Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates

Elvira Albert[1], Germán Puebla[2], and John Gallagher[3]

[1] School of Computer Science, Complutense U. of Madrid, `elvira@sip.ucm.es`
[2] School of Computer Science, Technical U. of Madrid, `german@fi.upm.es`
[3] Department of Computer Science, University of Roskilde, `jpg@ruc.dk`

Abstract. Partial evaluation of logic programs which contain *impure* predicates poses non-trivial challenges. Impure predicates include those which produce side-effects, raise errors (or exceptions), and those whose truth value varies according to the degree of instantiation of arguments[4]. In particular, non-leftmost unfolding steps can produce incorrect results since the independence of the computation rule no longer holds in the presence of impure predicates. Existing proposals allow non-leftmost unfolding steps, but at the cost of accuracy: bindings and failure are not propagated backwards to predicates which are potentially impure. In this work we propose a partial evaluation scheme which substantially reduces the situations in which such backpropagation has to be avoided. With this aim, our partial evaluator takes into account the information about purity of predicates expressed in terms of *assertions*. This allows achieving some optimizations which are not feasible using existing partial evaluation techniques. We argue that our proposal goes beyond existing ones in that it is a) accurate, since the classification of pure vs impure is done at the level of atoms instead of predicates, b) extensible, as the information about purity can be added to programs using assertions without having to modify the partial evaluator itself, and c) automatic, since (backwards) analysis can be used to automatically infer the required assertions. Our approach has been implemented in the context of `CiaoPP`, the abstract interpretation-based preprocessor of the `Ciao` logic programming system.

1 Introduction and Motivation

For logic programs without impure predicates, non-leftmost unfolding is sound thanks to the independence of the computation rule (see for example [13]).[5] Unfortunately, non-leftmost unfolding poses several problems in the context of *full* Prolog programs with *impure* predicates, where such independence does not hold anymore. For instance, `ground/1` is an *impure* predicate since, under LD resolution, the goal `ground(X),X=a` fails whereas `X=a,ground(X)` succeeds with computed answer X/a. Those executions are not equivalent and, thus, the independence of the computation rule does no longer hold. As a result, given the goal

[4] The term "partial deduction" is often used when referring to partial evaluation of pure logic programs [7]; hence we do not use it in this context.
[5] However, non-deterministic unfolding of nonleftmost atoms can degrade efficiency.

```
:- module(main_prog,[main/2],[]).
:- use_module(comp,[long_comp/2],[]).
:- entry main(X,a).

main(X,Y)   :- problem(X,Y), q(X).

problem(a,Y):- ground(Y),long_comp(c,Y).
problem(b,Y):- ground(Y),long_comp(d,Y).

q(a).
```

Fig. 1. Motivating Example

$\leftarrow$ ground(X),X=a, if we allow the non-leftmost unfolding step which binds the variable X in the call to ground(X), the goal will succeed at specialization time, whereas the initial goal fails in LD resolution at run-time. The above problem was early detected [16] and it is known as the problem of *backpropagation of bindings*. Also *backpropagation of failure* is problematic in the presence of impure predicates. For instance, $\leftarrow$ write(hello),fail behaves differently from $\leftarrow$ fail.

However, it is well-known that *non-leftmost* unfolding is essential in partial evaluation in some cases for the satisfactory propagation of static information (see, e.g., [8]). Informally, given a program P and a goal $\leftarrow A_1, \ldots, A_n$, it can happen that the leftmost atom A_1 cannot be selected for unfolding due to several circumstances. Among others, if A_1 is an atom for a predicate defined in P (thus the code is available to the partial evaluator) it can happen that i) unfolding A_1 endangers termination (for example, A_1 may homeomorphically embed [11] some selected atom in its sequence of covering ancestors), or ii) the atom A_1 unifies with several clause heads (deterministic unfolding rules do not unfold non-deterministically for atoms other than the initial query). If A_1 is an atom for an external predicate whose code is not present nor available to the partial evaluator, it can happen that A_1 is not sufficiently instantiated so as to be executed at this moment.

Example 1. Our motivating example is the Ciao program in Fig. 1, which uses the impure (predefined) predicate ground/1. Predicate long_comp/2 is external from the user module comp. Consider a deterministic unfolding rule and the entry declaration in Fig 1. The unfolding rule performs an initial step and derives the goal problem(X,a),q(X). Then, it cannot select the leftmost atom problem(X,a) because its execution performs a non deterministic step.

In this situation, different decisions can be taken. a) We can stop unfolding at this point. However, in general, it may be profitable to unfold atoms other than the leftmost. Interesting computation rules are able to detect the above circumstances and "jump over" the problematic atom in order to proceed with the specialization of another atom (in this case q(X)). We can then decide to b) unfold q(X) but avoiding backpropagating bindings nor failure onto problem(X,a). And the final possibility c) is to unfold q(X) while allowing backpropagation onto problem(X,a). However, this will require that some additional requirements hold on the atom(s) to the left of the selected one. Our main aim in this work is to identify and characterize the conditions under which the possibility c) above is applicable and build a partial evaluation system which can effectively prove such conditions in order to perform backpropagation of bindings and failure as much as possible.

