A Partial Deducer Assisted by Predefined Assertions and a Backwards Analyzer

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] School of Computer Science, University of Roskilde, jpg@ruc.dk

Abstract. Partial deduction is a program transformation technique which specializes a program w.r.t. its static data. If the program contains *impure* predicates, it is known that unfolding steps for atoms which are not leftmost is problematic. Impure predicates include those which may raise errors, exceptions or side-effects, external predicates whose definition is not available, etc. Existing proposals allow obtaining correct residual programs while still allowing non-leftmost unfolding steps, but at the cost of accuracy: bindings and failure are not propagated backwards to predicates which are classified as impure. Motivated by recent developments in the *backwards* analysis of logic programs, we propose a partial deduction algorithm which can handle impure features and non-leftmost unfolding in a more accurate way. We outline by means of examples some optimizations which are not feasible using existing partial deduction techniques. We argue that our proposal goes beyond existing ones and 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 which can guide the partial deduction process, without having to modify the partial deducer 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 Background

We assume some basic knowledge on the terminology of logic programming. See for example [16] for details. Very briefly, an *atom* A is a syntactic construction of the form $p(t_1, \ldots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \ldots, t_n$ are terms. The function *pred* applied to atom A, i.e., $pred(A)$, returns the predicate symbol p/n for A. 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 *definite 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 definite 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 unfolding as LD resolution.

Partial Deduction (PD) [15, 8] is a program transformation technique which specializes a program w.r.t. part of its known input data. Hence sometimes also known as program specialization. Informally, given an input program and a set of atoms, the PD 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. Each unfolding step during partial deduction 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.

Most of real-life Prolog programs use predicates which are not defined in the program (module) being developed. We will refer to such predicates as *external*. Examples of external predicates are traditional "built-in" predicates such as arithmetic operations (e.g., `is/2`, `<`, `=<`, etc.), basic input/output facilities, and predicates defined in libraries. We will also consider as external predicates those defined in a different module, predicates written in another language, etc. 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 PD of full Prolog programs. Such issues are discussed in depth throughout this extended abstract. When a (leftmost) atom A_R is selected during PD, 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 the 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 PD 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 [17] captures the requirements which allow executing external predicates at PD 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 binding insensitive. We use $\mathsf{eval}(E)$ to denote that the expression E is evaluable. We will discuss all these properties in depth in Section 3.

2 Non-Leftmost Unfolding in Partial Deduction

It is well-known that *non-leftmost* unfolding is essential in partial deduction in some cases for the satisfactory propagation of static information (see, e.g., [14]). Informally, given a goal $\leftarrow A_1, \ldots, A_n$, it can happen that the profitable criterion does not hold for the leftmost atom A_1. For example, if A_1 is an atom for an internal predicate, it might not be profitable to select A_1 because 1) unfolding A_1 endangers termination (for example, A_1 may homeomorphically embed [13] some selected atom in its sequence of covering ancestors), or 2) the atom A_1 unifies with several clause heads (for example, some unfolding rules do not unfold non-deterministically for atoms other than the initial query). If A_1 is an atom for an external predicate, it can happen that A_1 is not sufficiently instantiated so as to be executed at this moment. It may nevertheless be profitable to unfold atoms other than the leftmost. Therefore, it can be interesting to define a computation rule which is able to detect the above circumstances and "jump over" atoms whose profitability criterion is not satisfied in order to proceed with the specialization of another atom in the goal as long as it is correct.

