A Hybrid Approach to Conjunctive Partial Evaluation of Logic Programs[⋆]

Germán Vidal

MiST, DSIC, Universitat Politècnica de València, Spain
gvidal@dsic.upv.es

Abstract. Conjunctive partial deduction is a well-known technique for the partial evaluation of logic programs. The original formulation follows the so called online approach where all termination decisions are taken on-the-fly. In contrast, offline partial evaluators first analyze the source program and produce an annotated version so that the partial evaluation phase should only follow these annotations to ensure the termination of the process. In this work, we introduce a lightweight approach to conjunctive partial deduction that combines some of the advantages of both online and offline styles of partial evaluation.

1 Introduction

Partial evaluation [7] is a well-known technique for program specialization. From a broader perspective, some partial evaluators are also able to optimize programs further by, e.g., shortening computations, removing unnecessary data structures and composing several procedures or functions into a comprehensive definition. Within this broader approach, given a program and a *partial* (incomplete) call, the essential components of partial evaluation are: the construction of a *finite* representation—generally a graph—of the possible executions of (any instance of) the partial call, followed by the systematic extraction of a *residual* program (i.e., the partially evaluated program) from this graph. Intuitively, optimization can be achieved by compressing paths in the graph, by deleting unfeasible paths, and by renaming expressions while removing unnecessary function symbols.

Partial deduction. The theoretical foundations of partial evaluation for (normal) logic programs was first put on a solid basis by Lloyd and Shepherdson in [12]. When *pure* logic programs are considered, the term *partial deduction* is often used. Roughly speaking, in order to compute the partial deduction of a logic program P w.r.t. a set of atoms $\mathcal{A} = \{A_1, \ldots, A_n\}$, one should construct finite—possibly incomplete—SLD trees for the atomic goals $\leftarrow A_1, \ldots, \leftarrow A_n$, such that every leaf is either successful, a failure, or only contains atoms that

[⋆] This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant ACOMP/2010/042.

are *instances* of $\{A_1, \ldots, A_n\}$; this is the so-called *closedness* condition [12]. The residual program then includes a *resultant* of the form $A_i\sigma \leftarrow Q$ for every non-failing root-to-leaf derivation $\leftarrow A_i \leadsto^*_\sigma \leftarrow Q$ in the SLD trees. Similarly, we say that a residual program P' is *closed* when every atom in the body of the clauses of P' is an instance of a partially evaluated atom (i.e., an appropriate specialized definition exists).

From an algorithmic perspective, in order to partially evaluate a program P w.r.t. an atom A, one starts with the initial set $\mathcal{A}_1 = \{A\}$ and builds a *finite* (possibly incomplete) SLD tree for $\leftarrow A$. Then, all atoms in the leaves of this SLD tree which are not instances of A are added to the set, thus obtaining $\mathcal{A}_2$, and so forth. In order to keep the sequence $\mathcal{A}_1, \mathcal{A}_2, \ldots$ finite, some *generalization* is often required, e.g., by replacing some predicate arguments with fresh variables. Some variant of the *homeomorphic embedding* ordering [8] is often used to detect potential sources of non-termination (see Def. 6).

A sketch of this algorithm is shown in Fig. 1 (a), where $\mathit{unf}(\mathcal{A}_i)$ builds finite SLD trees for the atoms in $\mathcal{A}_i$ and returns the associated resultants, function *atoms* returns the atoms in the bodies of these resultants, and $\mathit{abs}(\mathcal{A}_i, \mathcal{A}')$ returns an approximation of $\mathcal{A}_i \cup \mathcal{A}'$ so that the sequence $\mathcal{A}_1, \mathcal{A}_2, \ldots$ is kept finite.

```
Initialization: $i := 1$;  $\mathcal{A}_i := \{A\}$;
Repeat
  $\mathcal{A}_{i+1} := \mathit{abs}(\mathcal{A}_i, \mathit{atoms}(\mathit{unf}(\mathcal{A}_i)))$;
  $i := i + 1$
Until $\mathcal{A}_i \approx \mathcal{A}_{i-1}$ (variants)
Return $\mathit{unf}(\mathcal{A}_i)$
```

(a) Partial deduction

```
Initialization: $i := 1$;  $\mathcal{C}_i := \{C\}$;
Repeat
  $\mathcal{C}_{i+1} := \mathit{abs}(\mathcal{C}_i, \mathit{unf}(\mathcal{C}_i))$;
  $i := i + 1$
Until $\mathcal{C}_i \approx \mathcal{C}_{i-1}$ (variants)
Return $\mathit{unf}(\mathcal{C}_i)$
```

(b) Conjunctive partial deduction

Fig. 1. Basic algorithms

Conjunctive partial deduction. One of the main drawbacks of partial deduction is the fact that the atoms in the leaves of every SLD tree are partially evaluated independently. Usually, this implies a significant loss of accuracy. To overcome this drawback, a new framework called *conjunctive partial deduction* (CPD) was introduced [5].

