Towards 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
{michael,dannyd}@cs.kuleuven.ac.be

Abstract

In [23] we presented a partial evaluation scheme for a "real life" subset of Prolog, containing first-order built-in's, simple side-effects and the operational predicate if-then-else. In this paper we apply this scheme to specialise integrity checking in deductive databases. We present an interpreter which can be used to check the integrity constraints in hierarchical deductive databases. This 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 we are able to obtain very efficient specialised update procedures, executing substantially faster than the original meta-interpreter. The partial evaluation scheme presented in [23] seems to be capable of automatically generating highly specialised update procedures for deductive databases.

1 Introduction

Partial evaluation has received considerable attention both in functional programming (see the book by Jones *et al* [20] and the references therein) and logic programming (e.g. [14, 15, 21, 37]). 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 very little 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 [25].

However pure logic programming is rarely considered to be viable for practical "real-life" programming and for instance Prolog incorporates non-declarative extensions. In [23] we presented a partial evaluation scheme for a practically usable subset of Prolog encompassing first-order[2] built-in's, like **var/1**, **nonvar/1** and **=../2**, simple side-effects, like **print/1**, and the operational if-then-else construct.

An important aspect is the inclusion of the if-then-else, which proved to be much better suited for partial evaluation than the "full blown" cut. For instance it was possible to obtain a Knuth-Morris-Pratt like search algorithm by specialising a "dumb" search algorithm for a given pattern. The if-then-else contains a local cut and is usually written as **(If -> Then ; Else)**. The following informal Prolog clauses can be used to define the if-then-else:

```
(If->Then;Else) :- If,!,Then.
(If->Then;Else) :- Else.
```

In other words if the test-part succeeds then a local cut is executed and the then-part is entered (this means that the test-part will yield at most one solution). If the test-part fails finitely then the else-part is executed and if the test-part "loops" (i.e. fails infinitely) then the whole construct loops. Note that most uses of the cut can be mapped to if-then-else constructs and the if-then-else can also be used to implement the **not/1**.[3]

In [23] we also showed that freeness and sharing information, which so far have been of no interest in (pure) partial deduction, can be important to produce efficient specialised programs. In this paper we apply the partial evaluation technique of [23] 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 the specification of deductive databases. 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 effort has been put into improving integrity checking such that it takes advantage of the fact that a database was consistent before any particular update

*Supported by Esprit BR-project Compulog II

†Senior research associate of the Belgian National Fund for Scientific Research

[1] Some notable exceptions are [16, 31].

[2] As opposed to "second order" built-in's which are predicates manipulating clauses and goals, like call/1 or assert/1.

[3] Both the unsound and the sound version (using a groundness check for soundness).

and only verifies the relevant parts of a database. Some references to this line of research are [4, 6, 8, 10, 26, 27, 36].

Some techniques also address pre-compilation aspects and some even explicitly generate ***specialised update procedures*** for certain update patterns and partial descriptions of the database (see for instance the approach by Wallace in [38]). The techniques are however restricted to very specific kinds of updates and specific kinds of partial knowledge. Usually the intensional database (i.e. the rules) and the integrity constraints are supposed to be fixed and known, the extensional database (i.e. the facts) is considered to be totally unknown and only updates to the extensional database are considered. This is for instance the case for the approach by Wallace in [38].

A ***meta-program*** is a program which takes another program, the ***object-program***, as input and manipulates it in some way. Some of the applications of meta-programming are (a much more detailed account can be found in [18]): extending the programming language, debugging, program analysis, program transformation and of course specialised integrity checking. In the latter case the object program is the (relevant) part of 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 [5], 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 wrt testing phantomness and idleness.[4]

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 [23] to a particular meta-program performing integrity checking in (hierarchical) deductive databases. Our results show 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 we present a method which performs specialised integrity checking. In section 3 we present a meta-interpreter which implements this method and in section 4 we discuss how this meta-interpreter can be unfolded by a partial evaluator. Section 5 discusses some implementation details of the meta-interpreter and in section 6 we generate specialised update procedures through partial evaluation and present experimental results. Some concluding remarks can be found in section 7.

