Preserving Termination of Tabled Logic Programs While Unfolding

Michael Leuschel, Bern Martens and Konstantinos Sagonas

K.U. Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: {michael,bern,kostis}@cs.kuleuven.ac.be

Abstract. We provide a first investigation of the specialisation and transformation of tabled logic programs through unfolding. We show that — surprisingly — unfolding, even determinate, can worsen the termination behaviour in the context of tabling. We therefore establish two criteria which ensure that such mishaps are avoided. We also briefly discuss the influence of some other transformation techniques on the termination and efficiency of tabled logic programs.

1 Introduction

The use of tabling in logic programming is beginning to emerge as a powerful evaluation technique, since it allows bottom-up evaluation to be incorporated within a top-down framework, combining the advantages of both. Although the concept of tabled execution of logic programs has been around for more than a decade (see [27]), practical systems based on tabling are only beginning to appear. Early experience with these systems suggests that they are indeed practically viable. In particular the XSB system [24], based on SLG-resolution [3], computes in-memory queries about an order of magnitude faster than current semi-naive methods, and evaluates Prolog queries with little reduction in performance when compared to well-known commercial Prolog systems.

At a high level, top-down tabling systems evaluate programs by recording subgoals (referred to as *calls*) and their provable instances (referred to as *answers*) in a table. Predicates are designated *a priori* as either *tabled* or *nontabled*. Clause resolution, which is the basic mechanism for program evaluation, proceeds as follows. For nontabled predicates the call is resolved against program clauses. For tabled predicates, if the call is new to the evaluation, it is entered in the table and Prolog-style program clause resolution is used to compute its answers which are also recorded in the table. If, on the other hand, a variant[1] of the call is already present in the table, then it is resolved against its recorded answers. By using answer tables for resolving subsequent invocations of the same call, tabled

[1] Tabling evaluation methods can be based either on variant checks, as SLG-resolution is, or on subsumption checks. Throughout this paper, unless otherwise specified, we assume tabling based on variance, and we refer the reader to [3] Section 7.1 for a discussion on some of the issues that are involved in this choice.

evaluation strategies prevent many cases of infinite looping which normally occur in Prolog-style SLD evaluation. As a result, termination characteristics of tabling-based logic programming systems are substantially better than those of Prolog [3].

Given the relative novelty of tabling-based implementations, many promising avenues for substantially improving the performance of tabled programs remain to be explored. Research in this topic has mainly addressed issues related to finding efficient data structures for tabling [22], or suggesting low-level modifications of the SLG-WAM [23]. In this paper we deviate from this path and investigate issues related to the optimisation of tabled programs using more portable techniques such as specialisation through unfolding or similar program transformations.

Program transformation is by now a widely accepted technique for the systematic development of correct and efficient programs. Given a program, the aim of program transformation is to produce a more efficient program which solves the same problem, that is, which is equivalent in meaning to the original one under a semantics of choice. Various systems for program transformation have been developed, usually based on the use of the fold/unfold framework. This framework dates back to at least [2], has been introduced to the logic programming community in a seminal paper of Tamaki and Sato [26], has since been the subject of considerable research (see e.g. the references in [18]), and has been successfully used in many partial evaluators for Prolog-style execution [20, 19, 25, 8]. Unfortunately, no methodology for the transformation or specialisation of tabled logic programs exists. All techniques stay within the context of untabled execution. Initially, one may expect that results established in the "classic" (S)LD setting more or less carry over. This, however, turns out to be far from obviously true as the differences between the execution models are significant.

In this paper, we mainly concentrate on issues related to the safety of unfolding in tabled logic programs. We do so because in the context of program specialisation, unfolding is the most important ingredient. For instance, partial deduction [17] basically only employs unfolding. In untabled execution of logic programs unfolding is not problematic. For example, it preserves both the least Herbrand model and set of computed answer substitutions semantics and — even in the context of the unfair Prolog selection rule — it *cannot* worsen the (universal) termination behaviour of a program [21]. Under tabled execution, however, unfolding — even determinate — may transform a terminating program into a non-terminating one ! Naturally, this is a situation that better be avoided.

To reason about unfolding of tabled logic programs, we describe a framework that captures their termination (under the left-to-right selection rule) and define applicability conditions that ensure the intended equivalence property between the original program and the transformed one. Using this framework we prove that certain non-trivial and commonly used in practice types of unfolding are safe with respect to termination.

In summary, our results regarding unfolding in the context of tabled execution are as follows:

– We prove that left-most unfolding or unfolding without any left-propagation of bindings preserves termination of tabled logic programs.
– We show that even though left-propagation of bindings through unfolding can worsen the termination characteristics of a tabled programs, left-propagation of grounding substitutions is safe wrt termination.

The rest of the paper is organised as follows. In the next section we introduce some preliminaries and in Section 3 we show through an example how unfolding endangers termination of tabled programs. To reason about preservation of termination by unfolding, in Section 4 we introduce the notion of *quasi-termination* of tabled programs, and based on this notion in Section 5 we prove the above results. We end with an extended discussion of the effect that some other commonly used transformation techniques have on the termination and efficiency of tabled programs.

