

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

Elvira Albert*, **Germán Puebla**** and **John Gallagher*****

() Complutense University of Madrid (Spain)*

*(**) Technical University of Madrid (Spain)*

*(***) University of Roskilde (Denmark)*

INTERNATIONAL SYMPOSIUM ON LOGIC-BASED PROGRAM
SYNTHESIS AND TRANSFORMATION (LOPSTR'05)

London, September 7-9, 2005

Introduction

- Non-leftmost unfolding poses problems in the context of *full* Prolog programs with *impure* predicates (independence does not hold).
 - ▶ `ground/1` is *impure* since, under LD resolution, `ground(X), X=a` fails whereas `X=a, ground(X)` succeeds with c.a. `X/a`.
- backpropagation of bindings: given $\leftarrow \text{ground}(X), X=a$, if we allow the non-leftmost unfolding step which binds `X`, it will succeed at PE time, whereas it fails in LD resolution at run-time.
- backpropagation of failure is also problematic in the presence of impure predicates: $\leftarrow \text{write}(\text{hello}), \text{fail}$ behaves differently from $\leftarrow \text{fail}$.

Introduction

- It is well-known that *non-leftmost* unfolding is essential in PE in some cases for the satisfactory propagation of static information.
 - Given a program P and a goal $\leftarrow A_1, \dots, A_n$, it can happen that A_1 cannot be selected for unfolding due to:
 - ① If A_1 is an atom for a predicate defined in P , it can happen
 - ★ unfolding A_1 endangers termination (for example, A_1 may embed some selected atom),
 - ★ A_1 unifies with several clause heads (deterministic unfolding rules only unfold non-deterministically initial query)
 - ② If A_1 is an atom for an external predicate (code not available to the partial evaluator,) it can happen that it is not sufficiently instantiated so as to be executed.
- ⇒ It may be profitable to unfold non-leftmost atoms.
- ▶ Computation rule which is able to detect the above circumstances and “jump over” those problematic atoms.
 - ▶ Proceed with the specialization of another atom in the goal as long as it is correct.

Motivation: Running Example

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

- Consider a deterministic unfolding rule which 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...

Motivation: Running Example

- Given the goal $\text{problem}(X, a)$, $q(X)$
 - (a) We can stop unfolding at this point \Rightarrow poor specialization.
 - ▶ “Jump over” the problematic atom in order to proceed with the specialization of $q(X)$
 - (b) Unfold $q(X)$ but avoiding backpropagating bindings and failure onto $\text{problem}(X, a) \Rightarrow$ poor specialization.
 - (c) Unfold $q(X)$ while allowing backpropagation onto $\text{problem}(X, a) \Rightarrow$ optimal specialization.
 - However, (c) will require that some additional conditions 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) is applicable and
 - ▶ build a PE system which can effectively prove such conditions in order to perform backpropagation of bindings and failure as much as possible.

Benefits of Backpropagation

- Several solutions in the literature allow unfolding non-leftmost atoms by representing explicitly the bindings using unification rather than backpropagating them.
 - ▶ we can unfold $q(X)$ and obtain the resultant $\text{main}(X, a) :- \text{problem}(X, a), X=a$
- This guarantees correctness, but it introduces some inaccuracy, since backpropagation of bindings and failure can lead to:
 - ▶ early detection of failure
 - ★ we get rid of the whole (failing) computation for $\text{problem}(b, a)$
 - ▶ more aggressive unfolding
 - ★ the (deterministic) atom $\text{problem}(a, a)$ allows more unfold
 - ▶ improved indexing by further instantiating arguments in clause heads
 - ★ the clause head $\text{main}(a, a)$ has more indexing than $\text{main}(X, a)$
- Backpropagation should be avoided only when it is really necessary