[4] See for instance the survey by Bry, Manthey and Martens in [6]. A brief description is also given in this paper after definition 2.3.

2 Basic Definitions

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 [24]. We use the convention to represent logical variables by (specially typeset) uppercase letters like $\mathtt{X}, \mathtt{Y}$. Predicates and functors will be represented by lowercase letters like p, q, f, g.

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

Note that we do not require a deductive database to be range-restricted. Range-restriction is not necessary for the approach in this paper[5] and would only burden the presentation. Also 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*** $\leftarrow$ *Body*. A database is said to be ***inconsistent***, or violating the integrity constraints, iff ***false*** is derivable in the database via SLDNF (which is a proof procedure for normal logic programs which handles "negation by failure", for further details see [1] or [24]).

For the simplicity of the presentation we also suppose that all clauses occuring in deductive databases are in some ***normal form***, meaning that if two clauses are variants of each other then they are syntactically identical.[6] We also take the liberty to not always explicitly cite the goal for which a SLDNF derivation is made. Finally we will suppose that any SLDNF-derivation may 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.

Definition 2.2 (Database Update)
A ***database update*** 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 $Db^- \cup Db^=$.

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.

In the following definition we 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 [26] (which in turn is an extension of the calculation of $atom_{D,D'}$ by Lloyd and Topor in [27]). The main difference being that we calculate $pos(U)$ and $neg(U)$ in one step instead of in two, which should be more efficient (however the result is the same as in each iteration step the

[5] This is because our notion of integrity is based on the SLDNF procedure, which always gives the same answer irrespective of the underlying language. However range-restriction is still useful as it ensures that no SLDNF refutation will flounder.

[6] This guarantees that a database does not contain rules which are variants of each other and it also guarantees that no composition of databases will contain rules which are variants of each other.

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 [26, 27]).

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:

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

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

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

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 $neg(U)$. As such the definition does not need to access the entire database, and in fact the above definition does not reference the set of facts in $Db^=$ (only clauses with at least one literal in the body are used). This somewhat restricts the usefulness of this method (and the one in [26, 27]) when rules and integrity constraints change more often than facts.[7]

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

$mother(X, Y) \leftarrow parent(X, Y), woman(X)$
$father(X, Y) \leftarrow parent(X, Y), man(X)$
$false \leftarrow man(X), woman(X)$
$false \leftarrow parent(X, Y), parent(Y, X)$

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:

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

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 [26] simplifies the integrity constraints by calculating $\Theta_U^+(\leftarrow Body_i)$ for each body $Body_i$ of an integrity constraint and instantiating the integrity constraints using the so obtained set of substitutions.[8] For the example 2.4 above we obtain:

$$\Theta_U^+(\leftarrow man(X), woman(X)) = \{\{X/a\}\} \text{ and}$$
$$\Theta_U^+(\leftarrow parent(X, Y), parent(Y, X)) = \emptyset$$

and thus obtain the following set of specialised integrity constraints:

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

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.

Note that 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.
The proof can be found in appendix A.

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 SLDNF sub-derivation leading from G_0 to G_i.

[7] Addressing this limitation in the context of this paper is subject of ongoing research.

[8] Note however that in [26] integrity constraints are closed typed first order formulas and that a database is inconsistent if the integrity constraints are *not* a logical consequence (of the completion of the database).

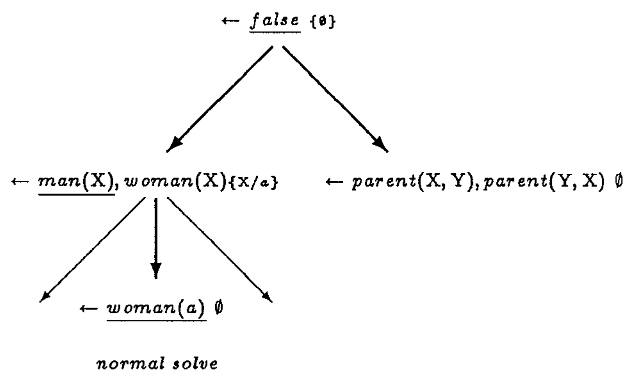
A refutation being a particular derivation we can specialise the concept and define ***relevant refutations***. The following theorem will form the basis of our method for performing specialised integrity checking.