2 Preliminaries

We denote by B_P^E the non-ground extended Herbrand base[2] (i.e. the set of atoms modulo the variant equivalence relation $\approx$ as defined in [7]). We also define the following notations: the set of variables occurring inside an expression E is denoted by $vars(E)$, the *domain* of a substitution θ is defined as $dom(\theta) = \{X \mid X/t \in \theta\}$ and the *range* of θ is defined as $ran(\theta) = \{Y \mid X/t \in \theta \wedge Y \in vars(t)\}$. Finally, we also define $vars(\theta) = dom(\theta) \cup ran(\theta)$ as well as the restriction $\theta|_{\mathcal{V}}$ of a substitution θ to a set of variables $\mathcal{V}$ by $\theta|_{\mathcal{V}} = \{X/t \mid X/t \in \theta \wedge X \in \mathcal{V}\}$. By $mgu(A, B)$ we denote a substitution θ which is an idempotent (i.e. $\theta\theta = \theta$) and relevant (i.e. $vars(\theta) \subseteq vars(A) \cup vars(B)$) most general unifier of two expressions A and B. In the remainder of this paper we will use the notations $hd(C), bd(C)$ to refer to the head and the body of a clause C respectively.

A program transformation process starting from an initial program P_0 is a sequence of programs $P_0, \ldots, P_n$, called a *transformation sequence*, such that program P_{k+1}, with $0 \leq k < n$, is obtained from P_k by the application of a *transformation rule*, which may depend on $P_0, \ldots, P_k$. Let us now formally define unfolding, slightly adapted from [18, (R1)].

Definition 1. (Unfolding rule) Let P_k contain the clause $C = H \leftarrow F, A, G$, where A is a positive literal and where F and G are (possibly empty) conjunctions of literals. Suppose that:

1. $\{D_1, \ldots, D_n\}$, with $n \geq 0$, [3] are all the clauses in a program P_j, with

[2] In some given language, usually inferred from the program and queries under consideration. The superscipt E is used to prevent confusion with the standard definition of the Herbrand base.

[3] [18, (R1)] actually stipulates that $n > 0$ and thus does not allow the selection of an atom which unifies with no clause. However, the "deletion of clauses with finitely failed body" rule [18, (R12)] can be used in those circumstances instead. We can thus effectively allow the case $n = 0$ as well.

$0 \leq j \leq k$, such that A is unifiable with $hd(D_1), \ldots, hd(D_n)$, with most general unifiers $\theta_1, \ldots, \theta_n$, and

2. C_i is the clause $(H \leftarrow F, bd(D_i), G)\theta_i$, for $i = 1, \ldots, n$.

Each binding in $\theta_i|_{vars(F) \cup vars(H)}$ is called a *left-propagated binding.*
If we *unfold C wrt A* (using $D_1, \ldots, D_n$) in P_j, we derive the clauses $C_1, \ldots, C_n$ and we get the new program $P_{k+1} = (P_k \setminus \{C\}) \cup \{C_1, \ldots, C_n\}$. When $n = 1$, i.e., there is exactly one clause whose head is unifiable with A, the unfolding is called *determinate*. Finally, *left-most unfolding* unfolds the first literal in $bd(C)$ (i.e. F is empty).[4]

For example, given $P_0 = \{p \leftarrow q \wedge r, q \leftarrow r\}$, we can unfold $p \leftarrow q \wedge r$ wrt q using P_0, deriving the clause $p \leftarrow r \wedge r$ and we get $P_1 = \{p \leftarrow r \wedge r, q \leftarrow r\}$.

Note that, in contrast to [18], we treat programs as *sets* of clauses and not as sequences of clauses. For pure tabled programs, the order (and multiplicity) of clauses makes no difference for the termination properties we are (primarily) interested in preserving (the order of clauses and the order of solutions has no incidence on *universal* — i.e. wrt the entire computation process — termination; it might however affect their *existential* termination, as discussed in Section 6.4).

3 Unfolding Endangers Termination

In the context of program specialisation, unfolding is the most important transformation rule. For instance, partial deduction [17] basically only employs unfolding (although a limited form of implicit folding is obtained by the Lloyd and Shepherdson closedness condition, see e.g. [14]). So in order to study specialisation and transformation of tabled logic programs we will first concentrate on the behaviour of unfolding.

In logic programs — with or without negation — executed under SLD(NF) (or variants thereof) any unfolding is totally correct and does not modify the termination behaviour of the program (see e.g. [18]). In the context of a fixed, unfair selection rule, like Prolog's left-to-right rule, unfolding can even improve termination (cf. [21]), but never worsen it. In the Prolog setting (i.e. if we take clause order, depth-first strategy into account), unrestricted unfolding can only affect the *existential* termination of programs, because unfolding can change the order of solutions. Moreover, *determinate unfolding does not modify the backtracking behaviour* [9] and can thus only be beneficial for efficiency (leading to smaller SLD(NF)-trees).

However, while unfolding is not problematic in the ordinary setting, its influence on efficiency and termination becomes rather involved in the context of tabled execution. On the one hand, contrary to Prolog-style execution, any unfolding of Datalog (or propositional) programs is safe wrt termination, as SLG terminates on such programs [3]. On the other hand however, as soon as function symbols are introduced, unfolding, even determinate, can ruin termination.

[4] Note that left-most unfolding is allowed to instantiate the head, while unfolding without left-propagation is not.

We suppose from now on, for simplicity of the presentation, that all predicates are tabled. We also suppose that the predicate $=/2$ is defined by the clause $=(X,X) \leftarrow$.

Example 1. Let P be the following program.

$$p(X) \leftarrow p(Y), Y = f(X)$$

Under a left-to-right selection rule, this program fails finitely (the selected atom is a variant of a call for which no answers can be produced) and thus terminates for e.g. the query $\leftarrow p(X)$. The following program, P', obtained by (determinately) unfolding the atom $Y = f(X)$ does not (see Fig. 1):

$$p(X) \leftarrow p(f(X))$$

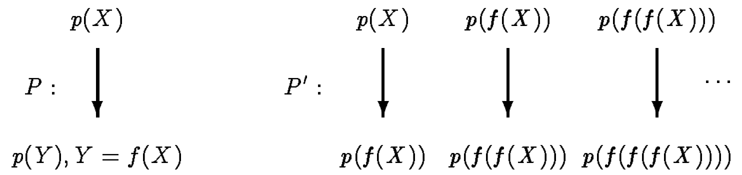

Fig. 1. SLG-forests for the query $\leftarrow p(X)$ before and after unfolding Example 1.