Loosely speaking, the main difference with standard partial deduction is that it considers the partial evaluation of non-atomic goals. Here, in order to partially evaluate a program P w.r.t. a conjunction C, one starts with the initial set $\mathcal{C}_1 = \{C\}$, and builds a finite SLD tree for $\leftarrow C$; then, every leaf in the SLD tree is added to the set and so forth. Trivially, this process is usually infinite. Now, in order to keep the sequence $\mathcal{C}_1, \mathcal{C}_2, \ldots$ finite, generalization of predicate arguments does not suffice and the conjunctions in the leaves of the SLD trees should often be *split up* to avoid conjunctions that keep growing infinitely.

The process is sketched in Fig. 1 (b), where *unf* now returns the conjunctions in the bodies of resultants and *abs* also includes some algorithm for splitting conjunctions so that the sequence $\mathcal{C}_1, \mathcal{C}_2, \ldots$ is still kept finite.

Online versus offline. Depending on *when* control issues—like deciding which atoms should or should not be unfolded or how conjunctions should be split up—are addressed, two main approaches to partial evaluation can be distinguished. In *offline* approaches to partial evaluation, these decisions are taken beforehand by means of a static analysis (where we know which parameters are known but not their values). In contrast, *online* partial evaluators take decisions on the way (so that actual values of static data are available).

While offline partial evaluators are usually faster, online ones produce more accurate results (though, from a theoretical point of view, they are equally powerful [3]). Partial evaluators based on the CPD scheme have traditionally followed the online approach.

Motivation. Some of the weaknesses of current online partial evaluators based on the CPD scheme (like ECCE [10]) are the following. First, partial evaluation algorithms are conceptually rather complex, which makes it difficult to predict the outcome of the process. Moreover, they are often computationally expensive to implement (e.g., the online partial evaluator ECCE runs almost three orders of magnitude slower than offline systems like LOGEN [10]).

Furthermore, current CPD algorithms do not consider *run-time* information. However, some run-time information, like groundness information, could be useful for improving existing strategies for splitting conjunctions so that no relevant run-time variable sharing is lost. Unfortunately, run-time information is rather difficult to preserve through partial evaluation. Consider, e.g., that an atom $q(X, Y)$ is partially evaluated. Then, if we follow the usual approach, an instance like $q(W, W)$ will be considered closed w.r.t. $q(X, Y)$. However, this implies a serious loss of run-time groundness information since we cannot ensure that X and Y are independent anymore. Of course, generalizing predicate arguments or splitting conjunctions arbitrarily are other ways of losing run-time information.

In this work, we try to overcome some of these drawbacks by introducing a hybrid approach to CPD of definite logic programs as follows:

Pre-processing stage: First, we apply a simple call and success pattern analysis that identifies which predicate arguments will be ground *at run-time*. A termination analysis is then applied to identify possibly non-terminating calls (for the computed call patterns). This information will become useful to decide when non-leftmost unfolding is admissible. Finally, we introduce a syntactic characterization to identify *non-regular* predicates whose unfolding might give rise to infinitely growing conjunctions during partial evaluation.

Partial evaluation: Roughly speaking, this stage can be seen as an instance of traditional CPD algorithms, though it also includes some significant differences: firstly, splitting of conjunctions is statically determined from the information gathered by the call and success pattern analysis and from the computed set of non-regular predicates (rather than inspecting the history of partially evaluated queries, which is much more computationally expensive); secondly, a conjunction is only considered closed when it is a variant of an already partially evaluated conjunction (thus avoiding the loss of run-time information); also, our procedure includes no generalization but simply gives up when termination cannot be ensured (thus returning calls to the original, not renamed predicates instead); finally, non-leftmost unfolding is allowed as long as the selected atoms are terminating (at run-time) according to the termination analysis performed in the pre-processing stage.[1]

Post-processing stage: Finally, we extract the residual clauses from the computed partial evaluations using the standard notion of resultant. In principle, we only compute one-step resultants (which increases the opportunities for *folding* back the calls of the SLD trees). Moreover, and in contrast to what is usually done, resultants are produced during the construction of SLD trees: every unfolding (or splitting) generates an associated resultant. In traditional approaches, resultants are computed *a posteriori* when the partial evaluation process is finished. As in the original CPD framework, all conjunctions are renamed (except for those where termination could not be ensured and we gave up). Finally, we apply a standard post-unfolding transformation where calls to *intermediate* predicates are unfolded.