There are several solutions in the literature (see, e.g.,[10, 2, 1, 8, 9]) which allow unfolding non-leftmost atoms by avoiding the backpropagation of bindings and failure, i.e., in the spirit of possibility b). Basically, the common idea is to represent explicitly the bindings by using unification [10] or residual case expressions [1] rather than backpropagating them (and thus applying them onto leftmost atoms). For our example, by using unification, we can unfold `q(X)` and obtain the resultant `main(X,a):-problem(X,a),X=a`. This guarantees that the resulting program is correct, but it definitely introduces some inaccuracy, since bindings (and failure) generated during unfolding of non-leftmost atoms are hidden from atoms to the left of the selected one. The relevant point to note is that preventing backpropagation, by using one of the existing methods, can be a bad idea for at least the following reasons:

1. *Backpropagation of bindings and failure can lead to an early detection of failure*, which may result in important speedups. For instance, if we allow backpropagating the binding `X=a` to the left atom, we get rid of the whole (failing) computation for `problem(b,a)` in the residual program.
2. *Backpropagation of bindings can make the profitability criterion for the leftmost atom to hold*, which may result in more aggressive unfolding. In the example, by backpropagating, we obtain the atom `problem(a,a)` which allows a deterministic computation rule to proceed to its unfolding.
3. *Backpropagation of bindings may allow improved indexing* by further instantiating arguments in clause heads. This is often good from a performance point of view (see, e.g., [17]). In our example, we will obtain the clause head `main(a,a)` with more indexing than `main(X,a)`.

The bottom-line is that backpropagation should be avoided only when it is really necessary since interesting specializations can no longer be achieved when it is disabled.

The remaining of the paper is organized as follows. The next section provides an overview of our partial evaluation scheme. Section 3 recalls some preliminary notions. In Sect. 4 we formalize the notion of purity at the level of atoms. Section 5 presents the soundness conditions which allow safe backpropagation of bindings and failure. In Sect. 6, we propose a partial evaluation scheme based on purity assertions which are automatically inferred by backwards analysis. We conclude in Sect. 7.

2 An Overview of our Partial Evaluation Scheme

Automatically figuring out when bindings and/or failure can be safely backpropagated onto an atom whose execution potentially reaches an impure predicate has been considered a difficult challenge and, to our knowledge, there is no accurate, satisfactory solution. Existing methods [8] are based on simple reachability analysis. As soon as an impure predicate p/n can be reached from a predicate q/m, also q/m is considered impure and backpropagation onto any atom A for

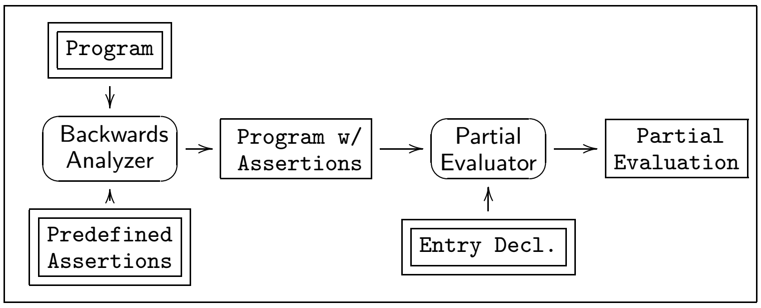

Fig. 2. Partial Evaluation based on Assertions and Backwards Analysis

q/m is not allowed. Unfortunately, this notion of impurity quickly expands from a predicate to all predicates which use it. For example, the fact that there is a call to an impure predicate within `problem/2` will avoid backpropagating the binding for `X` and thus achieving the above three enumerated effects.

Figure 2 illustrates our partial evaluation scheme which is made up of three main components. First, we propose to use *assertions* which establish the conditions under which atoms (i.e., calls) for potentially impure predicates become pure. The classification of pure vs impure is thus done at the level of atoms instead of predicates, which will give us more precise results. We start from a set of `Predefined Assertions` provided by the underlying system for predefined predicates. Second, the role of `Backwards Analyzer` is to automatically infer, from the predefined assertions, sufficient conditions under which atoms are pure. The result is specified by extending the program, resulting in `Program with Assertions`. Notice that this is a goal-independent process which can be started in our system regardless of PE being performed or not. Third, and independently from the backwards analysis process, the user can decide to partially evaluate the program. To do so, an initial call has to be provided by means of an `Entry Declaration`. A Partial Evaluator is executed from such program and entry with the only consideration that, whenever a non-leftmost unfolding step needs to be performed, it will take into account the information available in the generated assertions. In our example, we will show that it is able to detect that, in the context described by our entry, all calls to `problem/2` are pure since the second argument is always ground. This allows us to backpropagate the binding for `X` and obtain the fact "`main(a,a).`" as partially evaluated program which achieves the three benefits enumerated above.

3 Background

We assume some basic knowledge on the terminology of logic programming. See for example [13] for details. Very briefly, an *atom* A is a syntactic construction of the form $p(t_1, \dots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \dots, t_n$ are terms. The function *pred* applied to atom A, i.e., $pred(A)$, returns the predicate symbol p/n for A. Most real-life Prolog programs use predicates which are not defined in the program (module) being developed. Thus, predicates are classified into *internal* and *external*. Internal procedures are defined in the current program (module) and we assume that its code is available to the partial evaluator, whereas external predicates are not present. Examples of external predicates include the traditional "built-in" (predefined) predicates, such as constraints, basic input/output facilities (e.g., `open`). We will also consider as external predicates those defined in a different module, procedures written in another language, etc.

A *clause* is of the form $H \leftarrow B$ where its head H is an atom and its body B is a conjunction of atoms. A *program* is a finite set of clauses. A *goal* (or query) is a conjunction of atoms. The concept of *computation rule* is used to select an atom within a goal for its evaluation. The operational semantics of programs is based on derivations. Consider a program P and a goal G of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_k$. Let $\mathcal{R}$ be a computation rule such that $\mathcal{R}(G) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause in program P. Then $\theta(A_1, \ldots, A_{R-1}, B_1, \ldots, B_m, A_{R+1}, \ldots, A_k)$ is *derived* from G and C via $\mathcal{R}$ where $\theta = mgu(A_R, H)$. An *SLD derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P, and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in *SLD trees*. A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is empty. In that case $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the computed answer for goal G. Such a derivation is called *failed* if it is not possible to perform a derivation step with G_n. We will also allow *incomplete* derivations in which, though possible, no further resolution step is performed. We refer to SLD resolution restricted to the case of leftmost computation rule as LD resolution.