This "infinite slowdown" is of course highly undesirable. In the remainder of this paper we develop criteria which ensure that termination is preserved by unfolding.

4 Quasi-Termination

We start out with a formalisation of termination (under a left-to-right selection rule) in the setting of tabled execution of logic programs.

Definition 2. (call graph) Given a program P, the *call graph of* P is the graph whose nodes are the elements of B_P^E and which contains a directed edge from A to B, denoted by $A \rightarrow_P B$ (or more precisely $A \rightarrow_{P,C,i} B$), iff there exists a (renamed apart) clause $C = H \leftarrow B_1, \ldots, B_n$ in P such that

- A and H unify via an *mgu* θ and
- $B \approx B_i\theta\theta_1 \ldots \theta_{i-1}$ where θ_j is a c.a.s. for $P \cup \{\leftarrow B_j\theta\theta_1 \ldots \theta_{j-1}\}$.

By $\rightarrow_P^*$ we denote the transitive and reflexive closure of $\rightarrow_P$. Given an atom $A \in B_P^E$, we also define $A_P^* = \{B \in B_P^E \mid A \rightarrow_P^* B\}$. A subset $\mathcal{S}$ of B_P^E is said to be *closed* iff $A \rightarrow_P B$, for some $A \in \mathcal{S}$, implies that $B \in \mathcal{S}$.

Example 2. Let P be the following program.

$$p(a) \leftarrow$$

$p(X) \leftarrow q(X), p(X)$
$q(a) \leftarrow$
$q(b) \leftarrow$

We have that e.g. $p(a) \rightarrow_P q(a)$, $p(a) \rightarrow_P p(a)$ and $p(a)^* = \{p(a), q(a)\}$. The full call graph is depicted in Fig. 2.

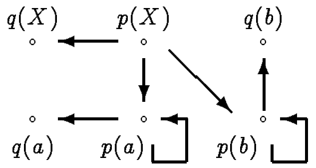

Fig. 2. Call graph of P for Example 2

We now define the notion of quasi-termination of tabled logic programs (a term borrowed from [12], defining a similar notion in the context of termination of off-line partial evaluation of functional programs).

Definition 3. (quasi-termination) Let P be a program and $\mathcal{S}$ a subset of B_P^E. P is said to be *quasi-terminating wrt* $\mathcal{S}$ iff for every $A \in \mathcal{S}$ the set A_P^* is finite. Also, P is *quasi-terminating* iff it is quasi-terminating wrt B_P^E.

E.g. the program P from Example 2 is quasi-terminating wrt the entire B_P^E.

The above definition in essence means that, starting from $\mathcal{S}$, evaluation produces only a finite number of different calls. Equivalently, every LD-tree (i.e. an SLD-tree using the left-to-right selection rule) for $P \cup \{\leftarrow A\}$, $A \in \mathcal{S}$, contains only a finite number of selected atoms modulo variable renaming. It is also equivalent to stating that every SLG-forest using the left-to-right selection rule for $P \cup \{\leftarrow A\}$, $A \in \mathcal{S}$, contains only finitely many SLG-trees. This means that non-termination can only occur if a call produces an infinite number of computed answers. Hence, *universal* termination holds iff we have quasi-termination and there are only finitely many c.a.s. for the selected atoms (for a more thorough discussion on termination issues in tabled execution see [6]) Thus, if a transformation sequence preserves the set of c.a.s. — something which holds for all the transformations we are interested in — then *preserving quasi-termination is equivalent to preserving universal termination.*

5 Preserving Quasi-Termination While Unfolding

In the problematic Example 1 we have, by unfolding, left-propagated the binding $Y/f(X)$ on the atom $p(Y)$. Without left-propagation the sequence of c.a.s. under untabled LD-resolution is not changed (see e.g. [21]) and even the behaviour of problematic non-logical built-in's like *var/1* is preserved (see e.g. [20, 25]). We first prove that such restricted unfolding is also safe wrt quasi-termination of tabled logic programs.

Theorem 4. *Let $\mathcal{S} \subseteq B_P^E$, and let P' be obtained from P by a sequence of left-most unfolding steps and unfolding steps without left-propagated bindings. If P is quasi-terminating wrt $\mathcal{S}$ then so is P'.*

Proof. In Appendix A. □

We will now explore extensions of this basic result, as the left-propagation of bindings can often be highly beneficial and, as it allows the evaluation to focus on only the relevant data, can lead to a dramatic pruning of the search space (see e.g. [13]).

However, considerable care has to be taken when performing instantiations in the context of tabled logic programs. We have already illustrated the danger of unfolding with left-propagation of bindings. Note, however, that if we just instantiate the query $\leftarrow p(X)$ to $\leftarrow p(f(X))$, but leave the clause in program P of Example 1 unmodified, quasi-termination is not destroyed. So, one might hope that instantiating a query and leaving the program unmodified should be safe wrt quasi-termination. Alas, there is another more subtle reason why left-propagation of bindings can endanger termination. Termination of ordinary SLD(NF)-resolution has the following property.

Definition 5. The termination of an evaluation strategy $\mathcal{E}$ is *closed under sustitution* iff whenever in a program P a query $\leftarrow Q$ terminates under $\mathcal{E}$, then so does every instance $\leftarrow Q\theta$ of it.