To summarize, our technique can be seen as a lightweight approach to CPD that combines some of the advantages of both online and offline styles of partial evaluation. Moreover, it represents a first step towards a technique that keeps run-time information during partial evaluation as much as possible (though the splitting of conjunctions still implies a loss of information).

Our scheme is potentially faster than existing approaches since some of the most expensive operations, generalization and splitting, do not exist anymore (generalization) or are much simpler (splitting) thanks to the use of information gathered by the static analyses. A prototype implementation of the hybrid partial evaluator is available at `http://german.dsic.upv.es/lite.html`. Despite its simplicity (a few hundred lines of Prolog code), the results for definite logic programs (including built-in's) are not far from those obtained with mature CPD systems like ECCE [10] when the residual program is closed (i.e., it contains no calls to the original predicates). Otherwise, we still get some modest improvements; in general, no significant slowdown was produced.

The paper is organized as follows. Section 2 presents the static analyses that are performed before partial evaluation starts. The main algorithm for CPD is then described in Sect. 3, while the extraction of residual programs is presented

[1] Observe that we do not require the selected atoms to terminate at partial evaluation time (actually, since they are less instantiated than usual run-time calls, partial computations are often non-terminating). The unfolding of non-leftmost non-terminating (*at run-time*) atoms is avoided because it can break the equivalence w.r.t. finite failures. This is often ensured by requiring the construction of weakly fair SLD trees [5], which is sometimes a too restrictive condition.

in Sect. 4. Section 5 summarizes our findings from an experimental evaluation of the new technique and, finally, Sect. 6 concludes and discusses some possibilities for future work.

2 Pre-Processing Stage

Our pre-processing stage consists of three different analyses. The first two analyses are well-known in the literature. The first one is a simple call and success pattern analysis like that introduced in [11]. Basically, given an initial query and the call patterns for the atoms in this query, the analysis infers for every predicate p/n a number of call/success patterns of the form $p/n : \pi_{in} \mapsto \pi_{out}$ such that π_{in} and π_{out} are subsets of $\{1,\ldots,n\}$ denoting the arguments π_{out} of p/n which are definitely ground after a successful derivation, assuming that it is called with ground arguments π_{in}. This information could also be provided by the user (as in Mercury [13]).

The second analysis is a standard left-termination analysis (i.e., an analysis for universal termination under Prolog's left-to-right computation rule) like those based on the abstract binary unfoldings [4], size-change analysis [1], etc. This information, together with the call and success pattern analysis, will be essential to determine when non-leftmost unfolding is admissible at specialization time.

Finally, we introduce a syntactic characterization that allows us to identify which predicate calls might give rise to infinitely growing conjunctions at partial evaluation time. Our formulation can be seen as a generalization of the notion of *B-stratifiable* programs in [6] to the context of partial evaluation. The main difference is that we consider a flexible computation rule, while [6] considers a fixed left-to-right rule (and thus their notion is more restrictive).

In the following, we say that the *call graph* of a program P is a directed graph that contains the predicate symbols of P as vertices and an edge from predicate p/n to predicate q/m for each clause $p(t_1,\ldots,t_n) \leftarrow \mathit{body}$ and atom $q(s_1,\ldots,s_m)$ of *body*.

Definition 1 (strongly regular logic programs). *Let P be a logic program and let $CG_1,\ldots,CG_n$ be the strongly connected components (SCC) in the call graph of P. We say that P is strongly regular if there is no clause $p(t_1,\ldots,t_n) \leftarrow$ body such that body contains two atoms $q(s_1,\ldots,s_m)$ and $r(l_1,\ldots,l_k)$ such that q/m and r/k belong to the same SCC of p/n.*

Intuitively speaking, strongly regular programs cannot produce infinitely growing conjunctions at partial evaluation time when the usual dynamic computation rules of existing partial evaluators are considered (like that of Def. 6 below).

When a program P is not strongly regular, we identify the predicates that are responsible for violating the strongly regular condition: we say that a predicate p/n is *non-regular* if there is a clause $p(t_1,\ldots,t_n) \leftarrow \mathit{body}$ and *body* contains two atoms with predicates q/m and r/k that belong to the same SCC of p/n.

Identifying non-regular predicates will become useful to (statically) decide how to split queries at partial evaluation time.

Example 1. Consider the following Prolog program from the DPPD library [9]:

```
applast(L,X,Last) :- append(L,[X],LX), last(Last,LX).
last(X,[X]).
last(X,[H|T]) :- last(X,T).
append([],L,L).
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).
```

Here, there are three SCCs, {`applast/3`}, {`append/3`} and {`last/2`}, but no clause violates the strongly regular condition. In contrast, the following program (also from the DPPD library [9]):

```
flipflip(XT,YT) :- flip(XT,TT), flip(TT,YT).
flip(leaf(X),leaf(X)).
flip(tree(L,I,R),tree(FR,I,FL)) :- flip(L,FL), flip(R,FR).
```

is not strongly regular. Here, we have two SCCs, {`flipflip/2`} and {`flip/2`}, and the second clause of `flip/2` violates the strongly regular condition. As a consequence, we say that `flip/2` is a non-regular predicate.