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 $G_0 = \leftarrow false, G_1, \ldots, G_k = \Box$ be the incremental refutation. For each G_i we either have that G_i occurs after the first incremental derivation step and hence δ_i is incremental (where δ_i is defined as in definition 2.8). 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. $\Box$

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 set 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.



$\leftarrow \underline{false}$ $\{\emptyset\}$
$\leftarrow \underline{man(X)}, woman(X)\{X/a\}$
$\leftarrow parent(X, Y), parent(Y, X)$ $\emptyset$
$\leftarrow \underline{woman(a)}$ $\emptyset$
normal solve

Figure 1: SLDNF tree for example 2.4

Note that the above method can be seen as an extension of the method in [26] 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 can be 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 [26] not only removes integrity constraints but also instantiates them (possibly generating several specialised integrity constraints for a single unspecialised one). This is 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 and definition 2.8 could actually be easily adapted to use Θ_U^+ for instantiating goals and theorem 2.9 would still be valid. Also, surprisingly, the instantiations will often be performed by the partial evaluation method itself and the results in section 6 illustrate this. We will elaborate on these aspects in section 5.

3 The Meta-interpreter

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 we will discuss the issue of how these object level expressions will be represented at the meta-level (in other words inside the meta-interpreter).

In logic programming there are basically two opposing schools of thought on how an object level expression, say the atom $p(X, a)$, should be represented at the meta-level. The first school would use the term $p(X, a)$ 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.

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-in's. The non-ground representation also has semantical problems (although they were solved to some extent in [9,29,30] by De Schreye and Martens). 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 [18] or to the conclusion of [29]. Also the programming language Gödel provides extensive support for the ground representation and further details and discussions can be found in the book on Gödel [19].

Sometimes however it is possible to combine both approaches into one. This was first exemplified by Gallagher in [13, 14] where an interpreter for the ground representation is presented which lifts the ground representation to the non-ground one for resolution. A similar technique was put to good use in the self-applicable partial evaluator Logimix by Mogensen and Bondorf [20, 31]. Hill and Gallagher [18] also provide a recent account of this style of writing meta-

Object level	Ground representation
X	*var*(1)
c	*struct*(*c*, [])
f(X, *a*)	*struct*(*f*, [*var*(1), *struct*(*a*, [])])
p ← *q*	*struct*(*clause*, [*struct*(*p*, []), *struct*(*q*, [])])

Figure 2: A ground representation

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 [13] 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 (a promising attempt is the partial evaluator SAGE for Gödel, see [3, 16, 17]).

We could try to obtain a similar result by partially evaluating a meta-interpreter for the non-ground representation. There is however one caveat: the object-program has to be stored explicitly using meta-program clauses instead of using a term-representation in the meta-program (unless we use non-logical built-in's like *copy*/2 to perform the standardising apart). This has two major disadvantages. Firstly 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[9] impossible. Secondly 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 constant. In the ground representation the user can express this 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.[10]

So our meta-interpreter is an adapted version of the interpreter presented by Gallagher in [13] and includes a predicate *make_non_ground*/2 which lifts a ground term to a non-ground one. For instance the query

← *make_non_ground*(*struct*(*f*, [*var*(1), *var*(2), *var*(1)]), X)

succeeds with a computed answer similar[11] to

{X/*struct*(*f*, [_49, _57, _49])}.

The code for this predicate is presented in figure 3 and a simple meta-interpreter based on it can be found in figure 4. Note that the first argument to *make_non_ground* will always be ground. This means that the if-then-else construct in the second clause for *mng*/1 behaves like a completely declarative if-then-else (like the one in Gödel [19]).

[9] See for instance [18].

[10] 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).