2.1 Non-Leftmost Unfolding and Impure Predicates

For pure logic programs without builtins, non-leftmost unfolding is safe thanks to the independence of the computation rule (see for example [16]).[4] 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, `var/1` is an *impure* predicate since, under LD resolution, the goal `var(X),X=a` succeeds with computed answer X/a whereas `X=a,var(X)` fails. They are not equivalent since the independence of the computation rule does not hold. Thus, given the goal $\leftarrow$ `var(X),X=a`, if we allow the non-leftmost unfolding step which binds the variable `X`, the goal will fail, either at specialization time or at run-time, whereas the initial goal succeeds in LD resolution. The above problem was early detected [18] and it is known as the problem of *backpropagation of bindings*. In addition to this, it is also problematic the *backpropagation of failure* in the presence of impure predicates. There are atoms A for impure predicates such that $\leftarrow A, fail$ behaves differently from $\leftarrow fail$. For instance,

[4] Although safe, non-leftmost unfolding presents problems with pure programs too since it may introduce extra backtracking over the atoms to the left. We are not concerned with such efficiency issues here.

we have to ensure that failure to the right of a call to `write` does not prevent the generation of the residual call to `write` nor its execution at runtime.

There are satisfactory solutions in the literature (see, e.g.,[11, 4, 1, 14]) which allow unfolding non-leftmost atoms while avoiding the backpropagation of bindings and failure. Basically, the common idea is to represent explicitly the bindings by using unification [11] or residual case expressions [1] rather than backpropagating them (and thus applying them onto leftmost atoms). 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. It should be noted that preventing backpropagation by introducing equalities can be a bad idea from the performance point of view too (see, e.g., [19]). Thus, these solutions should be applied only when it is really necessary, since backpropagation can 1) lead to early detection of failure, which may result in important speedups and 2) make the profitability criterion for the leftmost atom to hold, which may result in more aggressive unfolding. Thus, if backpropagation is disabled, some interesting specializations can no longer be achieved.

It should also be noted that the backpropagation problem is very much related to that of *reordering* of atoms within a goal. Such reordering transformation can be of interest for achieving powerful optimizations like tupling, for effectively handling the conjunction of atoms like conjunctive PD [3] and for the use of efficient stack-based unfolding rules [17].

3 From Impure Predicates to Impure Atoms

As mentioned in Section 2.1 above, existing techniques for PD 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 [14] are based on simple reachability analysis. As soon as an impure predicate p can be reached from a predicate q, also q is considered impure and backpropagation is not allowed. In other words, impurity is defined at the level of predicates. Unfortunately, this notion of impurity quickly expands from a predicate to all predicates which use it.

Our work improves on existing techniques by providing a more refined notion of impurity. Rather than being defined at the level of predicates, we define purity at the level of individual atoms. This 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). This allows *reducing* substantially the situations in which backpropagation has to be avoided. In the following, we characterize three different classes of impurities: binding-sensitiveness, errors and side effects.

3.1 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, ...`. However, rather than considering all atoms for such predicates as binding-sensitive, we propose to define binding sensitiveness at the atom level. The reason is that the fact that some atoms for the predicates above are indeed binding sensitive does not necessarily mean that all atoms for such predicates are. As we have seen above, the atom $var(f(X))$ is certainly not binding sensitive since its truth value is not changed by applying any substitution, i.e., the atom will not succeed in any context.

Definition 1 (binding insensitive atom). *An atom A is* binding insensitive*, denoted* bind_ins(A)*, if $\forall$ sequence of 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. More on this in Section 5.2.

If the atom contains no variables, binding insensitiveness trivially holds. The following proposition directly follows from the definition of binding insensitive atom.

Proposition 1. *Let A be a ground atom. Then A is binding insensitive.*

In spite of its simplicity, Proposition 1 can be quite useful in practice, since it may allow considering a good number of atoms as binding insensitive even if the predicate is in principle binding sensitive. All this without the need of sophisticated analyses.

3.2 Side-effects

Predicates p for which $\theta(p(X1, ..., Xn)), fail$ and $fail$ are not equivalent in LD resolution are termed as "*side-effects*" in [18].

Definition 2 (side-effect-free atom). *An atom A is* side-effect free*, denoted* 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, hides the existing side-effect.

3.3 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 partial deduction produces 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). The following definition introduces the notion of *error free* atom.

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 surprising this condition for PD corresponds to that used in [10] 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 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-effects at the level of predicates, whereas a more refined, atom-based classification will be used in the case of error-freeness.