Partial Evaluation (PE) [12, 4] is a program transformation technique which specializes a program w.r.t. part of its known input data. Hence it is sometimes also known as program specialization. Informally, given an input program and a set of atoms, the PE algorithm applies an *unfolding rule* in order to compute finite (possibly incomplete) SLD trees for these atoms. This process returns a set of *resultants* (or residual rules), i.e., a residual program, associated to the root-to-leaf derivations of these trees. Formally, an unfolding rule computes a set of finite SLD derivations $D_1, \ldots, D_n$ (i.e., a possibly incomplete SLD tree) of the form $D_i = A, \ldots, G_i$ with computed answer substitution θ_i for $i = 1, \ldots, n$ whose associated *resultants* (or residual rules) are $\theta_i(A) \leftarrow G_i$. Note that in contrast to PE of pure programs, in the presence of impure predicates, failing derivations cannot be blindly eliminated from the set of resultants, since this may not preserve the behaviour of the program w.r.t. side-effects. Each unfolding step during partial evaluation can be conceptually divided into two steps. First, given a goal $\leftarrow A_1, \ldots, A_R, \ldots, A_k$ the computation rule determines the selected atom A_R. Second, it must be decided whether unfolding (or evaluation) of A_R is *profitable*. It must be noted that the unfolding process requires the introduction of this profitability test in order to guarantee that unfolding terminates. Also, unfolding usually continues as long as some evidence is found that further unfolding will improve the quality of the resultant program.

3.1 Leftmost Unfolding with Impure and External Predicates

The trivial computation rule which always returns the leftmost atom in a goal is interesting in that it avoids several correctness and efficiency issues in the context of PE of full Prolog programs. Such issues are discussed in depth throughout this paper. When a (leftmost) atom A_R is selected during PE, with *pred*(A_R) $= p/n$ being an external predicate, it may not be possible to unfold A_R for several reasons. First, we may not have the code defining p/n and, even if we have it, unfolding A_R may introduce in the residual program calls to predicates which are private to the module where p/n is defined. Also, it can be the case that the execution of atoms for (external) predicates produces other outcomes such as side-effects, errors, and exceptions. Note that this precludes the evaluation of such atoms to be performed at PE time, since those effects need to be performed at run-time. In spite of this, if the executable code for the external predicate p/n is available, and under certain conditions, it can be possible to fully evaluate A_R at specialization time. The notion of *evaluable* atom [14] captures the requirements which allow the *leftmost* execution of external predicates at PE time. Informally, an atom is evaluable if its execution satisfies four conditions: 1) it universally terminates, 2) it does not produce side-effects, 3) it does not issue errors and 4) it is sufficiently instantiated. We use $\mathsf{eval}(E)$ to denote that the expression E is evaluable.

4 From Impure Predicates to Impure Atoms

Existing techniques for PE allow the unfolding of non-leftmost atoms by combining a classification of predicates into pure and impure with techniques for avoiding backpropagation of binding and failure in the case of impure predicates. In order to classify predicates as pure or impure, existing methods [8] are based on simple reachability analysis.

Our work improves on existing techniques by 1) providing a finer-grained notion of impurity, which rather than being defined at the level of *predicates*, is defined at the level of individual *atoms*, and 2) splitting the notion of purity into its constituent properties: binding-sensitiveness, errors and side effects. Defining purity at the level of atoms is of interest since it is often the case that some atoms for a predicate are pure whereas others are impure. As an example, the atom $var(X)$ is impure (binding sensitive), whereas the atom $var(f(X))$ is not (it is no longer binding sensitive). As will be seen later, this allows *reducing* substantially the situations in which backpropagation has to be avoided.

4.1 Binding-sensitiveness

A *binding-sensitive* predicate is characterized by having a different success or failure behaviour under leftmost execution if bindings are backpropagated onto it. Examples of binding-sensitive predicates are `var/1`, `nonvar/1`, `atom/1`, `number/1`, `ground/1`, etc.

Definition 1 (binding insensitive atom). *An atom A is* binding insensitive*, denoted* $\mathsf{bind\_ins}(A)$*, if* $\forall$ *sequence of distinct variables* $\langle X_1, \ldots, X_k \rangle$ *s.t.* $X_i \in vars(A)$*,* $i = 1, \ldots, k$ *and* $\forall$ *sequence of terms* $\langle t_1, \ldots, t_k \rangle$*, the goal* $\leftarrow (X_1 = t_1, \ldots, X_k = t_k, A)$ *succeeds in LD resolution with computed answer* σ *iff the* *goal* $\leftarrow (A, X_1 = t_1, \ldots, X_k = t_k)$ *also succeeds in LD resolution with computed answer* σ.

Let us note that in the definition above we are only concerned with successful derivations, which we aim at preserving. However, we are not in principle concerned about preserving infinite failure. For example, $\leftarrow (A, X = t)$ and $\leftarrow (X = t, A)$ might have the same set of answers but a different termination behaviour. In particular, the former might have an infinite derivation under LD resolution while the second may finitely fail.

If an atom contains no variables, binding insensitiveness trivially holds. This is quite useful in practice, since it may allow considering a good number of atoms as binding insensitive without the need of sophisticated analyses.

4.2 Side-effects

Predicates p/n for which $\leftarrow A, fail$ and $\leftarrow fail$, with $pred(A) = p/n$, are not equivalent in LD resolution are termed as "*side-effects*" in [16]. Typical examples of predicates with side-effects are `write/1` and `assert/1`.