3 Partial Evaluation Stage

In this section, we present the main stage of our CPD procedure. As it is common practice, we avoid infinite unfolding by means of a well-known strategy based on the use of the *homeomorphic embedding* ordering [8]. The embedding relation $\trianglerighteq$ is defined as the least relation satisfying (here f denotes a function symbol and p a predicate symbol):

- $x \trianglerighteq y$ for all variables x, y;
- $f(t_1, \ldots, t_n) \trianglerighteq s$ if $t_i \trianglerighteq s$ for some $i \in \{1, \ldots, n\}$;
- $f(t_1, \ldots, t_n) \trianglerighteq f(s_1, \ldots, s_n)$ if $t_i \trianglerighteq s_i$ for all $i = 1, \ldots, n$.
- $p(t_1, \ldots, t_n) \trianglerighteq p(s_1, \ldots, s_n)$ if $t_i \trianglerighteq s_i$ for all $i = 1, \ldots, n$ and $p(s_1, \ldots, s_n)$ is not a strict instance of $p(t_1, \ldots, t_n)$.[2]

When $A_i \trianglerighteq A_j$ holds, we say that atom A_i *embeds* atom A_j. The embedding relation is extended to queries as follows: $Q \trianglerighteq Q'$ if $Q = A_1, \ldots, A_n$, $Q' = A'_1, \ldots, A'_n$ and $A_i \trianglerighteq A'_i$ for all $i = 1, \ldots, n$.

Definition 2 (covering ancestors [2]**).** *Given an SLD resolution step*

$$\leftarrow A_1, \ldots A_i, \ldots, A_n \;\hookrightarrow_\sigma\; \leftarrow (A_1, \ldots A_{i-1}, A'_1, \ldots, A'_m, A_{i+1}, \ldots, A_n)\sigma$$

with selected atom A_i using clause $A \leftarrow A'_1, \ldots, A'_m$, $\sigma = mgu(A_i, A)$, we say that A_i is the parent of atoms $A'_1\sigma, \ldots, A'_m\sigma$ in this step. The ancestor relation is just the transitive closure of the parent relation.

Finally, the "covering" ancestors of a query atom in an SLD derivation is the subset of its ancestors with the same predicate symbol.

[2] This last condition is required to have $p(X,X) \trianglerighteq p(A,B)$ but not $p(A,B) \trianglerighteq p(X,X)$, see [8] for more details.

Basically, we will ensure termination by avoiding the unfolding of those calls that embed some of their covering ancestors (see Def. 6 below).

In contrast to previous approaches, we do not explicitly distinguish between the so-called *local* and *global* levels and construct a single partial evaluation tree that comprises both levels. Our CPD procedure deals with *extended queries*:

Definition 3 (extended query). *We consider extended queries (and goals) of the form* $(A_1, anc_1, \pi_1), \ldots, (A_n, anc_n, \pi_n)$*, where* A_i *is an atom,* anc_i *is a set of atoms (the—standardized apart—covering ancestors of* A_i*), and* π_i *is the call pattern of* A_i*,* $i = 1, \ldots, n$*. We denote the empty extended query by true.*

Given an extended query Q*, we introduce the following auxiliary function:* $query(Q) = A_1, \ldots, A_n$*, if* $Q = (A_1, anc_1, \pi_1), \ldots, (A_n, anc_n, \pi_n)$*.*

Before introducing the notion of SLD resolution over extended queries, we need the following preparatory definition:

Definition 4. *Let* $A = p(s_1, \ldots, s_n)$ *and* $B = p(t_1, \ldots, t_n)$ *be atoms and* π *be a call pattern. We define the function "propagate" in order to propagate the groundness information to the body atoms of a clause as follows:*

$$propagate(A, \pi, B) = \{i \mid \mathcal{V}ar(t_i) \cap \mathcal{V}ar(\pi(A)) = \emptyset\}$$

where $\pi(A)$ *returns the ground arguments of* A *according to* π*, i.e.,*

$$\pi(p(s_1, \ldots, s_n)) = \{s_j \mid j \in \pi\}$$

(so $\mathcal{V}ar(\pi(A))$ *are ground variables of* A *according to* π*).*[3]

We are now ready to introduce the extended notion of SLD resolution:

Definition 5 (extended SLD resolution). *Extended SLD resolution, denoted by* $\rightsquigarrow$*, is a natural extension of SLD resolution over extended queries. