Partial Evaluation of the "Real Thing"

Michael Leuschel

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

Abstract. In this paper we present a partial evaluation scheme for a "real life" subset of Prolog. This subset contains first-order built-in's, simple side-effects and the operational predicate if-then-else. We outline a denotational semantics for this subset of Prolog and show how partial deduction can be extended to specialise programs of this kind. We point out some of the problems not occurring in partial deduction and show how they can be solved in our setting. Finally we provide some results based on an implementation of the above.

1 Introduction

Partial evaluation has been established as an important research topic, especially in the functional and logic programming communities. The topic has been introduced to logic programming in [10] and has later been called *partial deduction* when applied to pure logic programs. A sound theoretical basis of partial deduction is given in [12].

Although a lot of papers address partial deduction there are few approaches addressing partial evaluation of "real life" programs written for instance in some "real life" subset of Prolog. Even fewer papers discuss the theoretical implications of making the move from partial deduction to partial evaluation. This is what we propose to examine in this paper. First though we have to choose an adequate subset of Prolog. We have chosen a subset encompassing:

1. first-order[1] built-in's, like `var/1`, `nonvar/1` and `=../2`,
2. simple side-effects, like `print/1`,
3. the operational if-then-else construct.

Our choice tries to strike a balance between practical usability, complexity of the semantics and the potential for effective (self-applicable) partial evaluation. The most important decision was the inclusion of the if-then-else. As we will see in Sect. 3 the first-order built-in's and the side-effects do not add much complexity to the semantics. In later sections we will also see that the if-then-else lends itself quite nicely to partial evaluation and is much easier to cope with than the "full blown" cut. Using the if-then-else instead of the cut was already advocated in [14] and performed in [21].

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

In this paper we first formally define the subset of Prolog in Sect. 2 and give it a semantics in Sect. 3. In the sections 4 through 8 we adapt partial deduction such that it can cope with this subset of Prolog and study some of the added complications. Notably we will show that freeness and sharing information, although completely uninteresting in partial deduction, can be vital to produce efficient specialised programs. In Sect. 9 we extend the partial evaluation technique such that a Knuth-Morris-Pratt like search algorithm can be obtained by specialising a "dumb" search algorithm for a given pattern. We conclude with a discussion of related work and summarise our results.

2 Definition of RLP

The syntax of "Real-life Logic Programming" or simply RLP is based on the syntax of definite logic programs with the following extensions and modifications.

The concept of *term* remains the same. The set of *predicates* P is partitioned into the set of "normal" predicates P_{cl} defined through clauses and the set of built-in predicates P_{bi}. A *normal atom* (respectively a *built-in atom*) is an atom which is constructed using a predicate symbol $\in P_{cl}$ (respectively $\in P_{bi}$).

A *literal* is either an atom or it is an expression of the form (If $\rightarrow$ Then; Else) where If, Then, Else are lists of literals. We will denote by *Goals* the set of all lists of literals. We will represent a list of n literals by $(L_1, \ldots, L_n)$. Sometimes we will also use the notation $\leftarrow L_1, \ldots, L_n$.

A *clause* is an expression of the form $Head \leftarrow Body$ where $Head$ is a normal atom and $Body$ is a list of literals.

We can see from the above that RLP does not incorporate the negation nor the cut, but uses an if-then-else construct instead. This construct will behave just like the Prolog version of the if-then-else which contains a local cut and is usually written as `(If -> Then ; Else)`. Most uses of the cut can be mapped to if-then-else constructs and the if-then-else can also be used to implement the `not`[2]. The following informal Prolog clauses can be used to define the if-then-else:

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

Thus the behaviour of the if-then-else is as follows:

1. If the test-part succeeds then a local cut is executed and the then-part is entered.
2. If the test-part fails finitely then the else-part is entered.
3. If the test-part "loops" (i.e. fails infinitely) then the whole construct loops.

3 Semantics of RLP

We will now outline a semantics for RLP. This semantics should be preserved by any reasonable partial evaluation procedure for RLP.

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

The inclusion of the if-then-else into RLP has important consequences on the semantic level. As we have already pointed out the if-then-else contains a local cut. It is thus sensitive to the sequence of computed answers of the test-part. An implication being that the computation rule and the search rule have to be fixed in order to give a clear meaning to the if-then-else. From now on we will presuppose the Prolog left-to-right computation rule and the lexical search rule. The two programs hereafter illustrate the above point:

Program P_1	*Program* P_2
$q(\mathrm{X}) \leftarrow (p(\mathrm{X}) \rightarrow r(\mathrm{X}); fail)$	$q(\mathrm{X}) \leftarrow (p(\mathrm{X}) \rightarrow r(\mathrm{X}); fail)$
$p(a) \leftarrow$	$p(c) \leftarrow$
$p(c) \leftarrow$	$p(a) \leftarrow$
$r(c) \leftarrow$	$r(c) \leftarrow$

Using the Prolog computation and search rules, the query $\leftarrow q(\mathrm{X})$ will fail for program P_1 whereas it will succeed for P_2. All we have done is change the order of the computed answers for the predicate $p/1$. This implies that a partial evaluator which handles the if-then-else has to preserve the sequence of computed answers of all goals prone to be used inside an if-then-else test-part. This for instance is not guaranteed by the partial deduction framework in [12] which only preserves the computed answers but not their sequence.

As noted in Sect. 2 the if-then-else is also sensitive to non-terminating behaviour of the test-part. It is thus vital that our semantics also captures this aspect of RLP programs.