Existing Methods

- **Challenge**: Automatically figuring out when bindings and/or failure can be safely backpropagated
- **Existing methods**: based on simple reachability analysis.
 - ▶ As soon as an impure predicate p can be reached from a predicate q , also q is considered impure
 - ▶ Since there is a call to an impure predicate within `problem/2`, backpropagation of the binding for X will be avoided
- Our work improves on existing techniques by:
 - 1 providing a finer-grained notion of impurity (at the level of *atoms* rather than *predicates*) and
 - ★ $var(X)$ is impure (binding sensitive), whereas $var(f(X))$ is not
 - 2 splitting the notion of purity into its constituent properties:
 - ★ binding-sensitiveness, errors, side effects.
 - 3 characterizing soundness conditions for backpropagation

From Impure Predicates to Impure Atoms

binding-sensitiveness

- *binding-sensitive predicate*: different success or failure behaviour under leftmost execution if bindings are backpropagated onto it.
- Examples: `var/1`, `nonvar/1`, `ground/1`, etc.
- `bind_ins(A)`: if $\leftarrow (X = t, A)$ succeeds in LD resolution with c.a. σ iff $\leftarrow (A, X = t)$ also succeeds in LD resolution with σ .

side-effects

- side-effects have to be preserved in the residual program (we have to avoid any kind of backpropagation which can anticipate failure)
- Examples: `write/1` and `assert/1`
- `sideff_free(A)`: if run-time behaviour of $\leftarrow A$, *fail* is equivalent \leftarrow *fail*.

errors

- Example: `is/2` requires its second argument to be arithmetic expression or error is issued.
- `error_free(A)`: if the execution of A does not issue any error.

Backpropagation of Failure

- Aggregate properties summarize the effect of such individual properties and define soundness conditions under which backpropagation of bindings and failure is correct.
- An atom A is `observable-free`, denoted `observable_free(A)`, if `error_free(A) ∧ sideff_free(A)`
- We say that the derivation step for $\leftarrow A_1, \dots, \underline{A_R}, \dots, A_k$ (where A_R is the selected atom for evaluation) is `observable-safe` if `observable_free(A_1) ∧ ... ∧ observable_free(A_{R-1})`.
- Simple integration in a PE system: the computation rule used for unfolding rule should select first those atoms whose evaluation gives rise to observable-safe steps.

Backpropagation of Bindings

- The notion of `pure atom` ensures that backpropagation of bindings does not change the runtime behaviour of original program.
- An atom A is *pure*, denoted $\text{pure}(A)$, if $\text{observable_free}(A) \wedge \text{bind_ins}(A)$
- The derivation step for $\leftarrow A_1, \dots, \underline{A_R}, \dots, A_k$ is `backpropagation-safe` if $\text{pure}(A_1) \wedge \dots \wedge \text{pure}(A_{R-1})$.
- Leftmost unfolding is always backpropagation-safe
- 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 existing proposals

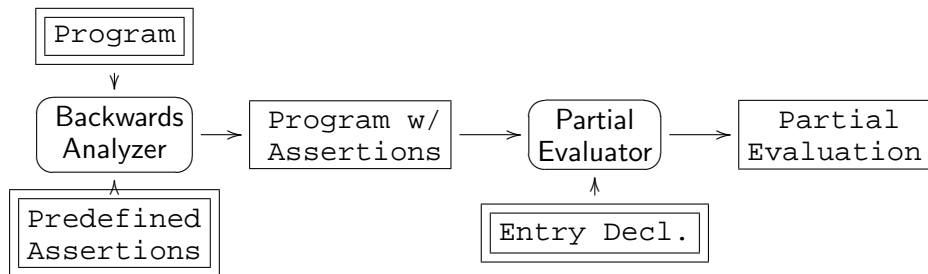
Sound Derivations

- The concept of *sound step* requires the selected atom to be either user-defined or can be *executed* + backpropagation-safe
- The notion of *evaluable* atom, denoted $\text{eval}(A)$, requires $\text{pure}(A) \wedge \text{termin}(A)$ (conditions under which an atom can be executed at specialization time).
- Derivation step for $\leftarrow A_1, \dots, \underline{A_R}, \dots, A_k$ is sound if

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

- independence of the computation rule Let \mathcal{R} be a sound 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'
- preservation of observables There is an LD derivation D for G with $\mathcal{O}(D)$ iff there is a SLD derivation D' for G via \mathcal{R} s.t. $\mathcal{O}(D') = \mathcal{O}(D)$.