Definition 2 (side-effect-free atom). *An atom A is* side-effect free*, denoted* $\mathsf{sideff\_free}(A)$*, if the run-time behaviour of* $\leftarrow A, fail$ *is equivalent to that of* $\leftarrow fail$*.*

Since side-effects have to be preserved in the residual program, we have to avoid any kind of backpropagation which can anticipate failure and, therefore, hide an existing side-effect.

4.3 Run-Time Errors

There are some predicates whose call patterns are expected to be of certain type and/or instantiation state. If an atom A does not correspond to the intended call pattern, the execution of A will issue some *run-time errors*. Since we consider such run-time errors as part of the behaviour of a program, we will require that the partial evaluation process produces a residual program whose behaviour w.r.t. run-time errors is identical to that of the original program, i.e., run-time errors must not be introduced to, nor removed from, the program.

For instance, the predefined predicate `is/2` requires its second argument to be an arithmetic expression. If that is detected not to be the case at run-time, an error is issued. Clearly, backpropagation is dangerous in the context of atoms which may issue run-time errors, since it can anticipate the failure of a call to the left of `is/2` (thus omitting the error), or it can make the call to `is/2` not to issue an error (if there is some free variable in the second argument which gets instantiated to an arithmetic expression after backpropagation).

Definition 3 (error-free atom). *An atom A is* error-free*, denoted* $\mathsf{error\_free}(A)$*, if the execution of A does not issue any error.*

Somewhat surprisingly this condition for PE corresponds to that used in [6] for computing safe call patterns. Unfortunately, the way in which errors are issued can be implementation dependent. Some systems may write error messages and continue execution, others may write error messages and make the execution of the atom fail, others may halt the execution, others may raise exceptions, etc. Though errors are often handled using side-effects, we will make a distinction between side-effects and errors for two reasons. First, side-effects can be an expected outcome of the execution, whereas run-time errors should not occur in successful executions. Second, it is often the case that predicates which contain side-effects produce them unconditionally for all (or most of) atoms for such predicate. However, predicates which can generate run-time errors can be guaranteed not to issue errors when certain preconditions about the call are satisfied, i.e., when the atom is well-moded and well-typed. A practical implication of the above distinction is that simple, reachability analysis will be used for propagating side-effect freeness at the level of predicates, whereas a more refined, atom-based classification will be used in the case of error-freeness.

5 Soundness Conditions for Backpropagation

Given the definitions of binding insensitive, side-effect free, and error free atoms, we proceed to define aggregate properties which summarize the effect of such individual properties. These properties will allow us to define the soundness conditions under which backpropagation of bindings and failure is correct.

5.1 Backpropagation of failure

The next definition formalizes the concept of *observable-free* atom which is required in order to determine whether backpropagation of *failure* is permitted.

Definition 4 (observable-free atom). *An atom A is* observable-free*, denoted* $\mathsf{observable\_free}(A)$*, if* $\mathsf{error\_free}(A) \wedge \mathsf{sideff\_free}(A)$

Intuitively, if an atom A is not observable-free, then $\leftarrow A, fail$ may behave differently from $\leftarrow fail$ and thus backpropagation onto A has to be avoided. The notion of *observable-safe step* characterizes the derivation steps for which backpropagation of failure is not problematic.

Definition 5 (observable-safe derivation step). *Let P be a program, let $G = \leftarrow A_1, \ldots, A_n$ be a goal and let $\mathcal{R}$ be a computation rule s.t. $\mathcal{R}(G) = A_R$. Let C be a renamed apart clause in P s.t. the head of C unifies with A_R. We say that the derivation step for G and C via $\mathcal{R}$ is* observable-safe *if* $\mathsf{observable\_free}(A_1) \wedge \ldots \wedge \mathsf{observable\_free}(A_{R-1})$.

The notion of observable-safe derivation step can be incorporated in a PE system in a straightforward way. More concretely, the computation rule used within the unfolding rule can be defined in such a way that tries to select first those atoms whose evaluation gives rise to observable-safe steps. Clearly, sometimes there will be no such possibility and it will be forced to either select an atom whose evaluation performs a non observable-safe step or stop unfolding. In each case, the partial evaluator will treat failing derivations as follows. 1) If all steps are observable-safe, then the failing derivation does not need to be taken into account for code generation, as it is done in traditional PE. 2) In contrast, if it contains one or more steps which are not observable-safe, then if the final goal in the derivation is of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_n$, the partial evaluator has to produce a resultant associated to it of the form $\theta(A) \leftarrow A_1, \ldots, A_{R-1}, fail$, where *fail/0* is a predefined predicate which finitely fails. Note that all atoms to the right of A_R, i.e., $A_{R+1}, \ldots, A_n$ can be safely be removed from the resultant.

5.2 Backpropagation of bindings

The notion of *pure* atom is necessary in order to ensure that backpropagation of bindings does not change the runtime behaviour of the original program.

Definition 6 (pure atom). *An atom A is* pure*, denoted* $\mathsf{pure}(A)$*, if* $\mathsf{observable\_free}(A) \ \wedge \ \mathsf{bind\_ins}(A)$

The notion of *backpropagation-safe* derivation step characterizes the derivation steps in which backpropagation of bindings (and failure) can be safely performed.

Definition 7 (backpropagation-safe derivation step). *In the same conditions as Definition 5, we say that the derivation step for G and C via $\mathcal{R}$ is* backpropagation-safe *if* $\mathsf{pure}(A_1) \wedge \ldots \wedge \mathsf{pure}(A_{R-1})$.