Surprisingly, this is not a property that carries over to tabled evaluation! We show that *termination (and quasi-termination) of SLG-resolution is not closed under sustitution* with the following counterexample.

Example 3. Let $p/2$ be a tabled predicate defined by the following clause.

$p(f(X), Y) \leftarrow p(X, Y)$

Then, *both* under variant- and subsumption-based tabling, the query $\leftarrow p(X, Y)$ terminates while $\leftarrow p(X, X)$ does not!

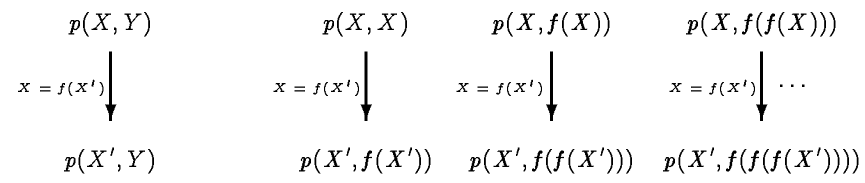

Fig. 3. SLG-forests for the queries $\leftarrow p(X, Y)$ and $\leftarrow p(X, X)$.

As a side-comment, note that, because termination of tabled execution is not closed under sustitution, tabling systems based on (*forward*) *subsumption* have unpredictable termination characteristics in general (termination of queries depends on the chronological order of encountering tabled calls). At least from a purely practical perspective, this is yet another advantage of tabling systems based on variance over those based on subsumption.

Example 3 can be adapted to show that *even left-propagation of bindings which do not introduce any new structure can be dangerous*: $t \leftarrow p(X,Y), X = Y$ terminates while a program containing $t \leftarrow p(X,X)$ does not. Moreover, a variant of the same example shows that *even left-propagation of bindings to variables that appear only in the head can ruin termination* as can be seen by the unfolding of the atom $Z = f(X)$ in the following clause:

$p(Z,Y) \leftarrow p(X,Y), Z = f(X)$

So, although left-propagation and instantiation in the context of tabled execution of logic programs seems like a hopeless endeavour, we will now formally establish that a non-trivial class of substitutions can actually be safely left-propagated.

Definition 6. A substitution γ is called a *grounding substitution* iff for all $X/t \in \gamma$ we have that t is a ground term.
We say that γ is *structurally simpler* than another grounding substitution σ, denoted by $\gamma \trianglelefteq \sigma$, iff for every $X/s \in \gamma$ there exists a $Y/t \in \sigma$ such that s is a subterm of t.

Note that any term is considered a subterm of itself.

Example 4. Let $\sigma = \{X/f(a), Y/b\}$ be a grounding substitution. Then σ itself as well as e.g. $\{Z/a, X/b\}$ and $\{X/f(a), Y/f(a), Z/b, V/a\}$ are structurally simpler than σ (it would be possible to disallow the last case by a more refined definition, but it is not required for our purposes). However, neither $\{Z/f(f(a))\}$ nor $\{X/c\}$ are structurally simpler than σ.

The interest of the relation $\trianglelefteq$, in the context of quasi-termination, derives from the following proposition.

Lemma 7. *Let σ be a grounding substitution and let A be an atom. Then the set $\{A'\gamma \mid A' \approx A$ and $\gamma \trianglelefteq \sigma\}$ is finite up to variable renaming.*

Proof. Let $vars(A') = \{X_1, \ldots, X_n\}$ and let $\gamma \trianglelefteq \sigma$. Then we either have $X_i\gamma = X_i$ or we have that $X_i\gamma = t_i$ where t_i is a subterm of some s_i with $Y/s_i \in \sigma$. Now, as there are only finitely many bindings Y/s in σ and as for each such s there are only finitely many subterms, we can only construct finitely many different atoms $A'\gamma$ up to variable renaming. □

Next, we prove the following lemma, capturing an interesting property of grounding substitutions. Together with Lemma 7, this will enable us to show that left-propagation of grounding substitutions is safe wrt quasi-termination.

Lemma 8. *Let γ be a grounding substitution and let $\leftarrow Q\gamma$ have a derivation leading to $\leftarrow RQ'$. Then $\leftarrow Q$ has a corresponding derivation leading to $\leftarrow RQ$ such that for some grounding substitution $\gamma' \trianglelefteq \gamma$, $RQ' \approx RQ\gamma'$.*

Proof. In Appendix B. □

We will now put the above lemmas to use.

Theorem 9. *Let P be a program, A an atom and let σ be a grounding substitution. If A_P^* is finite then so is $A\sigma_P^*$.*

Proof. By Lemma 8 we know that for every $A \rightarrow_P^* B$ we can only have $A\sigma \rightarrow_P^* B\gamma$ for grounding substitutions $\gamma \trianglelefteq \sigma$. This means that $\mathcal{A} = \{B'\gamma \mid B \in A^* \wedge B' \approx B$ and $\gamma \trianglelefteq \sigma\}$ is a safe approximation (i.e. a superset) of $A\sigma_P^*$. We can apply Lemma 7 to deduce that $\mathcal{A}$ is finite whenever A^* is. □

Theorem 10. *Let $\mathcal{S} \subseteq B_P^E$, and let P' be obtained from P by left-most unfolding steps and unfolding steps such that each left-propagated binding is a grounding substitution. If P is quasi-terminating wrt $\mathcal{S}$ then P' is quasi-terminating wrt $\mathcal{S}$.*

Proof Sketch. The full proof is obtained by adapting the proof of Theorem 4 to make use of Theorem 9 for the left-propagated grounding substitutions. The only tricky aspect is that, when instantiating a body atom B of a clause C to $B\gamma$, Theorem 9 only tells us that if B was terminating in P then $B\gamma$ is also terminating in P. To actually infer that $B\gamma$ also terminates in $P' = P \setminus C \cup \{C_1, \dots, C_n\}$ we have to take into account that γ might be ***repeatedly*** applied, i.e. whenever a derivation of B uses the clause C. This is no problem, however, because $\gamma' \trianglelefteq \gamma \Rightarrow \gamma'\gamma \trianglelefteq \gamma$, meaning that Lemma 8 (and thus also Theorem 9) also holds when the grounding substitution is repeatedly applied. □

A similar result does not hold when using tabling based on subsumption rather than variance as shown by the following example.

Example 5. The following program is quasi-terminating wrt $\{q\}$ when using subsumption checks (but not when using variant checks).

$p(X) \leftarrow p(f(X))$
$q \leftarrow p(X), X = a$

Unfolding $X = a$ in the last clause will result in the left-propagation of the grounding substitution $\{X/a\}$ and produce the clause $q \leftarrow p(a)$. The resulting program is no longer quasi-terminating wrt $\{q\}$ when using subsumption checks only (term-depth abstraction in the spirit of OLDT-resolution is then also required to ensure termination).

6 Extensions and Efficiency Considerations

6.1 Mixing Tabled and Prolog-Style Execution

So far we have assumed that all predicates are tabled. When not all predicates are tabled, then one can safely left-propagate any substitution on un-tabled predicates *if* they do not call tabled predicates themselves (otherwise a problem similar to Example 3 can arise; e.g. through left-propagation of X/Y on the untabled call $q(X,Y)$, where $q/2$ is defined by $q(X,Y) \leftarrow p(X,Y)$).

One also has to ensure that unfolding does not replace a tabled predicate by an un-tabled one. Otherwise, the termination might be affected as the following example shows.

Example 6. In program P of Figure 4 where only $t/1$ is tabled all queries finitely fail; so the program is terminating. However, by determinate unfolding of the first clause wrt to $t(X)$, we end up with the program P' on the right side of the same figure for which the query $\leftarrow p(X)$ is non-terminating.

$$P : \boxed{\begin{array}{c} p(X) \leftarrow t(X) \\ \\ t(X) \leftarrow p(X) \end{array}} \qquad\qquad P' : \boxed{\begin{array}{c} p(X) \leftarrow p(X) \\ \\ t(X) \leftarrow p(X) \end{array}}$$

Fig. 4. P' is obtained from P by unfolding wrt $t(X)$.

6.2 Polyvariance and Renaming

Most partial evaluators and deducers use a technique called *renaming* (see e.g. [10, 1]) to remove redundant structure from the specialised program but also to ensure the independence condition of [17] (thus avoiding the use of abstraction instead), thereby allowing unlimited polyvariance. In the context of SLD(NF)-execution, such additional polyvariance might increase the code size but is always beneficial in terms of the size of the (run-time) SLD(NF)-trees. The following example shows that, again, in the context of tabled execution, appropriate care has to be taken.

Example 7. Let P be the following program, containing some arbitrary definition of the predicate $p/2$.

$q(X,Y) \leftarrow p(a,Y), p(X,b)$

After specialising one might obtain the following program along with definitions for $p_a/1$ and $p_b/1$.

$q(X,Y) \leftarrow p_a(Y), p_b(X)$

For the query $\leftarrow q(a,b)$ the call $p(a,b)$ will only be executed once against program clauses in the original program while in the specialised program both $p_a(b)$ *and* $p_b(a)$ will be executed. The specialised program might thus actually be less efficient than the original one !

A conservative, but safe, approach is to apply renaming only when the atoms are independent — it should just be used to remove superfluous structure from the specialised program while the independence condition should be ensured via abstraction.

Similar difficulties can arise when performing conjunctive partial deduction [14, 11] (as well as tupling or deforestation), which specialises entire conjunctions and renames them into new atoms. Indeed, renaming a conjunction into a new atom might diminish the possibility for tabling, i.e. the possibility of reusing earlier computed results. If e.g. we rename the conjunction $p(X,Y) \wedge q(X,Z)$ into $pq(X,Y,Z)$ then the query $\leftarrow p(a,b) \wedge q(a,c)$ can reuse part of the results computed for $\leftarrow p(a,b) \wedge q(a,d)$, while the renamed query $\leftarrow pq(a,b,c)$ cannot reuse results computed for $\leftarrow pq(a,b,d)$.

6.3 A note on the efficiency of unfolding with left-propagation

In untabled execution the left-propagation of substitutions usually prunes the search space, and thus can only be beneficial for the efficiency of query evaluation. In tabled execution, as some program clause resolution is substituted by resolution against answers that are materialised in tables and can be retrieved without recomputation, the left-propagation of (even grounding) substitutions may sometimes worsen performance (of course it can also vastly improve it).

For example, in many tabled-based program analysers, it is usual practice to employ what is known as the *most-general call optimisation* [4]; i.e., compute analysis information for the most general form of each predicate and then retrieve this information from the tables, appropriately filtering it by explicit equality constraints. The basic idea of the approach can be illustrated by two simple abstract interpreters from [5] shown in Figure 5. The definition of the $fact/1$ predicate (which can be the only tabled predicate) is the same for both interpreters and assumes that each program clause $H \leftarrow G_1, \ldots, G_n$ is represented as a fact of the form $pc(H, [G_1, \ldots, G_n]) \leftarrow$. A top-level query $\leftarrow fact(X)$ triggers the computation of the program's non-ground minimal S-model that gets recorded in the tables. Only one such table is created when using the interpreter with the most-general call optimisation (shown on the right side of the figure). However, using the interpreter on the left side of the figure the number of tables depends on the number of distinct (up to variance) instantiation patterns for calls to $fact/1$. Besides the overhead in space, this has an associated performance cost especially in complicated abstract domains (see e.g. [4, 5] for more information and experimental evaluation of this technique over a variety of domains).