Partial Evaluation with Purity Assertions



- It is not possible to determine at specialization time whether a derivation step is backpropagation-safe or not.
- PE scheme which takes into account *decidable* purity conditions stated by means of *assertions*.
- `Backwards Analyzer` automatically infers, from the predefined assertions, sufficient conditions under which atoms are pure.

Sufficient Conditions for some Builtins in Ciao

	observable-free			
	pure			
	eval			
predicate	sideff_free	error_free	bind_ins	termin
var(X)	true	true	nonvar(X)	true
write(X)	false	true	ground(X)	true
$A \leq B$	true	$\text{arithexp}(A) \wedge \text{arithexp}(B)$	true	true
ground(X)	true	true	ground(X)	true
append(A,B,C)	true	true	true	$\text{list}(A) \vee \text{list}(C)$

- The assertions include *sufficient conditions* 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.

CiaoPP Assertions

- We use the concrete syntax of CiaoPP's assertion language for expressing in the form of assertions the sufficient conditions for each property stated in the previous table.
- The following assertions are predefined in Ciao for \geq / 2 :
 - `:- trust comp A >= B : (arithexp(A),arithexp(B)) + error_free`
 - `:- trust comp A >= B + sideff_free.`
 - `:- trust comp A >= B + bind_ins.`
- Side-effect assertions are always unconditional.
- Given a set of assertions AS and an atom A , $\text{property}(A, AS)$ denotes that there exists an assertion in AS :

```
:- trust comp p( $X_1, \dots, X_n$ ) :  $SC$  + property
```

s.t. $A = \theta(p(X_1, \dots, X_n))$ and $\theta(SC)$ succeeds.

Automatic Inference of Purity Assertions

- If non-leftmost unfolding is allowed, purity assertions are of interest not only for external predicates but also for the internal ones.
- Lack of purity assertions must be interpreted as the predicate not being pure
- Such assertions can be manually added by the user but we believe that this is too much of a burden
- 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.
 - ▶ Context-sensitive analysis would allow us to determine that some particular contexts guarantee the purity of atoms.

Automatic Inference by Backwards Analysis

- New application of backwards analysis for automatically inferring binding-insensitive, error-free and side-effect free assertions.
- Technique of [Gallagher – LOPSTR03]:
 - ① Identify properties required to hold at specific program points.
 - ② Meta-program automatically constructed to capture dependencies between goals and the specified program points.
 - ③ Standard abstract interpretation techniques applied to meta-program
 - ④ Output: conditions on initial goals which guarantee that given properties hold whenever the specified program points are reached.
- For our specific application:
 - ① observe the occurrences of *all* predicates (lack of purity assertions interpreted as the atom not being pure).
 - ② infer conditions under which calls to all predicates are pure.

Backwards Analysis for the Running Example

- 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.
```

- 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.
```

The last assertion indicates that calls performed to `problem(X,Y)` with the second argument being ground are binding insensitive. This will be very useful information for the specializer.

Combining Assertions with Partial Evaluation

- Extension of the definition of sound derivation to take into account the purity conditions in our assertions.
- The derivation step for G is *sound* w.r.t. AS if

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

- Consider the deterministic unfolding rule which derives the goal $\text{problem}(X, a), q(X)$ and cannot select the left atom.
- The assertions inferred for $\text{problem}(X, Y)$ allow us to jump over this atom and specialize first $q(X)$.
- Then X gets instantiated to a and the unfolding rule already can select the deterministic atom $\text{problem}(a, a)$.
- We obtain the specialized fact “ $\text{main}(a, a)$.” rather than:

$\text{main}(X, a) :- \text{problem}(X, a), q(X).$

which is much less efficient since `long_comp` remains residual

Conclusions

- We have presented a practical partial evaluation scheme for full Prolog programs with impure predicates.
- Existing approaches avoid backpropagating bindings and failure in the presence of such problematic predicates at the cost of accuracy.
- 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 to decide whether to backpropagate during non-leftmost unfolding.
- 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.