We say that a computation rule $\mathcal{R}$ is *backpropagation-safe* if it always selects atoms in such a way that the derivation step is backpropagation-safe. It is easy to incorporate the idea of backpropagation-safe in a PE system. Note that by definition, leftmost unfolding is always backpropagation-safe. Thus, one simple but very inaccurate policy is to restrict ourselves to leftmost unfolding in the presence of impure predicates. If we would like to use a computation rule which is not always backpropagation-safe, then backpropagation has to be avoided in those steps which are possibly unsafe by using one of the existing proposals (e.g.,[10, 2, 1, 8, 9]).

5.3 Sound Derivations

Finally, we introduce the concept of *sound step* which requires that the selected atom is either user-defined or can be executed (or both), as well as the step be backpropagation-safe. We first present the notion of *evaluable* atom which provides the conditions under which an atom can be executed at specialization time. In order to provide a precise definition in the context of external predicates, we need to introduce first the notion of terminating atom.

Definition 8 (terminating atom). *An atom A is called* terminating*, denoted* $\mathsf{termin}(A)$*, if the LD tree for $\leftarrow A$ is finite.*

The definition above is equivalent to *universal termination*, i.e., the search for all solutions to the atom can be performed in finite time. Note that this condition is not necessary for internal predicates since the unfolding rule incorporates mechanisms for ensuring their termination. If the code of the external predicate was available, we could simply unfold the predicate using the same mechanisms as for internal ones.

Definition 9 (evaluable atom). *An atom A is* evaluable*, denoted* $\mathsf{eval}(A)$*, if* $\mathsf{pure}(A) \wedge \mathsf{termin}(A)$.

The notion of evaluable atoms can be extended in a natural way to boolean expressions composed of conjunction and disjunctions of atoms.

Definition 10 (sound derivation step). *In the same conditions as Definition 5, we say that the derivation step for G and C via $\mathcal{R}$ is* sound *if*

$$\mathsf{pure}(A_1) \wedge \ldots \wedge \mathsf{pure}(A_{R-1})$$

$$\textit{pred}(A_R) \text{ is defined in } P \;\vee\; \mathsf{eval}(A_R)$$

It is important to note that if A_R is an atom for a predicate defined in program P, then no further condition is required on the selected atom itself. As a result, leftmost unfolding of user-defined predicates is always sound, even if the program contains impure predicates. Also, even if the predicate is user-defined, our implementation will fully execute the atom, rather than unfold it, if $\mathsf{eval}(A_R)$ can be guaranteed to hold. This produces important speedups in the PE process.

Our next theorem states that even in the presence of impure predicates, the independence of the computation rule still holds as long as we restrict ourselves to computation rules which are backpropagation-safe.

Theorem 1 (independence of the computation rule). *Let P be program and G a goal. Let $\mathcal{R}$ be a backpropagation-safe computation rule. There is a successful LD derivation for G with c.a. σ iff there is a successful SLD derivation for G via $\mathcal{R}$ with c.a. σ' s.t. $\sigma(G)$ is a variant of $\sigma'(G)$*

The above theorem extends the classical result in logic programming theory for pure programs to impure programs but only for those cases where the computation rule, though it can potentially choose a non-leftmost atom, it will never "jump over" a possibly impure atom.

Also, in the context of impure predicates we are interested in preserving the *observable*s which are generated during the execution of the program.

Definition 11 (observables). *Let P be a program and a G be a goal. Let D be a LD derivation for $P \cup \{G\}$. We define the sequence of* observables *of the derivation D, denoted $\mathcal{O}(D)$, as the sequence of side-effects and errors which occur in D.*

Our unfolding process has to preserve observables both for successful and failing derivations, since otherwise observables would be eliminated from the program.

Theorem 2 (preservation of observables). *Let P be program and G a goal. Let $\mathcal{R}$ be a backpropagation-safe computation rule. There is an LD derivation D for G with $\mathcal{O}(D) \neq \emptyset$ iff there is a SLD derivation D' for G via $\mathcal{R}$ s.t. $\mathcal{O}(D') = \mathcal{O}(D)$.*

Our safety conditions for non-leftmost unfolding preserve computed answers, but has the well-known implication that an infinite failure can be transformed into a finite failure. However, in our framework this will only happen for predicates which do not have side-effects, since non-leftmost unfolding is only allowed in the presence of pure atoms. Nevertheless, our framework can be easily extended to preserve also infinite failure by including termination as an additional property that non-leftmost unfolding has to take into account, i.e. this implies requiring that all atoms to the left of the selected atom should be evaluable and not only pure.

	observable-free			
	pure			
	eval			
predicate	sideeff_free	error_free	bind_ins	termin
var(X)	true	true	nonvar(X)	true
nonvar(X)	true	true	nonvar(X)	true
write(X)	false	true	ground(X)	true
assert(X)	false	false	ground(X)	true
A <= B	true	arithexp(A)∧arithexp(B)	true	true
A >= B	true	arithexp(A)∧arithexp(B)	true	true
ground(X)	true	true	ground(X)	true
A = B	true	true	true	true
append(A,B,C)	true	true	true	list(A)∨list(C)
functor(A,B,C)	true	nonvar(A)∨(atom(B)∧nnegint(C))	true	true
arg(A,B,C)	true	nnegint(A)∧struct(B)	true	true
open(A,B,C)	false	false	ground(C)	true

Fig. 3. Purity conditions for some predefined predicates.

6 Partial Evaluation with Purity Assertions

Though Definition 10 provides conditions under which backpropagation does not need to be hidden, it cannot be used as the basis for an effective PE mechanism, since in general it is not possible to determine at specialization time whether a derivation step is backpropagation-safe or not. In this section, we propose a PE scheme which takes into account purity conditions stated by means of *assertions*. We use the assertion language of `CiaoPP` [15] to provide the concrete syntax of several kinds of assertions. The assertions include *sufficient conditions* (*SC*) which are *decidable* and under which atoms for a predicate are pure. Thus, they can be used as an effective method to guarantee that certain non-leftmost derivation steps are backpropagation-safe.