$fact(Head) \leftarrow$ $\quad pc(Head, Body), prove(Body)$ $prove([\,]) \leftarrow$ $prove([G\|Gs]) \leftarrow$ $\quad fact(G), prove(Gs)$	$fact(Head) \leftarrow$ $\quad pc(Head, Body), prove(Body)$ $prove([\,]) \leftarrow$ $prove([G\|Gs]) \leftarrow$ $\quad fact(GenG), GenG = G, prove(Gs)$

Fig. 5. Two abstract meta-interpreters for concrete evaluation.

6.4 Taking program clause order into account

In existing implementations of tabling, program clause resolution is performed in a style similar to Prolog's; i.e. visiting clauses according to their textual order. Consequently, in tabled programs which are not quasi-terminating, non-determinate unfolding (even without left-propagation) can worsen their existential termination. This, in turn, might affect their behaviour under optimisations which involve pruning, such as *existential negation* (c.f. [24]). We illustrate the problem by the following example, which extends a similar example given in [21].

Example 8. Assuming a scheduling strategy that returns answers to calls as soon as these are generated, program P of Figure 6 produces the answer $X = 0$ for the query $\leftarrow p(X)$, and then loops, while program P' loops without producing any answers.

$$P: \boxed{\begin{array}{l} p(X) \leftarrow q(X), r \\ \\ q(0) \leftarrow \\ q(X) \leftarrow q(s(X)) \\ \\ r \leftarrow fail \\ r \end{array}} \qquad P': \boxed{\begin{array}{l} p(X) \leftarrow q(X), fail \\ p(X) \leftarrow q(X) \\ \\ q(0) \leftarrow \\ q(X) \leftarrow q(s(X)) \\ \\ r \leftarrow fail \\ r \end{array}}$$

Fig. 6. P' is obtained from P by unfolding wrt r.

7 Discussion

We conclude with a brief discussion of a possible application of this work, namely optimising integrity checking upon updates in *recursive* databases, by specialising meta-interpreters. For hierarchical databases successful results have already been achieved in [13]. Unfortunately, moving to recursive databases has proven to be difficult, because the loop check — which is necessary for termination — requires the use of the ground representation when using SLD(NF)-execution. This imposes a large initial overhead and leads to further difficulties in terms of specialisation [15, 16]. However, by writing the integrity checker in a tabled environment we can use the non-ground representation and together with the techniques explored in this paper, one might obtain effective specialised update procedures even for recursive databases. Note that in the setting of deductive databases left-propagation of grounding substitutions corresponds to the well-known optimisation principle of making selections first, and thus occurs very naturally.

Acknowledgements

Michael Leuschel is supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation". Bern Martens is a post-doctoral fellow of the K.U.Leuven Research Council. Konstantinos Sagonas is a post-doctoral fellow of the Flemish Fund for Scientific Research (FWO). We thank Danny De Schreye and Stefaan Decorte for interesting discussions, ideas and comments.

References

1. K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, October 1993.
2. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
3. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.
4. M. Codish, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Exploiting Goal Independence in the Analysis of Logic Programs. *J. of Log. Prog.*, 32(3):246–261, September 1997.
5. M. Codish, B. Demoen, and K. Sagonas. General Purpose Semantic Based Program Analysis using XSB. K.U. Leuven Technical Report CW 245. December 1996. Submitted for publication.
6. S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K. Sagonas. Termination analysis for tabled logic programming. Technical Report CW 2??, Departement Computerwetenschappen, K.U. Leuven, Belgium, April 1997. Submitted for Publication.
7. M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Comput. Sci.*, 69(3):289–318, 1989.
8. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
9. 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, Copenhagen, Denmark, 1993. ACM Press.
10. J. Gallagher and M. Bruynooghe. Some low-level transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 229–244, Leuven, Belgium, 1990.
11. R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling Conjunctive Partial Deduction of Definite Logic Programs. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, number 1140 in LNCS, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag. Extended version as Technical Report CW 226, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.
12. C. K. Holst. Finiteness Analysis. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA)*, number 523 in LNCS, pages 473–495. Springer-Verlag, August 1991.

13. 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.
14. M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press. Extended version as Technical Report CW 225, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.
15. M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J. W. 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`.
16. M. Leuschel and D. Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, number 1140 in LNCS, pages 137–151, Aachen, Germany, September 1996. Springer-Verlag. Extended version as Technical Report CW 232, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.
17. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
18. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19& 20:261–320, May 1994.
19. S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.
20. 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.
21. 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.
22. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 687–711, Japan, June 1995. The MIT Press.
23. K. Sagonas. *The SLG-WAM: A Search-Efficient Engine for Well-Founded Evaluation of Normal Logic Programs.* PhD thesis, Department of Computer Science, SUNY at Stony Brook, August 1996.
24. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM Press.
25. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
26. H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984.

27. H. Tamaki and T. Sato. OLD Resolution with Tabulation. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, number 225 in LNCS, pages 84–98, London, July 1986. Springer-Verlag.

A Proof of Theorem 4

Theorem 4. Let $\mathcal{S} \subseteq B_P^E$, and let P' be obtained from P by a sequence of left-most unfolding steps and unfolding steps without left-propagated bindings. If P is quasi-terminating wrt $\mathcal{S}$ then so is P'.