3.4 Pure and Evaluable Atoms

Given the definitions of binding insensitive, side-effect free, and error free atoms, it is useful to define aggregate properties which summarize the effect of such individual properties.

Definition 4 (pure atom). *An atom A is* pure*, denoted* $\mathsf{pure}(A)$*, if*

$$\mathsf{bind\_ins}(A) \wedge \mathsf{error\_free}(A) \wedge \mathsf{sideff\_free}(A)$$

	pure			
	eval			
predicate	sideff_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	nonvar(X)	ground(X)	true
A is B	true	arithexp(B)	ground(B)	true
A <= B	true	arithexp(A)∧arithexp(B)	ground(A)∧ground(B)	true
A >= B	true	arithexp(A)∧arithexp(B)	ground(A)∧ground(B)	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)

Fig. 1. Purity conditions for some predefined predicates.

In order to provide a precise definition of evaluable atom, we need to introduce first the notion of terminating atom.

Definition 5 (terminating atom). *An atom A is* 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.

Definition 6 (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.

Figure 1 presents sufficient conditions which guarantee that the atoms for the corresponding predicates satisfy the purity properties discussed above, where *arithexp(X)* stands for `X` being an arithmetic expression. For example, unification is pure and evaluable, whereas the library predicate `append/3` is pure but only evaluable if either the first or third argument is bound to a list skeleton.

4 Assertions about Purity of Atoms

In this section, we provide the concrete syntax of the assertions we propose to use to state the conditions under which atoms for a predicate are pure. Our assertions may include *sufficient conditions* (SC) which are *decidable* and ensure that, if the atom satisfies such conditions, then it meets the property.

We say that the execution of an atom A for p/n on a logic programming system Sys (e.g., `Ciao` or Sicstus) in which the module M (where the external predicate p/n is defined) has been loaded *trivially succeeds*, denoted by $\mathsf{triv\_suc}(Sys, M, A)$, when its execution terminates and succeeds only once with the empty computed answer, that is, it performs no bindings.

Definition 7 (binding insensitive assertion). *Let p/n be a predicate defined in module M. The assertion "*`:- trust comp p(X1,...,Xn) : SC + bind_ins.`*" in the code for M 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. 1 comprises the information stated in several binding insensitive assertions for a few predefined builtins in `Ciao`. In particular, this column represents the sufficient conditions (SC in Def. 7) for the predicates in the first column ($p(X1, ..., Xn)$ in Def. 7). For instance, the predicate `A is B` is bind_ins if `ground(B)`.

Definition 8 (error-free assertion). *Let p/n be a predicate defined in module M. The assertion "*`:- trust comp p(X1,...,Xn) : SC + error_free.`*" in the code for M is a correct* error-free 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. $\mathsf{triv\_suc}(Sys, M, \theta(SC)) \Rightarrow \mathsf{error\_free}(A)$.

For instance, the SC for predicate `is/2` states that the second argument is an arithmetic expression. This condition guarantees error free calls to predicate `is/2`.

Definition 9 (side-effect free assertion). *Let p/n be an external predicate defined in module M. The assertion* `:- trust comp p(X1,...,Xn) + sideff_free.` *in the code for M is a correct* side-effect free assertion *for predicate p/n in a logic programming system Sys if, $\forall\theta$, the execution of $\theta(p(X1, ..., Xn))$ does not produce any side effect.*

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.

Example 1. The following assertions are predefined in `Ciao` for predicate `ground/1`:

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

It can be seen that the third assertion for predicate `ground/1` is indeed redundant, since by Proposition 1 we already know that any atom which is ground is binding insensitive.

An important thing to note is that rather than using the overall eval assertions of [17], we prefer to have separate assertions for each of the different properties required for an atom to be evaluable. There are several reasons for this. On one hand, it will allow us the use of separate analysis for inferring each

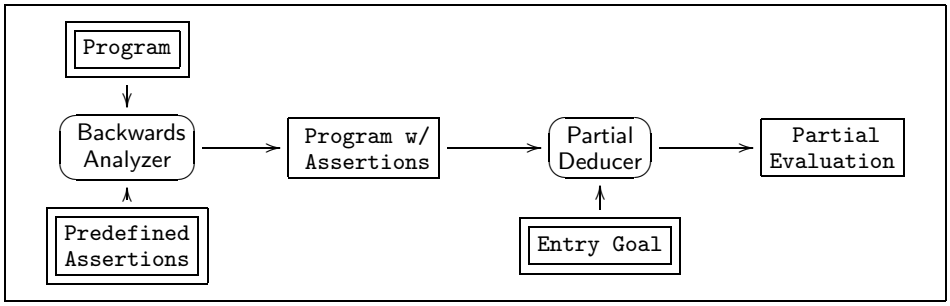

Fig. 2. Backwards Analysis in Non-leftmost Partial Deduction

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). Also, it will allow reusing such assertions for other purposes different from partial deduction. 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. Finally, eval assertions include termination which is not required for ensuring correctness w.r.t. computed answers (see Sect. 3).

5 Automatic Inference of Assertions by Backwards Analysis

Recent developments in backwards analysis of logic program [9, 7, 10] have pointed out novel applications in termination analysis and inference of call patterns which are guaranteed not to produce any runtime error. In this section, we outline a new application of backwards analysis for automatically inferring binding insensitive, error free and side-effect free annotations which are useful to this purpose. Automatically figuring out when a substitution can be safely backpropagated onto a call whose execution reaches an impure predicate has been considered a difficult challenge and, to our knowledge, no accurate, satisfactory solution exists.

Fig. 2 illustrates the PD scheme based on assertions and backwards analysis that we have implemented in CiaoPP. Initially, given a Program and a set of Predefined Assertions for the external predicates, the Backwards Analyzer obtains a Program w/ Assertions which includes error_free, sideff_free and bind_ins assertions for all user predicates. Notice that this is a goal-independent process which can be started in our system regardless PD being performed or not. Afterwards, 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 Goal. A Partial Deducer 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.

5.1 The Backwards Analyzer

Regarding the analyzer, we rely on the backwards analysis technique of [7]. 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. This meta-program is based on the *resultants* semantics of logic programs [6, 5], in which the meaning of a program is the set of all pairs (A, R) where $A = A'\theta$ and there is an LD derivation from $\leftarrow A'$ to $\leftarrow R$ with computed answer θ. An abstraction of the resultants semantics is then defined, containing all pairs (A, B) such that $A = A'\theta$ and there is an LD derivation from $\leftarrow A'$ to $\leftarrow B, B_1, \ldots, B_m$ with computed answer θ, where B corresponds to one of the specified program points. (This semantics is closely related to the binary clause semantics defined by Codish and Taboch [2]). The semantics is captured by a meta-program defining a meta-predicate `d/2`, such that `d(A,B)` is a consequence of the meta-program whenever a pair (A, B) as defined above exists. 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.

As indicated in Fig. 2, the analyzer starts from a program and an initial set of assertions which state the properties of interest defined in Sect. 2 for the external predicates. Essentially, the analysis algorithm propagates this information backwards in order to get the appropriate assertions for all predicates. The next example illustrates the use of backwards analysis to derive binding-insensitive assertions for an exported predicate, starting from the assertions on its imported predicates.

Example 2. Consider the predicate `vars/2` which computes the set of variables in a term, given in Figure 3.