Example 2. In Figure 3, we present sufficient conditions for a few predefined predicates (builtins) in `Ciao` which guarantee that the atoms for the corresponding predicates satisfy the purity properties discussed in the previous section, where *arithexp(X)* stands for `X` being an arithmetic expression which should be ground at the time of its evaluation, *struct(X)* succeeds iff X is bound to a functor with arity strictly greater than zero, and *nnegint(X)* succeeds iff X is bound to a non-negative integer. For example, unification is pure and evaluable in all circumstances. The library predicate `append/3` is pure but only evaluable if either the first or third argument is bound to a list skeleton. The library predicate `open/3` requires its third argument to be a variable. Thus, backpropagation in this case can introduce errors which would not appear in LD resolution.

Since we consider modular programs, in the following definitions, we have to indicate always the module in which the predicate is defined. We say that the execution of an atom A with $Pred(A) = p/n$ on a logic programming system

Sys (by Sys we mean a Prolog implementation, e.g., `Ciao` or `Sicstus`) in which the module M (where the predicate p/n is defined), together with all modules transitively used by M, have been loaded *trivially succeeds*, denoted by $\mathsf{triv\_suc}(Sys, M, A)$, when the execution of A terminates and succeeds only once with the empty computed answer, that is, it performs no bindings.

Definition 12 (binding insensitive assertion). *Let p/n be a predicate defined in module M. The assertion "*`:- trust comp p(X1,...,Xn):`*SC*`+bind_ins.`*" is a correct* binding insensitive assertion *for predicate p/n in a logic programming system Sys if, $\forall$ A s.t. $A = \theta(p(X_1, \ldots, X_n))$,*

1. $\mathsf{eval}(\theta(SC))$*, and*
2. $\mathsf{triv\_suc}(Sys, M, \theta(SC)) \Rightarrow \mathsf{bind\_ins}(A)$.

The fourth column in Fig. 3 shows the sufficient conditions (SC in Def. 12) stated in several binding insensitive assertions for the predicates in the first column ($p(X1, ..., Xn)$ in Def. 12). For instance, `ground(X)` is a sufficient condition for $\mathsf{bind\_ins}($`write(X)`$)$ to hold.

Given a set of assertions AS and an atom A, we use $\mathsf{bind\_ins}(A, AS)$ to denote that there exists an assertion `:- trust comp p(X1,...,Xn) :` *SC* `+ bind_ins` in AS s.t. $A = \theta(p(X_1, \ldots, X_n))$ and $\mathsf{triv\_suc}(Sys, M, \theta(SC))$.

Definition 13 (error-free assertion). *Let p/n be a predicate defined in module M. The assertion "*`:- trust comp p(X1,...,Xn) :` *SC* `+ error_free.`*" is a correct* error-free assertion *for predicate p/n if, $\forall$ A s.t. $A = \theta(p(X_1, \ldots, X_n))$,*

1. $\mathsf{eval}(\theta(SC))$*, and*
2. $\mathsf{triv\_suc}(Sys, M, \theta(SC)) \Rightarrow \mathsf{error\_free}(A)$.

It should be noted that some builtin predicates can behave in a different way on different systems. In particular, certain calls can fail in a system and issue an error in a different one.

The third column in Fig. 3 illustrates some sufficient conditions for error-freeness for a few predefined predicates. For instance, the SC for predicate `A>=B` states that both arguments should be arithmetic expressions. This guarantees error free calls to predicate `>=/2`.

Given a set of assertions AS and an atom A, we use $\mathsf{error\_free}(A, AS)$ to denote that there exists an assertion `:- trust comp p(X1,...,Xn) :` *SC* `+ error_free` in AS s.t. $A = \theta(p(X_1, \ldots, X_n))$ and $\mathsf{triv\_suc}(Sys, M, \theta(SC))$.

Definition 14 (side-effect free assertion). *Let p/n be an external predicate defined in module M. The assertion* `:- trust comp p(X1,...,Xn) + sideff_free.` *is a correct* side-effect free assertion *for predicate p/n if, $\forall\theta$, the execution of $\theta(p(X1, ..., Xn))$ does not produce any side effect, i.e.,* $\mathsf{sideff\_free}(A)$.

The second column in Fig. 3 shows which predicates are side-effect free. In contrast to the two previous assertions, side-effect assertions are unconditional, i.e., their SC always takes the value true. For brevity, both in the text and in the implementation we omit the SC from them. Let us note that the set of side-effect free atoms is included in the set of error-free atoms, i.e., if A is not a side-effect free atom, then the execution of $\leftarrow A, fail$ is not equivalent to $\leftarrow fail$ and, thus, A is also not side-effect free. Nevertheless, we differentiate side-effects and errors both for conceptual clarity and also because a simple reachability analyses can be used to infer side-effects while errors are more accurately dealt by context-sensitive analyzers.

Given a set of assertions AS and an atom A, we use sideff_free(A, AS) to denote that there exists an assertion `:- trust comp p(X1,...,Xn) + sideff_free` in AS s.t. $A = \theta(p(X_1, \ldots, X_n))$.