Finally a semantics for RLP has to take into account that the computed answers of built-in's cannot always be specified logically (`var/1` for instance) and that built-in's can generate side-effects which have no impact on the computed answers (`print/1` for example).

We fulfilled all the above requirements by adapting the denotational semantics of pure Prolog as defined in [2] and [17]. In these papers the semantics sem_P of a pure Prolog program P is a mapping from goals to (possibly infinite) sequences of computed answers. These sequences can be terminated by a least element $\perp$ which captures divergence (non-termination producing no computed answer).

Our semantics is based upon the view that computed answers and side-effects are events which can be observed by a person executing a RLP program. In that sense we will be characterising the observable behaviour of RLP programs.

Definition 1. The *side-effect domain* S_D is a (possibly infinite) set equipped with an equivalence relation $\equiv$. The elements of S_D are called *side-effects*.

In the remaining of this paper we suppose that the side-effect domain S_D is the set of built-in atoms and that the equivalence relation is syntactical identity. For instance the side-effect that "occurs" when $print(a)$ gets executed will be simply represented by the built-in atom $print(a)$. Our approach however makes no assumptions on S_D and it can thus be replaced by a finer structure if required.

Definition 2. An *event* is either a substitution representing a computed answer or a side-effect $\in S_D$. We will denote by $Event$ the set of all events.

Definition 3. Two events e_1, e_2 are equivalent with respect to a given goal G, denoted by $e_1 \equiv_G e_2$, if either

1. both e_1 and e_2 are side-effects and $e_1 \equiv e_2$ or
2. both $e_1 = \theta_1$ and $e_2 = \theta_2$ are substitutions and $G\theta_1$ and $G\theta_2$ are variants.[3]

Definition 4. An *event sequence* is an element of one of the following sets

1. $Event^*$: the set of finite sequences of events
2. $Event^* \times \{\bot\}$: the set of finite sequences of events terminated with $\bot$
3. $Event^\omega$: the set of infinite sequences of events

We define the notation $EventSeq = Event^* \cup Event^* \times \{\bot\} \cup Event^\omega$. Furthermore two event sequences will be equivalent w.r.t. a given goal G ($\equiv_G$) iff they have the same length and the corresponding elements of the sequences are equivalent w.r.t G.

Definition 5. An *ES-semantics sem* for a set of goals $G \subseteq Goals$ is a mapping $G \rightarrow EventSeq$. If $G = Goals$ then *sem* will be called *complete*. An *oracle* is an ES-semantics for $G = \{(A) \mid A \text{ is a built-in atom }\}$.

The concept of oracle as it is presented here, has nothing to do with the concept as presented in [1] where an oracle is used to abstract away from the sequential depth-first strategy of Prolog. We use an oracle to provide us with the meaning of the built-in's. Starting in the next section we will try to develop a partial evaluation procedure which preserves the semantics of a given RLP program independently of the actual oracle used to model the built-in's.

It is important to note that in Def. 5 the value returned by an oracle depends only on the actual call. In particular it does not depend on the program from which the built-in got called nor from any run-time environment. This makes our approach unsuitable to model built-in's which are not first order (like `assert/1` or `call/1`). But note that unrestricted use of built-in's like `assert/1` or `retract/1`, makes effective partial evaluation almost impossible.[4] Furthermore if the oracle's responses depended on something else it would be impossible to do semantics preserving program transformation without intricate assumptions on how the oracle's responses vary when the program or some run-time environment varies.

Unfortunately due to space restrictions we cannot elaborate on the exact details of our denotational ES-semantics and its fixpoint construction. Let us just state that the equivalence notion $\equiv$ for event sequences induces an equivalence on ES-semantics and we thus obtain an equivalence relation between programs. All further details will be made available in an upcoming technical report.

---

[3] This definition avoids a mistake of [17] which uses equivalence up to a renaming substitution (for which, contrary to what is stated in [17], $\{Y/X_1\}$ and $\{Y/X_2\}$ are not equivalent).

[4] On page 48 of [19] it is stated that "It is a question open to future research whether it is feasible to execute **assert/1** and **retract/1** by a partial evaluator."

The following table might help in giving the reader an intuition of our ES-semantics. Note that, when abstracting away from the side-effects, all the programs have the same least Herbrand model $M_P = \{q(a)\}$.

RLP Program P	*ES-Semantics* $sem_P(\leftarrow q(\mathrm{X}))$
$q(a) \leftarrow$	$(\{\mathrm{X}/a\})$
$q(a) \leftarrow print(a)$	$(print(a), \{\mathrm{X}/a\})$
$q(\mathrm{X}) \leftarrow q(\mathrm{X})$ $q(a) \leftarrow$	$(\bot)$
$q(a) \leftarrow$ $q(\mathrm{X}) \leftarrow q(\mathrm{X})$	$(\{\mathrm{X}/a\}, \{\mathrm{X}/a\}, \{\mathrm{X}/a\}, \ldots)$

4 LDR Trees

In the previous section we have sketched a semantics for RLP. In this section we elaborate on some of the problems when trying to preserve this semantics while performing partial evaluation. It should be clear that "standard" unfolding preserves the ES-semantics of a program as long as it uses the Prolog left-to-right computation rule[5] and it doesn't evaluate any non-logical predicates.