[11] The variables _49 and _57 are fresh variables. Their actual names may vary and are not important.

```
make_non_ground(GrTerm, NgTerm) ←
    mng(GrTerm, NgTerm, [], _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(struct(F, GrArgs), struct(F, NgArgs), InSub, OutSub) ←
    l_mng(GrArgs, NgArgs, InSub, OutSub)

l_mng([], [], Sub, Sub) ←
l_mng([GrH|GrT], [NgH|NgT], InSub, OutSub) ←
    mng(GrH, NgH, InSub, InSub1),
    l_mng(GrT, NgT, InSub1, OutSub)
```

Figure 3: Lifting the ground representation

```
solve(Prog, []) ←
solve(Prog, [H|T]) ←
    non_ground_member(struct(clause, [H|Body]), Prog),
    solve(Prog, Body),
    solve(Prog, T)

non_ground_member(NonGrTerm, [GrH|GrT]) ←
    make_non_ground(GrH, NonGrTerm)
non_ground_member(NonGrTerm, [GrH|GrT]) ←
    non_ground_member(NonGrTerm, GrT)
```

Figure 4: An interpreter for the ground representation

We can now use theorem 2.9 to extend the interpreter in figure 4 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.

The skeleton of our meta-interpreter in figure 5 implements this idea. Note that the argument Updt stands for the ground representation of the update $\langle Db^+, Db^=, Db^- \rangle$. Also note that from now on "$\mathcal{T}$" denotes the ground representation of the 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.

```
incremental_solve(Updt, Goal) ←
    potentially_added(Updt, Goal),
    resolve(Updt, Goal)

resolve(Updt, Goal) ←
    resolve_unincrementally(Updt, Goal, NewGoal),
    incremental_solve(Updt, NewGoal)
resolve(Updt, Goal) ←
    resolve_incrementally(Updt, Goal, NewGoal),
    Updt = "⟨Db+, Db=, Db-⟩",
    solve("Db= ∪ Db+", NewGoal)
```

Figure 5: Skeleton of the integrity checker

Specialised integrity checking now consists in calling

$$\leftarrow \textit{incremental\_solve}\,(\text{“}\langle Db^{+}, Db^{=}, Db^{-}\rangle\text{”}, \leftarrow \textit{false})$$

The query will succeed if the integrity of the database has been violated by the update. In the next section we will examine how this meta-interpreter can be unfolded by a partial evaluator and in section 5 we will study the implementation of the predicate *potentially_added/2*.

4 Unfolding the Meta-interpreter

In this section we will examine how the meta-interpreter of figure 5 can be unfolded by a partial evaluator. The discussions are also valid for the simpler meta-interpreter of figure 4. Some of the problems discussed in this section are of less relevance for hierarchical object programs but we should not forget that our final goal is to move to recursive databases.

Unfolding a meta-interpreter in a satisfactory way is a non-trivial issue and in general 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[12] inside the partial evaluator can quite often be sufficient to guarantee termination when unfolding such a meta-interpreter. This is illustrated in figure 6, where Prog represents an object program inside of which the predicate $p/1$ is recursive via $q/1$. Note that intermediate goals have been removed for clarity. 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 generate good specialised code.[13] However if we unfold the meta-interpreter of figure 4 the variant test is sufficient to detect the loop and no abstraction is needed to generate efficient specialised code. Note that 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 (or whether it is written in another language).

A ground solve	*Non-ground solve of figure 4*
$solve(\text{Prog}, [struct(p, [var(1)])])$	$solve(\text{Prog}, [p(\_1)])$
$\downarrow$	$\downarrow$
$solve(\text{Prog}, [struct(q, [var(3)])])$	$solve(\text{Prog}, [q(\_3)])$
$\downarrow$	$\downarrow$
$solve(\text{Prog}, [struct(p, [var(3)])])$	$solve(\text{Prog}, [p(\_3)])$

Figure 6: 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. We will now give some details of this implementation (which is based on the first author's Master's thesis [22], further refined in [23]). The partial evaluation process 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, see [23]) and generates the residual program.

Such a decomposition has already proven to be useful for self-application in the world of functional programming (see for instance the book by Jones *et al* [20]). Also the partial evaluator Logimix by Mogensen and Bondorf [31] (also in [20]) uses such an approach.

Unfortunately the annotation phase is not yet automatic and must usually be performed by hand. This gives the knowledgeable user very precise control over the unfolding, especially since some quite sophisticated annotations can be provided. The annotations allow the user to give conditions on

1. when a literal 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 literal should be left untouched).

For instance the user can specify that the literal $var(X)$ 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. Note that the filters in the partial evaluator Schism for applicative languages (see [7]) also allow for conditions on when to unfold (U) and when to residualise (R). They are however used in an off-line fashion and also control the binding-time analysis.

The annotations, if designed properly, are independent of the query for which the program is to be specialised. In our case this means that the annotations have to be created only once for the incremental integrity checker. After that the second and third phases will be able to derive ***fully automatically*** specialised update procedures.

Since the annotation only needs to be done once (for each meta-interpreter), there is no objection against producing it by hand. So the hand made annotations, in conjunction with the fact that the variant test can be used for the non-ground representation, are the crucial aspects to be able to perfectly unfold our meta-interpreter. We have not yet addressed the problem of performing fully automatic unfolding of meta-interpreters in general. This problem is non-trivial and a promising approach is presented by Martens in [28]. In future, we hope to combine both lines of work into a running system which is able to fully automatically partially evaluate meta-interpreters in a satisfactory way.

[12] 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.

[13] 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 $solve(\text{Prog}, struct(p, [var(1)]))$ cannot be used for $solve(\text{Prog}, struct(p, [var(3)]))$, 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.

The issue of adequately unfolding our meta-interpreter for integrity checking being solved, there remains only one problem: namely how to implement *potentially_added* such that it can effectively be partially evaluated. This turns out to be non-trivial as well and in the next section we propose a solution for hierarchical databases.

5 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. 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) solutions and which might instantiate the goal under consideration (because the goal is in the non-ground representation). This means that ideally we would either have to perform an expensive subsumption test (which is very hard to partially evaluate satisfactorily) to retain only the most general solutions or backtrack and try out a lot of useless instantiations.[14] Also in the case of recursive databases this approach will unavoidably lead to problems due to non-termination. We will however not tackle recursive databases in this paper and only discuss some issues in section 7. Let us illustrate the problem through an example.

Example 5.1
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

$$\leftarrow potentially\_added(\leftarrow false, \text{“U”})$$

and twice for the query

$$\leftarrow potentially\_added(\leftarrow father(\mathrm{X},\mathrm{Y}), \text{“U”})$$

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

The above example shows that a naive top-down implementation 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 5.1 above, both

$$\leftarrow verify(potentially\_added(\leftarrow false, \text{“U”}))$$
$$\leftarrow verify(potentially\_added(\leftarrow father(\mathrm{X},\mathrm{Y}), \text{“U”}))$$

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

will succeed just once with the empty computed answer and no backtracking is required. The $verify(.)$ primitive can be implemented with the if-then-else construct in the following way: `((Goal->fail;true)->fail;true)`. Thus we can use the partial evaluation method of [23] which incorporates extensive support for the if-then-else.

The disadvantage of using $verify$ is of course that no instantiations are performed (which in general cut down the search space dramatically). However these instantiations can often be performed by the partial deduction method through pruning and safe left- and right-propagation of bindings. Take for instance a look at the specialised update procedure presented in figure 9 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 5 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 following section. In a sense, part of the specialised integrity checking is performed by the meta-interpreter and part is performed by the partial evaluator. Also note that the analysis performed by the partial evaluator used in the experiments 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 "determinate" parts of the bindings are propagated meaning that no additional choice-points are generated. This can only be beneficial but is not always optimal.

6 Results

Throughout the remainder of this section the rules in figure 7 form the intensional part of $Db^=$ (the facts are unknown at partial evaluation time).

$$\boxed{\begin{array}{l}
mother(\mathrm{X},\mathrm{Y}) \leftarrow \\
\quad parent(\mathrm{X},\mathrm{Y}), woman(\mathrm{X}) \\
father(\mathrm{X},\mathrm{Y}) \leftarrow \\
\quad parent(\mathrm{X},\mathrm{Y}), man(\mathrm{X}) \\
grandparent(\mathrm{X},\mathrm{Z}) \leftarrow \\
\quad parent(\mathrm{X},\mathrm{Y}), parent(\mathrm{Y},\mathrm{Z}), \\
married\_to(\mathrm{X},\mathrm{Y}) \leftarrow \\
\quad parent(\mathrm{X},\mathrm{Z}), parent(\mathrm{Y},\mathrm{Z}), \\
\quad man(\mathrm{X}), woman(\mathrm{Y}) \\
married\_man(\mathrm{X}) \leftarrow \\
\quad married\_to(\mathrm{X},\mathrm{Y}) \\
married\_woman(\mathrm{X}) \leftarrow \\
\quad married\_to(\mathrm{Y},\mathrm{X}) \\
unmarried(\mathrm{X}) \leftarrow \\
\quad man(\mathrm{X}), \neg married\_man(\mathrm{X}) \\
unmarried(\mathrm{X}) \leftarrow \\
\quad woman(\mathrm{X}), \neg married\_woman(\mathrm{X}) \\
\\
false \leftarrow \\
\quad man(\mathrm{X}), woman(\mathrm{X}) \\
false \leftarrow \\
\quad parent(\mathrm{X},\mathrm{Y}), parent(\mathrm{Y},\mathrm{X}) \\
false \leftarrow \\
\quad parent(\mathrm{X},\mathrm{Y}), unmarried(\mathrm{X})
\end{array}}$$

Figure 7: Intensional part of $Db^=$

Example 6.1
Before showing the results of our method, let us first illustrate in what sense it improves upon the method of Lloyd *et al* in [26]. Given $Db^+ = \{man(a) \leftarrow\}$, $Db^- = \emptyset$, we have that (independent of the facts in $Db^=$):

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

The method of [26] would thus generate the following specialised integrity constraints:

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

This is clearly not optimal (given the available information), as is illustrated in figure 8 by the incomplete SLDNF tree for the second specialised integrity constraint. Suppose that some fact matching $parent(a, \mathrm{Y})$ exists in the database. Evaluating the specialised integrity constraints obtained by [26] will then yield 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 (unless the database was already inconsistent before the update). Our meta-interpreter improves upon this and will never evaluate the goal:

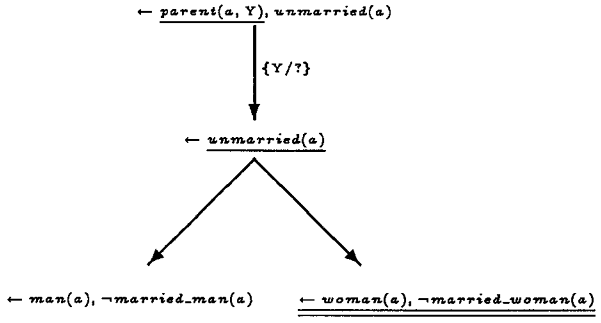
$$\leftarrow woman(a), \neg married\_woman(a)$$

In fact our method will prune this useless branch already at partial evaluation time, for instance when generating a specialised update procedure for the update pattern $Db^+ = \{man(\mathcal{A}) \leftarrow\}$, $Db^- = \emptyset$, where $\mathcal{A}$ is unknown at partial evaluation time. This is one of the experiments shown below. A slightly sugared and simplified version of the resulting update procedure is presented in figure 9. 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 (this is usually the case but it is not required by the current method, which explains the test in figure 9 whether there is a fact *married_to*). Note that the benchmarks were executed on the un-sugared and un-simplified version but that there is no problem whatsoever, apart from finding the time for coding, to directly produce the sugared and simplified version.

The following are some of the experiments that have been conducted. Times are in seconds and were obtained by calling the *time*/2 predicate of Prolog by BIM (which incorporates the time needed for garbage collection, see [35]) using 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 7. 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.