Example 3. The following assertions are predefined in `Ciao` for predicate `>=/2`:

```
:- trust comp A >= B : (arithexp(A),arithexp(B)) + error_free.
:- trust comp A >= B + sideff_free.
:- trust comp A >= B + bind_ins.
```

An important thing to note is that rather than using the overall eval assertions (see [14]), we prefer to have separate assertions for each of the different properties required for an atom to be evaluable. However, users can write eval assertions directly if they prefer so. There are several reasons for this. On one hand, it will allow weakening the conditions required for different purposes. For example, binding insensitiveness is not required for avoiding backpropagation of failure. Also, eval assertions include termination which is not required for ensuring correctness w.r.t. computed answers (see Sect. 4) nor termination of internal predicates. Second, it will allow us the use of different analyses for inferring each of these properties (e.g., a simple reachability analysis is sufficient for unconditional side-effects while more elaborated analysis tools are needed for error and binding sensitiveness). Finally, having separate properties will allow reusing such assertions for other purposes different from partial evaluation. For instance, side-effect and error free assertions are also interesting for other purposes (like, e.g., for program verification, for automatic parallelization) and are frequently required by programmers separately.

6.1 Automatic Inference of Purity Assertions

In the case of leftmost unfolding, eval assertions [14] can be used in order to determine whether evaluation of atoms for external predicates can be fully done at specialization time or not. Such eval assertions (or assertions for their constituent properties) should be present whenever possible for all library (including builtin) predicates. Though the presence of such assertions is not required, as the lack of assertions is interpreted as the predicate not being evaluable under any circumstances, the more eval assertions are present for external predicates, the more profitable partial evaluation will be. Ideally, eval assertions can be provided by the system developers and the user does not need to add any eval assertion.

If non-leftmost unfolding is allowed, an important distinction is that pure assertions are of interest not only for external predicates but also for internal, i.e., user-defined predicates. As already mentioned, the lack of pure assertions must be interpreted as the predicate not being pure, since impure atoms can be reached from them. Thus, for non-leftmost unfolding to be able to "jump over" internal predicates, it is required that such pure assertions are available not only for external predicates, but also for predicates internal to the module. Such assertions can be manually added by the user or, much more interestingly, as our system does, by *backwards* analysis [5, 3, 6]. Indeed, we believe that manual introduction of assertions about purity of goals is too much of a burden for the user. Therefore, accurate non-leftmost unfolding becomes a realistic possibility only thanks to the availability of analysis.

Using a simple reachability analysis for error-free and binding-insensitivity assertions would result in very imprecise results, as in other existing approaches. Thus, we would like to perform a context-sensitive analysis which would allow us to determine that some particular contexts guarantee the purity of atoms. The main difficulty with this context-sensitive approach to purity analysis is that it is rather difficult to find out which are the contexts of interest which may appear during a particular PE process. One possibility would be to use a set of representative initial contexts, but this is rather difficult to do, especially for domains with an infinite number of abstract values.

A much more promising approach is based on backwards analysis [5, 3, 6] of logic programs. This kind of analysis has been successfully applied in termination analysis and inference of call patterns which are guaranteed not to produce any runtime error. We propose a novel application of backwards analysis for automatically inferring binding-insensitive, error-free and side-effect free assertions which are useful for improving the accuracy of partial evaluation, as it has been discussed throughout the paper. In our implementation, we rely on the backwards analysis technique of [3]. In this approach, the user first identifies a number of properties that are required to hold at body atoms at specific program points. A meta-program is then automatically constructed, which captures the dependencies between initial goals and the specified program points. For our specific application, we need to observe the occurrences of *all* predicates since the lack of purity assertions must be interpreted as the atom not being pure. Therefore, all program points are subject of analysis. Standard abstract interpretation techniques are applied to the meta-program; from the results of the analysis, conditions on initial goals can be derived which guarantee that all the given properties hold whenever the specified program points are reached. In our particular application, we infer the conditions under which calls to all predicates are pure. The details on how the meta-program is constructed are outside the scope of this paper (see [3]). We simply show by means of an example the kind of information it infers.

Example 4. Consider the purity conditions for predicate `ground/1` in Fig 3 and the program in Fig. 1. Predicate `long_comp/2` is externally defined in module `comp` along with these predefined assertions:

```
:- trust comp long_comp(X,Y) : true + error_free.
:- trust comp long_comp(X,Y) + sideff_free.
:- trust comp long_comp(X,Y) : ground(Y) + bind_ins.
```

For simplicity we consider in this example a simple domain with elements `ground` and `nonground`. Note that our framework can be extended to reason about many other properties like `arithexp`, `list`, etc. by using an abstract domain which captures such information. In particular, we need to include the definitions for the properties we want to capture.

Backwards analysis of the running example and the available assertions (for `long_comp/2` and `ground/1`), infers the following assertions for `problem/2`:

```
:- trust comp problem(X,Y) : true + error_free.
:- trust comp problem(X,Y) + sideff_free.
:- trust comp problem(X,Y) : ground(Y) + bind_ins.
```

The last assertion indicates that calls performed to `problem(X,Y)` with the second argument being ground are binding insensitive. This allows our specializer to "jump over" the call to problem and backpropagate bindings, which will in turn trigger further unfolding.

6.2 Combining Assertions with Partial Evaluation

We now provide an extension of the definition of safe derivation which takes into account the purity conditions in our assertions. We use $\mathsf{pure}(A, AS)$ to denote $\mathsf{bind\_ins}(A, AS) \wedge \mathsf{error\_free}(A, AS) \wedge \mathsf{sideff\_free}(A, AS)$.

Definition 15 (backpropagation-safe derivation step w.r.t. assertions). *Let AS be a correct set of assertions. Let P be a program, let $G = \leftarrow A_1, \ldots, A_n$ be a goal and let $\mathcal{R}$ be a computation rule s.t $\mathcal{R}(G) = A_R$. Let C be a renamed apart clause in P s.t. the head of C unifies with A_R. We say that the derivation step for G and C via $\mathcal{R}$ is* backpropagation-safe *w.r.t. AS if* $\mathsf{pure}(A_1, AS) \wedge \ldots \wedge \mathsf{pure}(A_{R-1}, AS)$.