This of course summons the question of what we should do if the left-most literal is non-logical (or if it is logical but further unfolding it would lead us into an infinite loop). The easiest solution would be to just stop unfolding the goal completely. However the loss in specialisation is unacceptable. For instance in the framework of [12] the links (through variable sharing) between the remaining literals of the goal would be lost because the literals will be (even in the best possible case) partially evaluated separately. Worse, all specialisation w.r.t. the remaining literals would be lost if we use a method that constructs just one big SLD tree, as is the case with the partial evaluator developed by the author in [11]. To illustrate the problem let us take a look at the following simple program P_3 and try to specialise it for the query $\leftarrow q(\mathrm{X})$:

Program P_3		
$q(f(\mathrm{Y})) \leftarrow print(\mathrm{Y}), r(\mathrm{Y}), s(\mathrm{Y})$		
$r(a) \leftarrow$	$r(b) \leftarrow$	$r(c) \leftarrow$
$s(a) \leftarrow$	$s(b) \leftarrow$	$s(d) \leftarrow$

After the first unfolding step we obtain the goal $\leftarrow print(\mathrm{Y}), r(\mathrm{Y}), s(\mathrm{Y})$. If we stop unfolding and generate partial evaluations for the uncovered goals $r(\mathrm{Y})$ and $s(\mathrm{Y})$ we get the unmodified and unspecialised program P_3 back.

On the other hand if we do not follow the Prolog left-to-right rule and unfold the goals to the right of $print(\mathrm{Y})$ we obtain the specialised program P_4 which has a different ES-semantics.

[5] This is imperative even for purely logical programs. Take for instance the program "$p(\mathrm{X}, \mathrm{Y}) \leftarrow q(\mathrm{X}), q(\mathrm{Y}) \quad q(a) \leftarrow \quad q(b) \leftarrow$". If we unfold without following the left-to-right computation rule we change the order of solutions and the ES-semantics. In some cases this might change the (existential) left-termination behaviour.

Program P_4
$q(f(a)) \leftarrow print(a)$
$q(f(b)) \leftarrow print(b)$

For instance we have that $sem_{P_3}(\leftarrow q(f(\mathrm{X}))) = (print(\mathrm{X}), \{\mathrm{X}/a\}, \{\mathrm{X}/b\})$ while $sem_{P_4}(\leftarrow q(f(\mathrm{X}))) = (print(a), \{\mathrm{X}/a\}, print(b), \{\mathrm{X}/b\})$. The problem is caused by the backpropagation of substitutions onto the *print* atom.[6]

So we are faced with a dilemma: on the one hand we cannot select the left-most literal $print(\mathrm{Y})$ but on the other hand we cannot select $r(\mathrm{Y})$ or $s(\mathrm{Y})$ either because the substitutions will backpropagate onto $print(\mathrm{Y})$ changing the ES-semantics. The solution is however quite simple and just requires a minor extension of LD[7] trees: in addition to resolution steps we allow *residualisation* steps in the LD tree. These residualisation steps are not labelled with substitutions but with the selected literal which is thereby removed from the goal and hidden from vicious backpropagations. This extension allows us to follow the Prolog left-to-right computation rule without having to sacrifice neither specialisation nor correctness. This leads to the following definition of LDR trees.

Definition 6. A *complete LDR tree* for a program P is a tree whose nodes consist of goals such that the children for each goal $\leftarrow L_1, \ldots, L_k$ are:

1. either obtained by performing a resolution step with selected literal L_1. In this case the children are ordered according to the lexical ordering of the clauses used in the resolution step and the arcs are labelled with the substitutions of the resolution step.
2. or the goal has just one child which is obtained by performing a residualisation step on selected literal L_1. In this case the arc is labelled with L_1.

For a *(partial) LDR tree* we also allow any node to have no children at all (this corresponds to stopping unfolding). In this case there will be no selected literal.

Figure 1 depicts an LDR tree and shows how the introduction of the residualisation step solves the above dilemma. The generation of residual code from an LDR tree will be addressed in the next section. Note that the selected literal of a residualisation step is not necessarily a built-in. For instance it can be necessary to residualise a purely logical predicate to avoid infinite unfolding (see Sect. 6).

5 Generating Code from LDR trees

It should be quite obvious by looking at Fig. 1 that resultants are no longer sufficient for code generation. In fact we have to avoid the backpropagation of the bindings $\{\mathrm{Y}/a\}$ and $\{\mathrm{Y}/b\}$ onto the $print(\mathrm{Y})$ atom while ensuring that the

[6] The problem of backpropagation seems to have been overlooked in [6], thereby yielding a simple (non-pure) self-applicable partial evaluator whose semantics is however modified by self-application. Also note that under some circumstances backpropagation of single bindings can be allowed, see [19] and [16].

[7] SLD using the Prolog left-to-right computation rule and the lexical search rule.

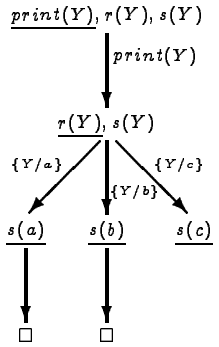

Fig. 1. LDR tree for program P_3

print(Y) atom gets executed only once. The only way to do this in the RLP framework is to generate a new predicate for the goal $\leftarrow r(\mathrm{Y}), s(\mathrm{Y})$. The code for this new predicate can then be generated by taking resultants as there are no more residualisation steps in its subtree. Using that code generation strategy we get the following specialised version of P_3:

Program P_5
$q(f(\mathrm{Y})) \leftarrow print(\mathrm{Y}), newp(\mathrm{Y})$
$newp(a) \leftarrow$
$newp(b) \leftarrow$

To formalise this process we first need the following notations:

Definition 7. Let τ be a LDR tree. Then $root(\tau)$ denotes the root of the tree. Also $\tau \xrightarrow{\alpha} \tau'$ holds iff τ' is a subtree of τ accessible via a branch whose sequence of labels is α.
We define $children(\tau) = \{(a, \tau') \mid \tau \xrightarrow{(a)} \tau'\}$. We also define $vars(\tau)$ to be the variables occurring in the arcs and leaves (but not the inner nodes) of τ.