Proof. Let $P = P_0, P_1, \ldots, P_r = P'$ be the transformation sequence used to obtain P'. We will prove by induction on the number of unfolding steps that in each intermediate program P_i the transitive closure of $\rightarrow_{P_i}$ will be smaller or in the worst case equal to the transitive closure of $\rightarrow_P$. This ensures that quasi-termination is indeed preserved. Let P_{i+1} be obtained from unfolding the clause $C = H \leftarrow F, A, G$ in P_i wrt A using $D_1, \ldots, D_n$ in P_j. Let $B \rightarrow_{P_i, C', k} B'$. If $C' \neq C$ then we trivially have that this part of the call graph is not modified in P_{i+1} and we have $B \rightarrow_{P_{i+1}, C', k} B'$. If on the other hand $C = C'$ then we do not have $B \rightarrow_{P_{i+1}, C, k} B'$ as C has been removed and has been replaced by the clauses $\{C_1, \ldots, C_n\}$ of Definition 1. Let $C = H \leftarrow B_1, \ldots, B_r$ with $F = B_1, \ldots, B_{q-1}$, $A = B_q$ and $G = B_{q+1}, \ldots, B_r$. There are now three possibilities:

1. $k < q$. This implies that we have applied unfolding without left-propagation (for left-most unfolding no atoms can be to the left of the unfolded atom B_k). We know that $B \rightarrow_{P_{i+1}, C'', k} B'$ for every $C'' \in \{C_1, \ldots, C_n\}$ and as no bindings are left-propagated, $C'' = H \leftarrow F \wedge Rest$. So, if $n > 0$ we have $B \rightarrow_{P_{i+1}} B'$ and the relation $\rightarrow_{P_{j+1}}$ is the same as $\rightarrow_{P_i}$. If $n = 0$ then the unfolding has possibly (there could be other ways to establish $B \rightarrow_{P_i} B'$) removed an arrow by going from $\rightarrow_{P_i}$ to $\rightarrow_{P_{i+1}}$ (if it is actually removed then termination might be improved by pruning a non-terminating computation).
2. $k > q$. In that case we have $B \rightarrow_{P_{i+1}} B'$ as unfolding preserves the set of computed answers. More precisely, by Definition 2, we know that $B' = B_k\hat{\theta}$ where $\hat{\theta} = \theta\theta_1 \ldots \theta_{k-1}$. Let us consider the following auxiliary clause $Aux = aux(\bar{X}) \leftarrow (B_1, \ldots, B_q, \ldots, B_{k-1})\theta$ where $\bar{X}$ are the variables in the body of Aux. We know that if we unfold Aux wrt B_q then the set of computed answers of e.g. the query $\leftarrow Aux$ are preserved for any selection rule (so also the left-to-right one). This means that if $\leftarrow (B_1, \ldots, B_q, \ldots, B_{k-1})\theta$ had the computed answer $\hat{\theta}$, then there must be a similar computed answer in the unfolded program and thus we also have $B \rightarrow_{P_{i+1}} B_k\hat{\theta}$.
3. $k = q$. In that case we do not have $B \rightarrow_{P_{i+1}} B'$ — unless there was another way to establish $B \rightarrow_{P_i} B'$ — but it is replaced by several (possibly) new arrows. Let $B' = B_k\hat{\theta}$ where $\hat{\theta} = \theta\theta_1 \ldots \theta_{k-1}$. Let us first consider the case that unfolding without left-propagation has been performed, i.e. let $C_s = H \leftarrow F, bd(D_s)\theta_s, G\theta_s$ be an unfolded clause where $\theta_s = mgu(B_q, hd(D_s))$. If $B \rightarrow_{P_{i+1}, C_s, q+k'} B''$ for $0 \leq k' < l$ where l is the number of atoms in $bd(D_s)$ (i.e. we have a new arrow) then we must have $B' \rightarrow_{P_j, D_s, k'} B''$:

   Take the clause $D_s = hd(D_s) \leftarrow bd(D_s)$. Then the clause $D'_s = B_q\theta_s \leftarrow bd(D_s)\theta_s$ is actually the resultant of a simple derivation of length 1 for $\leftarrow B_q$. Hence we can use e.g. Lemma 4.12 from [17] to deduce that the computed answers of an instance $\leftarrow B'$ of $\leftarrow B_q$ are the same when resolving with D_s as when resolving with D'_s.

By the induction hypothesis we know that both $B \to_{P_i} B'$ and $B' \to_{P_j} B''$ were already in the transitive closure of $\to_P$ and thus the (possibly) new arrow $B \to_{P_{i+1}} B''$ was also already in the transitive closure of $\to_P$. Now, in the case that left-most unfolding has been performed then F is empty and we have that $C_s = H\theta_s \leftarrow bd(D_s)\theta_s, G\theta_s$ with $\theta_s = mgu(B_q, hd(D_s))$. Note that in contrast to unfolding without left-propagation the head of C_s is not necessarily equal to H. However, by a similar application of Lemma 4.12 from [17] we can deduce that this has no influence and we can apply the same reasoning as above.

So, the only new arrows which are added were already in the transitive closure of $\to_P$ and the new transitive closure will thus be smaller (as some arrows might be removed) or equal than the one of $\to_P$. □

B Proof of Lemma 8

Lemma 11. *Let A and H be atoms with $vars(A) \cap vars(H) = \emptyset$. Let $\sigma = \{X/t\}$ be a grounding substitution with $X \notin vars(H)$ and $\theta = mgu(A, H)$ and $\theta' = mgu(A\sigma, H)$.*

1. *If $X \notin dom(\theta)$ then $H\theta' \approx H\theta\sigma$.*
2. *If $X \in dom(\theta)$ then there exists a substitution $\gamma \trianglelefteq \sigma$ such that $H\theta' \approx H\theta\gamma$.*

Proof. **Point 1.** If $X \notin vars(A)$ then the property is trivial, as A and $A\sigma$ are identical and X cannot be in $ran(\theta)$ by idempotence of θ.
Now let $X \in vars(A)$. Because $X \notin dom(\theta)$ and t is ground, we have, by definition of composition of substitutions, that $\theta\sigma = \sigma\theta\sigma$.[5] Hence, $(A\sigma)\theta\sigma = A\theta\sigma = H\theta\sigma$ and $\theta\sigma$ is a unifier of $A\sigma$ and H. Thus, as θ' is a most general unifier there must exist a substitution γ' such that $\theta'\gamma' = \theta\sigma$ and thus $H\theta\sigma$ is an instance of $H\theta'$.
We now establish that $H\theta'$ is in turn an instance of $H\theta\sigma$, and thus prove that $H\theta' \approx H\theta\sigma$.
Let $\sigma' = \sigma\theta'$. This substitution is a unifier of A and H: $H\sigma' = H\theta'$ (because $X \notin vars(H)$) and $A\sigma' = A\sigma\theta' = H\theta'$. So there must exist a substitution γ such that $\theta\gamma = \sigma'$ (because θ is an mgu). This means, as $X \notin dom(\theta)$, that $X/t \in \gamma$, i.e. $\gamma = \sigma\hat{\gamma}$ for some $\hat{\gamma}$ because t is ground. Now, $H\theta' = H\sigma' = H\theta\gamma = H\theta\sigma\hat{\gamma}$ and we have established that $H\theta'$ is an instance of $H\theta\sigma$.

Point 2. We have that $X/s \in \theta$ for some term s. We must have that, for some γ, $s\gamma = t$. Indeed, $\sigma\theta'$ is a unifier of A and H (see proof of Point 1.) and therefore for some substitution $\hat{\gamma}$ we have $\theta\hat{\gamma} = \sigma\theta'$. Now, because $X/t \in \sigma\theta'$ (as t is ground) we have that $s\hat{\gamma} = t$. Let $\gamma = \hat{\gamma}|_{vars(s)} = \{X_1/t_1, \ldots, X_n/t_n\}$. We have that all t_i are subterms of t and, unless $n = 1$ and $s = X_1$, the t_i must also be strict subterms. In other words, $\gamma \trianglelefteq \sigma$. We now prove that $H\theta' \approx H\theta\gamma$. First, we have that $X/t \in \theta\gamma$, by definition of γ, and we can thus find some substitution $\hat{\theta}$ such that $\theta\gamma = \sigma\hat{\theta}$ (because t is ground). Thus $\theta\gamma$ is a unifier of H and $A\sigma$: $(A\sigma)\theta\gamma = A\sigma\sigma\hat{\theta} =$ (because t is ground) $A\sigma\hat{\theta} = A\theta\gamma$ $= H\theta\gamma$. Thus $\theta\gamma$ is an instance of the mgu θ' and $H\theta\gamma$ and $A\theta\gamma$ is an instance of $H\theta'$. Secondly, as already mentioned above, $\theta\hat{\gamma} = \sigma\theta'$. Now, because $X \notin vars(H)$,

[5] This also holds if σ is not a grounding substitution but if we have $ran(\sigma) \cap vars(A) = \emptyset$ instead.

$H\theta' = H\sigma\theta'$ and thus we have: $H\theta' = H\sigma\theta' = H\theta\hat{\gamma} = H\theta\gamma\gamma''$ for some γ'' (because the t_i are ground) and $H\theta'$ is in turn an instance of $H\theta\gamma$. $\square$

The following lemma follows immediately from Definition 6.

Lemma 12. *Let $\gamma_1, \gamma_2, \sigma_1, \sigma_2$ be grounding substitutions such that $dom(\sigma_1) \cap dom(\sigma_2) = \emptyset$. If $\gamma_1 \trianglelefteq \sigma_1$ and $\gamma_2 \trianglelefteq \sigma_2$ then $\gamma_1\gamma_2 \trianglelefteq \sigma_1\sigma_2$.*

Lemma 13. *Let A and H be atoms with $vars(A) \cap vars(H) = \emptyset$. Let σ be a grounding substitution with $dom(\sigma) \cap vars(H) = \emptyset$ and $\theta = mgu(A, H)$ and $\theta' = mgu(A\sigma, H)$. There exists a substitution $\gamma \trianglelefteq \sigma$ such that $H\theta' \approx H\theta\gamma$.*

Proof. By induction on the number of elements in σ, using Lemma 11 and Lemma 12. $\square$

The following lemma again follows immediately from Definition 6.

Lemma 14. *If $\gamma_1 \trianglelefteq \gamma_2$ and $\gamma_2 \trianglelefteq \gamma_3$ then $\gamma_1 \trianglelefteq \gamma_3$*

We can finally prove Lemma 8.

Proof of Lemma 8. By induction on the length of the derivation, using Lemma 13 for each derivation step (and the fact that unifiers are relevant and the head of clauses renamed apart meaning that $H\theta' \approx H\theta\gamma$ implies $Body\theta' \approx Body\theta\gamma$ and thus also implies, together with $A\theta' = H\theta'$ and $A\theta\gamma = A\theta\gamma$, that $RQ' \approx RQ\gamma$) and using Lemma 14. $\square$

This article was processed using the LaTeX macro package with LLNCS style