There are several binding-sensitive predicates in the program, namely `var/1`, `atomic/1`, `nonvar/1`, `\==` and `==`. We can give assertions for each of these, indicating the conditions under which they are binding-insensitive, as follows:

```
:- trust comp var(X) : nonvar(X) + bind_ins.
:- trust comp nonvar(X) : nonvar(X) + bind_ins.
:- trust comp atomic(X) : nonvar(X) + bind_ins.
:- trust comp X==Y : ground(X), ground(Y) + bind_ins.
:- trust comp X\==Y : ground(X), ground(Y) + bind_ins.
```

After performing a backwards analysis with respect to the occurrences of these predicates, over the abstract domain `{ground, nonground}`, we obtain the following model for the meta-predicate `d/2`.

```
d(vars(A,ground),\==(A,ground)),
d(vars(A,ground),==(A,ground)),
d(vars(A,nonground),\==(A,ground)),
d(vars(ground,A),\==(ground,ground)),
```

```
:- module(vars, [vars/2]).

vars(T,Vs) :- vars3(T,[],Vs).

vars3(X,Vs,Vs1) :- var(X), insertvar(X,Vs,Vs1).
vars3(X,Vs,Vs) :- atomic(X).
vars3(X,Vs,Vs1) :- nonvar(X), X =.. [_|Args], argvars(Args,Vs,Vs1).

argvars([],Q,Q).
argvars([X|Xs],Vs,Vs2) :- vars3(X,Vs,Vs1), argvars(Xs,Vs1,Vs2).

insertvar(X,[],[X]).
insertvar(X,[Y|Vs],[Y|Vs]) :- X == Y.
insertvar(X,[Y|Vs],[Y|Vs1]) :- X \== Y, insertvar(X,Vs,Vs1).
```

Fig. 3. The `vars/2` procedure

```
d(vars(ground,A),==(ground,ground)),
d(vars(A,ground),atomic(A)),
d(vars(ground,A),atomic(ground)),
d(vars(A,B),var(A)),
d(vars(A,B),nonvar(A)),
d(vars(nonground,A),\==(B,C)),
d(vars(nonground,A),==(B,C)),
d(vars(nonground,A),var(B)),
d(vars(nonground,A),nonvar(B)),
d(vars(nonground,A),atomic(B))
```

It can automatically be deduced from these facts that whenever `vars(X,Y)` is called with `X` ground, then all the conditions for binding-insensitivity are satisfied (noting that `ground(X)` implies `nonvar(X)`). Thus we can export the assertion on binding-insensitivity of `vars/2`.

```
:- trust comp vars(X,Y) : ground(X) + bind_ins.
```

We next consider a small example (continued in Ex. 4) illustrating how backwards analysis can assist non-leftmost unfolding .

Example 3. Consider the predefined assertions in `Ciao` for predicate `ground/1` of Ex. 1 and the `Ciao` program in Fig. 4 whose modular structure appears to the right. `term_typing` is the name of the module in `Ciao` where `ground/1` is defined (and thus where the assertions for `ground/1` are).

Predicate `long_comp/2` is externally defined in module `comp` where also these predefined assertions for it are:

```
:- 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.
```

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

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

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

q(a).
```

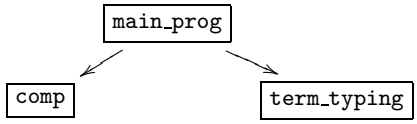

Fig. 4. Program from Example 3

From the program and the available assertions (for `long_comp/2` and `ground/1`), the backwards analyzer 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.
```

Backwards analysis of the above program, with analysis over a simple domain with elements `ground` and `nonground`, yields the following dependencies, represented using the meta-predicate `d(A,B)` described above.

```
d(problem(X,ground), long_comp(ground,ground)).
d(problem(X,nonground), long_comp(ground,nonground)).
```

These facts imply that whenever a call `problem(X,Y)` is made where `Y` is ground, any subsequent assertions concerning binding insensitivity are satisfied; specifically, calls to `long_comp(X,Y)` satisfy the assertion `ground(Y)`. Hence the last assertion (binding insensitivity) on `problem(X,Y)` is established. The analysis results for `d/2` also clearly establish first two assertions on `problem(X,Y)`, with condition `true`, since any call to `problem(X,Y)` is guaranteed to satisfy all the (trivial) error-freeness and side-effect-freeness assertions.

The last assertion indicates that calls performed to `problem(X,Y)` with the second argument being ground are not binding sensitive. This will be very useful information for the specializer.

5.2 The Partial Deducer

In our system, we use a standard partial deducer (like, e.g., the ECCE system [12]), with the notable difference of using a *observable-preserving unfolding rule*. The following definition introduces this idea.

Definition 10 (observable-preserving unfolding rule). *Let AS be a set of correct assertions. We say that an unfolding rule is* observable-preserving *w.r.t. AS if, for any goal* $\leftarrow G_1, \ldots, G_n$*, it always selects an atom* G_k *for unfolding with* $k = 1, \ldots, n$ *such that all atoms* $G_1, \ldots, G_{k-1}$ *are binding insensitive, error free and side-effect w.r.t. AS.*

The above definition allow us to ensure that our PD scheme is *correct* in the sense that the partially evaluated program preserves the runtime behaviour (or observables) of the original one w.r.t. the predefined assertions. Let us see an example.

Example 4. Consider a deterministic unfolding rule (i.e., an unfolding rule which cannot perform non-deterministic steps other than the first one). Given the program of Ex. 3 and the entry goal: " `:- entry main(X,a).`" The unfolding rule 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. 3 allow us to jump over this atom and specialize first `q(X)`. In particular, the first two assertions do not pose any restriction because their conditions are `true`, thus, 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 ot note is that, without the use 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 specialized program in this case is:

```
main(X,a):-problem(X,a),q(X).
```

Intuitively, this residual program is much less efficient than our specialization since the execution of the call to `long_comp` has been totally performed at PD time in our program while it remains residual in the above one.

As already mentioned in Section 1, 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 avaluable and not only pure (see Section 3.4).

6 Conclusions

In the case of leftmost unfolding, eval assertions can be used in order to determine whether evaluation of atoms for external predicates can be fully evaluated at specialization time or not. Such eval assertions 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 deduction 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, the following conditions are required: given a goal $\leftarrow A_1, \ldots, A_R, \ldots, A_n$, backpropagation of bindings and failure for the execution of A_R is only allowed if $\mathsf{pure}(A_1) \wedge \ldots \wedge \mathsf{pure}(A_{R-1})$. An important distinction w.r.t. the case of leftmost unfolding above 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. 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 backwards analysis.

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. Michael Codish and Cohavit Taboch. A semantic basic for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
3. Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.
4. 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.

5. Maurizio Gabbrielli and Roberto Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the 1994 ACM Symposium on Applied Computing, SAC 1994*, pages 394 – 399, 1994.
6. Maurizio Gabbrielli, Giorgio Levi, and Maria Chiara Meo. Resultants semantics for Prolog. *Journal of Logic and Computation*, 6(4):491–521, 1996.
7. J. Gallagher. A Program Transformation for Backwards Analysis of Logic Programs. In *Logic Based Program Synthesis and Transformation: 13th International Symposium, LOPSTR 2003*, number 3018 in LNCS, pages 92–105. Springer-Verlag, 2004.
8. 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.
9. Jacob M. Howe, Andy King, and Lunjin Lu. Analysing Logic Programs by Reasoning Backwards. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic*, LNCS, pages 380–393. Springer-Verlag, May 2004.
10. A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, page 32, July 2002. (Theory and Practice of Logic Programming was formally known as The Journal of Logic Programming).
11. Michael Leuschel. Partial evaluation of the "real thing". In Laurent Fribourg and Franco Turini, editors, Logic Program Synthesis and Transformation — Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, Lecture Notes in Computer Science 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.
12. Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
13. Michael Leuschel. On the power of homeomorphic embedding for online termination. In Giorgio Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.
14. Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
15. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.
16. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
17. G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *14th International Symposium on Logic-based Program Synthesis and Transformation*, LNCS. Springer-Verlag, 2005. To appear.
18. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
19. R. Venken and B. Demoen. A partial evaluation system for prolog: some practical considerations. *New Generation Computing*, 6:279–290, 1988.