In the following definition we present a way to remove a residualisation step from LDR trees. The basic idea is to split an LDR tree at a given residualisation step into two LDR trees which will be linked through the introduction of a new predicate. Figure 2 serves as an illustration of this definition.

Definition 8. Let S be a set of LDR trees. If we can find an LDR tree $\tau \in S$ such that

1. $\tau \xrightarrow{\alpha} \tau' \xrightarrow{R} \tau''$ where R is a residualised literal
2. $root(\tau') = \leftarrow R, G_1, \ldots, G_k$ and $root(\tau'') = \leftarrow G_1, \ldots, G_k$

then $(S \backslash \{\tau\}) \cup \{\tau_1, \tau_2\} \in rsplit(S)$ where

1. τ_1 is obtained from τ by replacing the subtree τ' by a tree consisting of the single node $\leftarrow R, newp(Args)$
2. $children(\tau_2) = \{(\phi, \tau'')\}$ and $root(\tau_2) = newp(Args)$

3. $newp$ is a new unused predicate
4. $Args = vars(\leftarrow G_1, \ldots, G_k) \cap vars(\tau'')$.

Note that in a sense we add a clause of the form $newp(Args) \leftarrow G_1, \ldots, G_k$ to the program to be specialised (and use it for folding).

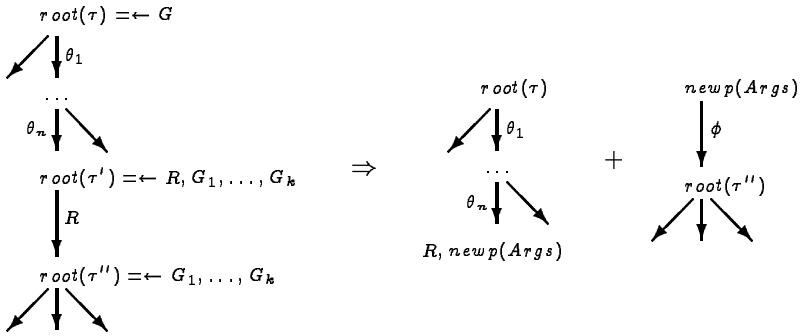

Fig. 2. Illustrating Def. 8 of $rsplit$

Using the above definitions we can now formalise a way to generate code for an LDR tree τ. We will denote by $rsplit^*(\tau)$ the set S_i obtained by the algorithm below. Note that $rsplit^*(\tau)$ is independent of the order in which the residualisation steps are removed.

Algorithm 1: Code Generation
$i = 0, S_0 = \{\tau\}$
while $rsplit(S_i) \neq \phi$ ***do***
S_{i+1} = some element $S' \in rsplit(S_i)$
increment i
od
take resultants for the set of partial LD trees S_i

Although the above technique always produces correct code it can generate superfluous predicates: if a newly generated predicate $newp$ has less than two defining clauses its introduction was not necessary. We present a post-processing scheme which remedies this (in practice this will be incorporated into $rsplit$).

Definition 9. Let $\theta = \{X_1/T_1 \ldots, X_n/T_n\}$ be a substitution and let eq be a binary predicate defined by the following single clause: "eq(X, X) ←".
Then we define $eqcode(\theta) = eq([X_1, \ldots, X_n], [T_1, \ldots, T_n])$. In practice we will write $eqcode(\theta)$ as $X_1 = T_1 \ldots, X_n = T_n$ and $eqcode(\phi)$ as $true$.

Using the above definition we define the post-processing as follows:

1. Transform all clauses "$Head \leftarrow L_1, \ldots, L_{i-1}, newp(Args)$" where $newp$ is a new predicate defined by the single clause "$newp(Args\theta) \leftarrow Body$" into the following clause (and then remove $newp$ from the program):
"$Head \leftarrow L_1, \ldots, L_{i-1}, eqcode(\theta), Body$".

2. Transform all clauses "$Head \leftarrow L_1, \ldots, L_{i-1}, newp(Args)$" where $newp$ has no defining clause into "$Head \leftarrow L_1, \ldots, L_{i-1}, fail$".

Note that, by definition of *rsplit*, a newly generated predicate $newp$ can only occur in the last position of a residual clause. Point 2 becomes interesting if we allow "safe" left-propagation of failure (see for instance [16] and [19]).

In [7] it is stated that "only the leftmost choice-point in a goal should be unfolded". Doing otherwise can be harmful for efficiency, for instance it can lead to the situation where an expensive goal has to be re-solved in the specialised program. This is certainly true for partial deduction. But in our case there is no backpropagation of bindings and the introduction of the new predicates not only guarantees that side-effects don't multiply but also that no expensive goal has to be re-solved. So in our framework unfolding (deterministic or not) is never harmful with respect to this problem.

6 Useless Bindings

In this section we will point at a particular problem which occurs when we want to preserve the ES-semantics during partial evaluation. This problem will be linked to "useless bindings" which are informally defined by the following:

> A *useless binding* inside a partial SLD tree (respectively LDR tree) is a binding which can be removed without changing the semantics (under consideration) of the resulting residual program.

Let us first show why useless bindings are no problem for partial deduction. When performing partial deduction according to [12] the specialised program is obtained by taking resultants of a partial SLD-tree. The combination of the mgu's, required to reach a leaf, is thus backpropagated on the top-level goal and any useless binding in the combination of mgu's automatically disappears. Suppose for instance that starting from the goal $\leftarrow p(\mathrm{X}, \mathrm{Y})$ we reach the leaf $\leftarrow q(\mathrm{Z})$ via the combination of mgu's $\theta = \{\mathrm{X/h(Z)}, \mathrm{W/\mathbf{h(Z)}}, \mathrm{Y/a}, \mathrm{V/\mathbf{b}}\}$. The resultant will be $p(h(\mathrm{Z}), a) \leftarrow q(\mathrm{Z})$ and all the useless bindings in bold have disappeared. In other words backpropagation on the top-level goal ensures that useless bindings have not even an effect on the *syntax* of the residual program.

When we want to preserve the ES-semantics and backpropagation is forbidden these useless bindings can become a big problem. This is even the case for purely logical programs. We will use the following programs to illustrate this.

Program P_6	*Program* P_7
$p(\mathrm{X}, \underline{\mathrm{Z}}, a) \leftarrow$	$p(\mathrm{X}, \underline{\mathrm{X}}, a) \leftarrow$
$p(\mathrm{X}, \mathrm{Y}, b) \leftarrow p(\mathrm{X}, \mathrm{Z}, \mathrm{W}), q(\mathrm{Z}, \mathrm{W})$	$p(\mathrm{X}, \mathrm{Y}, b) \leftarrow p(\mathrm{X}, \mathrm{Z}, \mathrm{W}), q(\mathrm{Z}, \mathrm{W})$
$q(b, b) \leftarrow$	$q(b, b) \leftarrow$
$q(a, a) \leftarrow$	$q(a, a) \leftarrow$

In Fig. 3 we show an LDR tree for the programs P_6 and P_7 in which $p(\mathrm{X}, \mathrm{Z}, \mathrm{W})$ has been residualised to avoid infinite unfolding. In this LDR tree the problematic

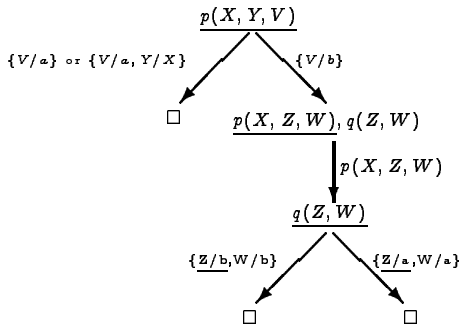

Fig. 3. LDR tree for Programs P_6 and P_7

substitutions $\{\mathrm{Z}/a, \mathrm{W}/a\}$ and $\{\mathrm{Z}/b, \mathrm{W}/b\}$ occur. In contrast to partial deduction these substitutions cannot be backpropagated onto the top-level goal.[8] Using the code generation strategy of algorithm 1 in Sect. 5 they will have an effect on the residual code. However for program P_6 the bindings Z/a and Z/b inside these substitutions are useless, i.e. removing them will yield a more efficient residual program with the same ES-semantics. The following table shows the generated residual programs with and without the bindings.

Specialised P_6 *with* $\mathrm{Z}/a, \mathrm{Z}/b$	*Specialised* P_6 *without* $\mathrm{Z}/a, \mathrm{Z}/b$
$p(\mathrm{X}, \mathrm{Z}, a) \leftarrow$	$p(\mathrm{X}, \mathrm{Z}, a) \leftarrow$
$p(\mathrm{X}, \mathrm{Y}, b) \leftarrow p(\mathrm{X}, \mathrm{Z}, \mathrm{W}), newp(\underline{\mathrm{Z}}, \mathrm{W})$	$p(\mathrm{X}, \mathrm{Y}, b) \leftarrow p(\mathrm{X}, \mathrm{Z}, \mathrm{W}), newp(\mathrm{W})$
$newp(\underline{b}, b) \leftarrow$	$newp(b) \leftarrow$
$newp(\underline{a}, a) \leftarrow$	$newp(a) \leftarrow$

This is just one simple example but our experiments have shown that bad code can be generated when no additional measures are taken. A way to solve this problem is to improve the calculation of the arguments *Args* for the new predicates in Def. 8. For example in Fig. 3 we can detect for P_6 that the variable Z will always be free after a successful call to $p(\mathrm{X}, \mathrm{Z}, \mathrm{W})$ and that the variable Z cannot possibly share with a variable of the top-most goal $p(\mathrm{X}, \mathrm{Y}, \mathrm{V})$. Thus Z can be removed from *Args* because no value will ever be transmitted via Z to the subtree for $q(\mathrm{Z}, \mathrm{W})$ and the bindings of the subtree which affect Z are of no consequence. By removing Z from *Args* the bindings Z/a and Z/b no longer have an effect on the residual code.[9]

[8] Backpropagating $\{\mathrm{Z}/a, \mathrm{W}/a\}$ and $\{\mathrm{Z}/b, \mathrm{W}/b\}$ by applying them on the top-level goal and all residualisation steps would generate a program which diverges after the first answer for $p(\mathrm{X}, \mathrm{Y}, \mathrm{Z})$.

[9] The interested reader might have noticed that for program P_6 in Fig. 3 the entire goal $q(\mathrm{Z}, \mathrm{W})$ can be replaced by *true* without changing the semantics. To do this safely however we have to calculate a safe approximation of the possible computed answers for the goal $p(\mathrm{X}, \mathrm{Z}, \mathrm{W})$. This could be: $\{\{\mathrm{W}/b\}, \{\mathrm{W}/a\}\}$ allowing us to conclude that one of the branches of $q(\mathrm{Z}, \mathrm{W})$ will succeed with the empty computed answer substitution and the other one will fail. Formalising and implementing this technique is still a matter of ongoing research.

Generalising the above we see that we can remove from $Args$ all those variables which are guaranteed to be free and guaranteed not to share with any variable of the root of the LDR tree. In order to detect this we have implemented an abstract interpretation scheme which calculates for each node i the following sets of variables:

1. $Free_i$, which are variables guaranteed to be free at that node
2. $Share_i$, which are potentially sharing with the variables of the root

We can remove a variable V from the set $Args$ of Def. 8 if $V \in Free_{root(\tau'')}$ and $V \notin Share_{root(\tau'')}$ where $root(\tau'')$ is the node of the LDR tree following the residualisation step. For example for Fig. 3 we have the following:

Goal of node i	$Free_i$	$Share_i$	*Valid for*
$\leftarrow p(\mathrm{X},\mathrm{Y},\mathrm{V})$	ϕ	$\{\mathrm{X},\mathrm{Y},\mathrm{V}\}$	P_6, P_7
$\leftarrow p(\mathrm{X},\mathrm{Z},\mathrm{W}), q(\mathrm{Z},\mathrm{W})$	$\{\mathrm{Z},\mathrm{W}\}$	$\{\mathrm{X}\}$	P_6, P_7
$\leftarrow q(\mathrm{Z},\mathrm{W})$	$\{\mathrm{Z}\}$	ϕ	P_6
$\leftarrow q(\mathrm{Z},\mathrm{W})$	ϕ	$\{\mathrm{Z}\}$	P_7

Thus for program P_6 we can remove the variable Z from the arguments $Args$ because $\mathrm{Z} \in Free_{root(\tau'')}$ and $\mathrm{Z} \notin Share_{root(\tau'')}$ while for program P_7 we cannot (where $root(\tau'')$ is the node with goal $\leftarrow q(\mathrm{Z},\mathrm{W})$).

In our implementation we also calculate for each root of an LDR tree (except possibly for the query goal) a set of "unused" argument variables. As we perform renaming and argument filtering (see [8], [3]) these unused variables can be removed from the residual code. More importantly this may diminish the number of variables sharing with the topmost goal and may allow us to generate much better code. A further improvement lies in generating multiple versions of a predicate call according to varying freeness information of the arguments. In our implementation we have accomplished this by integrating the freeness and sharing analysis into the partial evaluation process and using more refined notions of "instance" and "variant". Our system might thus generate two versions for a given goal $\leftarrow p(\mathrm{X})$: one version where X is guaranteed to be free and one where it is not. Under some circumstances this can substantially improve the quality of the generated code (especially when useless if-then-else constructs come into play, see Sect. 8).

We go even further and let the user of our system specify freeness for his top-level goal. In that case the set $Free_i$ for the top-level goal $\leftarrow p(\mathrm{X},\mathrm{Y},\mathrm{V})$ in the above table would no longer be ϕ but some user specified set. Note that now we are no longer performing partial evaluation in the usual sense. The specialised program will not be correct for all instances of the top-level goal, but only for those instances respecting the freeness information provided by the user.

From this section we can draw the conclusion that, as soon as we want to preserve the sequence of computed answers or the existential termination behaviour of programs (w.r.t. a given static computation rule), partial evaluation should be combined with additional analysis (i.e. abstract interpretation) to give good results.

7 Unfolding the If-then-else

Until now we have not touched the specialisation of the if-then-else predicate. Using the method presented so far we could perform a residualisation step on a selected if-then-else construct. However this technique performs no specialisation whatsoever inside the if-then-else. To remedy this problem we adapt residualisation such that, in addition to removing a selected if-then-else from a goal, it specialises the construct. The technique is based on partially evaluating the test-part and then based on this generating specialised then- and else-parts.

Suppose that the construct (If $\rightarrow$ Then; Else) is selected by a residualisation step. First we partially evaluate If yielding an LDR tree τ. We now calculate $S = rsplit^*(\tau)$ to remove all residualisation steps. We can now take the tree $\tau' \in S$ which has the goal If as root and obtain its sequence of branches β. The sequence β is useful because it provides us with substitutions which can be applied to the then-part. The process of generating the specialised if-then-else construct using β is detailed in algorithm 2 below and we obtain the label of the residualisation step by calling $code(\beta, \text{Then}, \text{Else})$. We also have to add $S \backslash \{\tau'\}$ to the global set of LDR trees determining our partial evaluation.

Algorithm 2: $code(\beta, \text{Then}, \text{Else})$

if β is empty *then*
  Partially evaluate Else yielding LDR tree τ_1
  Add τ_1' to the global set of LDR trees where
    $root(\tau_1') = newp_1(var(\text{Else}))$ and $children(\tau_1') = \{(\phi, \tau_1)\}$
  return $newp_1(var(\text{Else}))$
else
  Let $\beta = (b_1, \ldots, b_k)$ and let θ be the composition of substitutions of b_1
  Partially evaluate Thenθ yielding LDR tree τ_2
  Add τ_2' to the global set of LDR trees where
    $root(\tau_2') = newp_2(var(\text{Then}))$ and $children(\tau_2') = \{(\phi, \tau_2)\}$
  Let G be the leaf goal of b_1 and let $R = code((b_2, \ldots, b_k), \text{Then}, \text{Else})$
  return $((eqcode(\theta), G) \rightarrow newp_2(var(\text{Then}))\ ;\ R)$
fi

This technique generates a cascade of if-then-else's and an example is given in the next section. Note that to "unfold" the if-then-else we did not have to resort to other non-pure predicates. So in a sense RLP is closed under unfolding.

This unfolding resembles constructive negation (see e.g. [4]) when the if-then-else is used to represent the $not/1$. The definition based on if-then-else seems however much simpler from a practical viewpoint. Also note that our technique is as rather aggressive with respect to specialisation (it can be made even more aggressive by pushing code to the right of an if-then-else inside its then- and else-branches). For instance it specialises more than [9] but there is a risk of code-explosion. The results are however quite similar to the ones obtained by Mixtus in [18] which first transforms the if-then-else using a special form of cut and disjunctions. A similar approach is used in [16] which transforms the if-then-else using "ancestral cuts" and disjunctions. The problems discussed in the next section should also be relevant to these approaches.

8 Useless If-then-else Constructs

The problem of useless bindings discussed in Sect. 6 is also present for the new predicates $newp_1$ and $newp_2$ introduced in algorithm 2 and can be solved in a similar way. The problem tackled in this section usually occurs as soon as predicates with output arguments are present inside the test-part of an if-then-else. For example the variable Y in the following simple program P_8 is used as an output argument to *arc*.

Program P_8
$arcs\_leave(\mathrm{X}) \leftarrow one\_arc(\mathrm{X}, \mathrm{Z})$
$one\_arc(\mathrm{X}, \mathrm{Y}) \leftarrow (arc(\mathrm{X}, \mathrm{Y}) \rightarrow true; fail)$
$arc(a, b) \leftarrow \quad arc(b, a) \leftarrow \quad arc(a, c) \leftarrow$

Without freeness information we can only come up with the following specialisation for the goal $\leftarrow arcs\_leave(a)$:

$arcs\_leave(a) \leftarrow ((\mathrm{Z} = b) \rightarrow true; ((\mathrm{Z} = c) \rightarrow true; fail))$

Our experiments showed that such useless if-then-else constructs can become very frequent for programs with output arguments. Again through the use of freeness information, we can remedy this problem. For instance, knowing that Z is guaranteed to be free and and hence that $\mathrm{Z} = b$ will always succeed exactly once at run-time, we can come up with the following specialisation for $arcs\_leave(a)$:

$arcs\_leave(a) \leftarrow$

Note that it is not sufficient to just detect existential variables in residual clauses because output arguments can still be present in the residual program. Again a freeness and sharing analysis is needed to solve the problem. In our implementation we have used the same analysis which was used to solve the problem of useless bindings in Sect. 6. Using this information we can refine algorithm 2 by improving the calculation of arguments to $newp_1$ and $newp_2$ and by detecting when a test-branch (b_1 in Algorithm 2) is guaranteed to succeed exactly once. When this is the case we do not have to create a residual if-then-else construct (a conjunction suffices) and we can drop the else-part.

9 Results

All the ideas of the previous sections have been incorporated into a running system (based on [11]) whose results were very satisfactory. The removal of useless bindings and useless if-then-else constructs turned out to be vital for a lot of non-toy examples, especially for self-application. The system also performs well on the benchmarks used in [18]. We also performed tests with a regular expression parser taken from [13] yielding similar results, i.e. we obtain a deterministic automaton.

We now take a look at the famous match example. We will see that our treatment of the if-then-else is not yet optimal in the sense that we need one further optimisation to get a Knuth-Morris-Pratt (KMP) like search algorithm out of match.

Program P_9
$match(\mathrm{Pat}, \mathrm{T}) \leftarrow match1(\mathrm{Pat}, \mathrm{T}, \mathrm{Pat}, \mathrm{T})$
$match1([], \mathrm{Ts}, \mathrm{P}, \mathrm{T}) \leftarrow$
$match1([\mathrm{A}\|\mathrm{Ps}], [\mathrm{B}\|\mathrm{Ts}], \mathrm{P}, [\mathrm{X}\|\mathrm{T}]) \leftarrow$
$\quad (A = B \rightarrow match1(\mathrm{Ps}, \mathrm{Ts}, \mathrm{P}, [\mathrm{X}\|\mathrm{T}])\ ;\ match1(\mathrm{P}, \mathrm{T}, \mathrm{P}, \mathrm{T}))$

We do not obtain a KMP algorithm using the unfolding method presented so far because we do not use the information that the test has failed in the else-branch. This problem can been solved in several ways. For instance in [5],[20] constraints similar to $A \neq B$ are propagated. The SP system presented in [7] derives a KMP algorithm by left-propagation of bindings and the execution of ground not's. In this case the non-ground negative literals play the role of the constraints and the left-propagation of bindings takes care of testing the constraints. The partial evaluators Mixtus (see [18]) and Paddy (see [15]) are also able to derive KMP algorithms in what seems to be a quite similar way.

The question is how do we achieve KMP without left-propagation of bindings nor the explicit introduction of constraints. The solution is quite simple and consists in checking whether there are any apparent incompatibilities between the else-part and the fact that the test-part has to fail in order to reach the else-part. Formalising this we can transform constructs of the form $(\mathrm{If} \rightarrow \mathrm{Th}\ ;\ (\mathrm{E}_1, \mathrm{P}, \mathrm{E}_2))$ where the success of P implies the success of If and where If is purely logical into the simplified construct $(\mathrm{If} \rightarrow \mathrm{Th}\ ;\ (\mathrm{E}_1, fail))$. For instance we can transform $(\mathrm{X} = \mathrm{Y} \rightarrow p(\mathrm{X}); (\mathrm{X} = a, \mathrm{Y} = a, q(\mathrm{Z})))$ into $(\mathrm{X} = \mathrm{Y} \rightarrow p(\mathrm{X}); fail)$.[10] This simple transformation is sufficient to obtain a KMP-like string matcher out of P_9. Program P_{10} below was obtained by partially evaluating P_9 for the query $\leftarrow match([a, a, b], \mathrm{X})$. The above technique can be quite powerful because it does not require that bindings are actually left-propagated on E_1 and because it can extract information out of non-ground not's (when implemented using the if-then-else). Also freeness and sharing information can again be important to obtain good results (to detect success of If).