We have also included two well known and publicly available partial evaluators. The players in the benchmark game are:

1. **solve:** This is the naive meta-interpreter of figure 4. It does not use the fact that the database was consistent before the update and simply tries to find a refutation for $\leftarrow false$.



← parent(a, Y), unmarried(a)
{Y/?}
← unmarried(a)
← man(a), ¬married_man(a)
← woman(a), ¬married_woman(a)

Figure 8: 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 9: Specialised update procedure for adding $man(\mathcal{A})$

2. **ic-solve:** This is the meta-interpreter performing specialised integrity checking described in section 3. The skeleton of the meta-interpreter can be found in figure 5.

3. **leupel:** This is the partial evaluation system based on work reported in [22, 23].

4. **leupel$^-$:** This is the above system where the safe left-propagation of bindings has been disabled. It is included to show the positive effect of left-propagating bindings.

5. **Mixtus:** This is the automatic partial evaluation system for full Prolog presented in [37].

6. **Paddy:** This is the automatic partial evaluation system for full Sepia (a variant of Prolog) presented in [32]. Some aspects of the system are also described in [33, 34]. The resulting specialised programs had to be converted for Prolog by BIM (**get_cut/1** to **mark/1** and **cut_to/1** to **cut/1**).

The first experiment we present consists in generating an update procedure for the update pattern (where $\mathcal{A}$ is unknown at partial evaluation time):

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

The result of the partial evaluation obained by leupel can be seen in figure 9 and the timings are summarised in the following table 1. The first row of figures contains the absolute and relative time for a database with 108 facts and the second row contains the corresponding figures for 216 facts.

solve	ic-solve	leupel	leupel$^-$	Mixtus	Paddy
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
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 $Db^+ = \{man(\mathcal{A}) \leftarrow\}$, $Db^- = \emptyset$

solve	ic-solve	leupel	leupel$^-$	Mixtus	Paddy
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
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 $Db^+ = \{parent(\mathcal{A}, \mathcal{B}) \leftarrow\}$, $Db^- = \emptyset$

Here are some small additional remarks. For Paddy we had to increase the "term_depth" parameter from its default value. With the default value the integrity checking took 9.51 s (126.8 relative) for 108 facts (i.e. a slow down of 1.27 over ic-solve). Mixtus needed no adjustments.

The time needed to obtain the leupel specialisation was 78.19 s, meaning that the time invested into partial evaluation will pay off rather quickly for larger databases (given that the meta-interpreter was specialised for a full specification of the rules and integrity constraints in $Db^=$, the update procedures only have to be re-generated when the rules change[15]). Note that the current implementation of leupel has a very slow post-processor, displays tracing information and uses the ground representation. It should be possible to dramatically cut down the time needed for partial evaluation. Leupel seemed however still to be faster than Mixtus and almost half as fast as Paddy.[16]

To give an idea about the performance of the Lloyd *et al* method of [26] relative to our specialised update procedures we have done a quick and dirty non-ground implementation of the method without handling negation (i.e. the implementation will in some cases not test negative literals although it should; for this experiment it is not relevant but for the next one it would). The time needed to perform the integrity checking for 108 facts was 1.35 s (18.0 times slower than the specialised update procedure generated by leupel). After specialising this program for the update pattern using Mixtus we obtained a small speedup and an execution time of still 1.28 s for 108 facts.

Another experiment consisted in generating a specialised update procedure for the following update pattern (where $\mathcal{A}$ and $\mathcal{B}$ are unknown at partial evaluation time):

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

This update offers less opportunities for specialisation than the previous update and the speedup figures are still satisfactory but less spectacular. The results are summarised in the following table 2.

To conclude this section we can say that the speedup figures obtained with leupel 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 (even for only 216 facts, this 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 4 usually yields speedups reaching at most 1 order of magnitude.

[15]As pointed out by Bern Martens, a technique based on work by Benkerimi and Shepherdson [2] could be used to incrementally adapt the specialised update procedure whenever the rules or integrity constraints in $Db^=$ change.

