T,IT,IntSub,OutSub).


/* ------------------------------------------- */
/* SIMULATING THE DEDUCTIVE DATABASE FACT LOOKUP */
/* ------------------------------------------- */

db_fact_lookup(term(Pred,Args)) :-
    fact(Pred,Args).

fact(female,[term(mary,[])]).
fact(male,[term(peter,[])]).
fact(male,[term(paul,[])]).
...
```

B   The *ic-lst* Meta-Interpreter for the Benchmarks

This appendix contains the code of an implementation of the method by Lloyd, Topor and Sonenberg [53] for specialised integrity checking in deductive databases.

```
/* ================================================================= */
/* Bottom-Up Propagation of updates according to Lloyd et al's Method */
/* ================================================================= */

:- dynamic lts_rules/1.

construct_lts_rules :-
    retract(lts_rules(R)),fail.
construct_lts_rules :-
    findall(clause(Head,Body),rule(Head,Body),Rules),
    assert(lts_rules(Rules)).

lts_check(Nr,Update) :-
    lts_rules(Rules),
    check_ic(Nr,Update,Rules).

check_ic(Nr,Update,Rules) :-
    bup(Rules,Update,AllPos),!,
    member(false(Nr),AllPos),
    member(clause(false(Nr),Body),Rules),
    member(Atom,Body),
    member(Atom,AllPos),
    normal_solve(Body,Update).

/* This is the main Predicate */
/* Rules is the intensional part of the database */
/* Update are the added facts to the extensional database */
/* Pos is the set of (most general) atoms potentially affected by the update */
bup(Rules,Update,Pos) :-
    bup(Rules,Update,Update,Pos).

bup(Rules,Update,InPos,OutPos) :-
    bup_step(Rules,Update,[],NewPos,InPos,IntPos),
    ((NewPos=[])
    -> (OutPos=IntPos)
    ;  (bup(Rules,NewPos,IntPos,OutPos))
    ).

bup_step([],_Pos,NewPos,NewPos,AllPos,AllPos).
bup_step([Clause1|Rest],Pos,InNewPos,ResNewPos,InAllPos,ResAllPos) :-
    Clause1 = clause(Head,Body),
    bup_treat_clause(Head,Body,Pos,InNewPos,InNewPos1,InAllPos,InAllPos1),
    bup_step(Rest,Pos,InNewPos1,ResNewPos,InAllPos1,ResAllPos).

bup_treat_clause(Head,[],Pos,NewPos,NewPos,AllPos,AllPos).
bup_treat_clause(Head,[BodyAtom|Rest],Pos,InNewPos,OutNewPos,
                      InAllPos,OutAllPos) :-
    bup_treat_body_atom(Head,BodyAtom,Pos,InNewPos,InNewPos1,
                           InAllPos,InAllPos1),
    bup_treat_clause(Head,Rest,Pos,InNewPos1,OutNewPos,InAllPos1,OutAllPos).
```

```prolog
bup_treat_body_atom(Head,BodyAtom,[],NewPos,NewPos,AllPos,AllPos).
bup_treat_body_atom(Head,BodyAtom,[Pos1|Rest],InNewPos,OutNewPos,InAllPos,OutAllPos) :-
    copy(Pos1,Pos1C),
    copy(g(Head,BodyAtom),g(CHead,CBodyAtom)),
    (propagate_atom(CHead,CBodyAtom,Pos1C,NewHead)
    -> (add_atom(NewHead,InAllPos,InAllPos1,Answer),
        ((Answer=dont_add)
          -> (InNewPos2=InNewPos,InAllPos2=InAllPos1)
          ;  (add_atom(NewHead,InNewPos,InNewPos1,Answer2),
          ((Answer2=dont_add)
           -> (InNewPos2=InNewPos1,InAllPos2=InAllPos1)
           ;  (InNewPos2=[NewHead|InNewPos1],
               InAllPos2=[NewHead|InAllPos1]
               )
          )
         )
        )
       )
    ; (InNewPos2=InNewPos,InAllPos2=InAllPos)
    ),
    bup_treat_body_atom(Head,BodyAtom,Rest,InNewPos2,OutNewPos,
                        InAllPos2,OutAllPos).


propagate_atom(Head,BodyAtom,Pos,NewAtom) :-
    BodyAtom = Pos, !,
    NewAtom = Head.
propagate_atom(Head,BodyAtom,not(Pos),NewAtom) :- !,
    BodyAtom = Pos,
    NewAtom = not(Head).
propagate_atom(Head,not(BodyAtom),Pos,NewAtom) :- !,
    BodyAtom = Pos,
    NewAtom = not(Head).


add_atom(NewAtom,[],[],add).
add_atom(NewAtom,[Pos1|Rest],OutPos,Answer) :-
    (covered(NewAtom,Pos1)
    -> (OutPos = [Pos1|Rest],
        Answer=dont_add
       )
    ; (covered(Pos1,NewAtom)
        -> (OutPos=OutRest,
        add_atom(NewAtom,Rest,OutRest,Answer)
           )
        ; (OutPos=[Pos1|OutRest],
        add_atom(NewAtom,Rest,OutRest,Answer)
           )
      )
    ).
```

C A More Sophisticated Database

The following is the intensional part of a database adapted from [74] (where it is the most complicated database) and transformed into rule format (using Lloyd-Topor transformations [54] done by hand) required by [44].

```
parent(B,C) <- father(B,C)
parent(B,C) <- mother(B,C)

mother(B,C) <- father(D,C) & husband(D,B)

age(B,C) <- civil_status(B,C,D,E)

sex(B,C) <- civil_status(B,D,C,E)

dependent(B,C) <- parent(C,B) & occupation(C,service) & occupation(B,student)

occupation(B,C) <- civil_status(B,D,E,C)

eq(B,B) <-

aux_male_female(male) <-
aux_male_female(female) <-
aux_status(student) <-
aux_status(retired) <-
aux_status(business) <-
aux_status(service) <-
aux_limit(B,C) <- greater_than(B,0) & less_than(B,100000) &
                  greater_than(C,0) & less_than(C,125)

false(a1) <- civil_status(B,C,D,E) & civil_status(B,F,G,H) & ~eq(C,F)
false(a2) <- civil_status(B,C,D,E) & civil_status(B,F,G,H) & ~eq(D,G)
false(a3) <- civil_status(B,C,D,E) & civil_status(B,F,G,H) & ~eq(E,H)
false(2) <- father(B,C) & father(D,C) & ~eq(B,D)
false(3) <- husband(B,C) & husband(D,C) & ~eq(B,D)
false(4) <- husband(B,C) & husband(B,D) & ~eq(C,D)
false(5) <- civil_status(B,C,D,E) & aux_male_female(B) &
            aux_status(E) & ~aux_limit(B,C)
false(6) <- civil_status(B,C,D,student) & ~less_than(C,25)
false(7) <- civil_status(B,C,D,retired) & ~greater_than(C,60)
false(8) <- father(B,C) & ~sex(B,male)
false(9a) <- husband(B,C) & ~sex(B,male)
false(9b) <- husband(B,C) & ~sex(C,female)
false(10a) <- husband(B,C) & age(B,D) & ~greater_than(D,19)
false(10b) <- husband(B,C) & age(C,D) & ~greater_than(D,19)
false(11) <- civil_status(B,C,D,E) & less_than(C,20) & ~eq(E,student)
```