Program P_{10}
$match\_1(\mathrm{T}) \leftarrow match1\_2(\mathrm{T})$
$match1\_2([\mathrm{H}\|\mathrm{T}]) \leftarrow (H = a \rightarrow match1\_3(\mathrm{T})\ ;\ match1\_2(\mathrm{T}))$
$match1\_3([\mathrm{H}\|\mathrm{T}]) \leftarrow (H = a \rightarrow match1\_4(\mathrm{T})\ ;\ match1\_2(\mathrm{T}))$
$match1\_4([\mathrm{H}\|\mathrm{T}]) \leftarrow (H = b \rightarrow true\ ;\ (H = a \rightarrow match1\_4(\mathrm{T})\ ;\ match1\_2(\mathrm{T})))$

10 Discussion and Conclusion

The aspect of trying to preserve the sequence of computed answers (and side-effects) has been dealt with in some of the approaches based on the unfold/fold transformations of [22]. For instance [19] talks about the dangers of left-propagation of bindings and solves this by transforming the bindings into explicit code based on the `=/2` predicate and by introducing disjunctions. But as stated in [7]

[10] Which can be further simplified (using another simple optimisation based on the fact that a substitution will succeed at most once) into "$\mathrm{X} = \mathrm{Y}, p(\mathrm{X})$".

such an approach can have a harmful effect by removing indexing information. The method used in [16] relies on the introduction of new predicates in a similar way to ours.[11] However the problems of useless bindings or useless if-then-else constructs are not dealt with in these papers and the semantics that is used stays rather informal and is not denotational.

To our knowledge the preservation of the sequence of computed answers has not been dealt with in the framework of [12] and the concept of a residualisation step is new. Two other partial evaluators handle a significant subset of Prolog. One is Logimix coming from the functional world, described in [13]. The other one is described in [6] but, as noted earlier, does not always preserve the sequence of answers. The primary concern of these papers was self-application and the problems of useless bindings or useless if-then-else constructs are not dealt with.

In this paper we have introduced the ES-semantics for a non-trivial subset of Prolog. We have shown how, through the use of LDR trees, the ES-semantics can be preserved by partial evaluation. An important conclusion of this paper is that using just partial evaluation based on unfold/fold transformations can yield surprisingly bad results, leaving a lot of useless assignments, arguments or if-then-else constructs in the residual program. We have shown how freeness and sharing information, which are of no interest in partial deduction, can be used to solve this problem. Finally our results confirm that the if-then-else is very well suited for partial evaluation and can be used to obtain a KMP like string matcher by partially evaluating the simple "match" algorithm.

Acknowledgements

The author is supported by GOA "Non-standard applications of abstract interpretation." I would like to thank Danny De Schreye and Bern Martens for proof-reading an earlier version of this paper. I also would like to thank them for the stimulating discussions and for their encouragement. I also thank anonymous referees for their helpful remarks and for pointing out related work.

References

1. R. Barbuti, M. Codish, R. Giacobazzi, and M. Maher. Oracle semantics for Prolog. In H. Kirchner and G. Levi, editors, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, Lecture Notes in Computer Science 632, pages 100–114. Springer Verlag, 1992.
2. M. Baudinet. Proving termination of Prolog programs: A semantic approach. *The Journal of Logic Programming*, 14(1 & 2):1–29, 1992.
3. K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, October 1993.
4. D. Chan and M. Wallace. A treatment of negation during partial evaluation. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 299–318. MIT Press, 1989.

[11] There is however a small mistake w.r.t. the arguments for new predicates.

5. H. Fujita. An algorithm for partial evaluation with constraints. Technical Memorandum TM-0367, ICOT, 1987.
6. H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2 & 3):91–118, 1988.
7. J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
8. J. Gallagher and M. Bruynooghe. Some low-level transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 229–244, Leuven, Belgium, 1990.
9. C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* PhD thesis, Department of Computer Science, University of Bristol, January 1994.
10. J. Komorowksi. *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation.* PhD thesis, Linköping University, Sweden, 1981. Linköping Studies in Science and Technology Dissertations 69.
11. M. Leuschel. Self-applicable partial evaluation in Prolog. Master's thesis, K.U. Leuven, 1993.
12. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.
13. T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 214–227. Springer-Verlag, 1992.
14. R. O'Keefe. On the treatment of cuts in Prolog source-level tools. In *Proceedings of the Symposium on Logic Programming*, pages 68–72. IEEE, 1985.
15. S. D. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.
16. S. D. Prestwich. An unfold rule for full Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, Workshops in Computing, University of Manchester, 1992. Springer-Verlag.
17. M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics based Program Manipulation, PEPM'91*, Sigplan Notices, Vol. 26, N. 9, pages 274–284, Yale University, New Haven, U.S.A., 1991.
18. D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Swedish Institute of Computer Science, Mar. 1991.
19. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
20. D. A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In N. D. Jones and P. Hudak, editors, *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–71. ACM Press Sigplan Notices 26(9), 1991.
21. A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta programming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420, 1986.
22. H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984.

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