[16]Exact comparisons where not made because Mixtus runs under Sicstus Prolog, Paddy under Eclipse and leupel under Prolog by BIM.

7 Conclusion

The integrity checking experiments have been carried out on an interpreter for the ground representation based on the one by Gallagher in [13]. This interpreter lifts the ground representation to the non-ground one for resolution yielding good speeds and simplifies the control of unfolding. The interpreter has been extended to incorporate the knowledge that prior to the update the integrity constraints were not violated. This is accomplished by using a *verify* primitive implemented via the if-then-else construct. Thus the partial evaluation scheme described in [23], which is able to specialise the if-then-else, turned out to be practically useful. This *verify* construct is now used to test whether a given goal, encountered while checking the integrity of a database after an update, might be influenced by that update. If this is not the case the integrity checker will stop the derivation of the goal.

For simplicity we have restricted ourselves to normal, hierarchical databases in our experiments. Perfect unfolding of the meta-interpreter for integrity checking is obtained by a combination of the variant test and by, once only, handmade annotations of the meta-interpreter.

The results of the initial experiments were very encouraging, with speedups reaching 2 orders of magnitude when specialising the integrity checker for a given set of integrity constraints and a given set of rules (the speedups are of course much higher when compared with a non-incremental solve which re-checks the entire database). These high speedups are also due to the fact that the partial evaluator performs part of the integrity checking.

To summarise, it seems that partial evaluation is capable of automatically generating highly specialised update procedures. Based on these encouraging result we conjecture that self-applicable partial evaluation can be extremely useful for optimising integrity checks in deductive databases and for generating update procedure compilers.

Work in this direction is ongoing and we will examine 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. Ongoing research is also concerned with extending the results of this paper to recursive, stratified databases. The added complications are that in a top-down method a loop check has to be incorporated into *potentially_added*. This loop check is non-declarative for the non-ground representation and is difficult to partially evaluate satisfactorily. Similarly a bottom-up method also needs a loop check and also seems to be difficult to partially evaluate effectively.

Finally it is also 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 4. In that case self-applicable partial evaluation could be used to obtain specialised update procedures (by performing the second Futamura projection, [11, 12]) and update procedure compilers (by performing the third Futamura projection).

Acknowledgements

We would like to thank Bern Martens for proof-reading several versions of this paper and for his helpful insights and comments on the topic of this paper. We would also like to thank him for his huge pile of references on integrity checking and for introducing the first author to the subject. Our thanks also go to John Gallagher for pointing out several errors and for the fruitful discussions. Finally we would like to thank anonymous referees for their useful remarks.

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.

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

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

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

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

[23] M. Leuschel. Partial evaluation of the "real thing". In Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'94*, Pisa, Italy, 1994. To be published.

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

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

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

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

[28] 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.

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

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

[31] 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.

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

[33] 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.

[34] 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.

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

[36] 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.

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

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

A Proof for Lemma 2.7

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[17] 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.

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 $1 \leq m \leq k-1$, 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: $G_{m-1} = \leftarrow L'_1, \ldots, L'_j, \ldots, L'_r$ and $G_m = \leftarrow (L'_1, \ldots, L'_{j-1}, B_1, \ldots, B_q, L'_{j+1}, \ldots, L'_r)\theta$.

There are again two cases. Either there exists a L'_p, with $1 \leq p \leq r \wedge p \neq j$, 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.[18]

In the other case there only exists a B_p, with $1 \leq p \leq q$, 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 calcualation 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 $G = G_0$. □

[17] So far we have not provided a formal definition of the notion of "standardising apart" (several ones, correct and incorrect, exist in the literature). Just suppose for the remainder of this proof that fresh variables, not occurring "anywhere else", are used.

[18] As already pointed out this has been overlooked in [26, 27]. Take for instance $L'_p = p(a, Y, b, X)$ and $\theta = \{X/Y, Y/X\}$. Then $L'_p\theta$ unifies with $p(X, a, Y, b) \in pos(U)$ and the more general L'_p does not!