In order to integrate the above notion in an unfolding rule, the same ideas sketched in Sect. 5.3 apply here. We also give the corresponding definition for sound derivation based on purity assertions.

Definition 16 (sound derivation step w.r.t. assertions). *In the same conditions as Definition 7, we say that the derivation step for G and C via $\mathcal{R}$ is* sound *w.r.t. AS if*

$$\mathsf{pure}(A_1, AS) \wedge \ldots \wedge \mathsf{pure}(A_{R-1}, AS)$$
$$pred(A_R) \text{ is defined in } P \ \vee \ \mathsf{eval}(A_R, AS)$$

An important difference of the above definition w.r.t Definition 10 is that the former is *effective* since the sufficient conditions provided by assertions can effectively be used at specialization time in order to determine that certain atoms are pure. This in turn will allow performing backpropagation of bindings and failure for non-leftmost unfolding steps under circumstances where existing techniques would need to resort to not backpropagating.

Similar theorems to Theorem 1 and Theorem 2 can be enunciated which guarantee the correctness of derivation steps performed using a computation rule which is backpropagation-safe with respect to a set of correct purity assertions.

Example 5. Consider a deterministic unfolding rule which only performs sound derivation steps. In our running example, it performs an initial step and derives the goal `problem(X,a),q(X)`. Now, it cannot select the atom `problem(X,a)` because its execution performs a non-deterministic step. Fortunately, the assertions inferred for `problem(X,Y)` in Ex. 4 allow us to jump over this atom and specialize first `q(X)`. In particular, the first two assertions, since their SC is `true`, guarantee that there is no problem related to errors or side-effects. From the last assertion, we know that the above call is binding insensitive, since the condition "`ground(a)`" trivially succeeds. If atom `q(X)` is evaluated first, then variable `X` gets instantiated to `a`. Now, the unfolding rule already can select the deterministic atom `problem(a,a)` and obtain the fact " `main(a,a).`" as partially evaluated program. The interesting point to note is that, without the help of assertions, the derivation is stopped when the atom `problem(X,a)` is selected because any call to `problem` is considered potentially dangerous since its execution reaches a binding sensitive predicate. The equivalent specialized rule in this case is: "`main(X,a):-problem(X,a),q(X).`" A detailed explanation on the improvements achieved by our specialized program is provided in the three points enumerated in Sect. 1.

7 Conclusions

We have presented a practical partial evaluation scheme for full Prolog programs with impure predicates. As it is well known, impure features pose non-trivial challenges in the context of non-leftmost unfolding in partial evaluation. Existing (more conservative) approaches avoid backpropagating bindings and failure in the presence of such problematic predicates at the cost of accuracy. However, under certain conditions, calls to apparently impure predicates in reality are pure and thus backpropagation can be safely performed onto them. Our proposal is more accurate in that the partial evaluator takes into account purity conditions (stated by means of assertions) in order to decide whether backpropagation during non-leftmost unfolding is safe. Thanks to the use of backwards analysis, correct and precise sufficient conditions can be automatically inferred for all predicates from a set of predefined assertions available in the system. Our approach has been successfully integrated in the context of `CiaoPP`, the analysis/specialization preprocessor of the `Ciao` logic programming system, in which we have available a full assertion language and a number of analyzers. As for future work, we plan to exploit our automatically inferred assertions for purity in an abstract partial evaluation framework, where we can prove that certain backpropagations are safe using a combination of sharing analysis with refined notions of independence.

Acknowledgments

This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 *ASAP* project and by the Spanish Ministry of Science and

Education under the MCYT TIC 2002-0055 *CUBICO* project. Part of this work was performed during a research stay of Elvira Albert and Germán Puebla at University of Roskilde supported by respective grants from the Secretaría de Estado de Educación y Universidades, Spanish Ministry of Science and Education. J. Gallagher's research is supported in part by the IT-University of Copenhagen.

References

1. E. Albert, M. Hanus, and G. Vidal. A practical partial evaluation scheme for multi-paradigm declarative languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.
2. S. Etalle, M. Gabbrielli, and E. Marchiori. A Transformation System for CLP with Dynamic Scheduling and CCP. In *Proc. of the ACM Sigplan PEPM'97*, pages 137–150. ACM Press, New York, 1997.
3. J. Gallagher. A Program Transformation for Backwards Analysis of Logic Programs. In Proc. of *LOPSTR 2003*, number 3018 in LNCS, pages 92–105. Springer-Verlag, 2004.
4. J.P. 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.
5. Jacob M. Howe, Andy King, and Lunjin Lu. Analysing Logic Programs by Reasoning Backwards. *Program Development in Computational Logic*, LNCS, pages 380–393. Springer-Verlag, May 2004.
6. A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, 2(4–5):32, July 2002.
7. J. Komorovski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta Programming in Logic, Proceedings of META'92*, volume 649 of *LNCS*, pages 49–69. Springer-Verlag, 1992.
8. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
9. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in prolog using a hand-written compiler generator. *TPLP*, 4(1–2):139 – 191, 2004.
10. Michael Leuschel. Partial evaluation of the "real thing". In *Proc. of LOPSTR'94 and META'94*, LNCS 883, pages 122–137. Springer-Verlag, 1994.
11. Michael Leuschel. On the power of homeomorphic embedding for online termination. Proc. of SAS'98, LNCS 1503, pages 230–245, 1998. Springer-Verlag.
12. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.
13. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
14. G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In Proc. of *LOPSTR'04*, number 3573 in LNCS, pages 149–165. Springer-Verlag, June 2005.
15. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.
16. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
17. R. Venken and B. Demoen. A partial evaluation system for prolog: some practical considerations. *New Generation Computing*, 6:279–290, 1988.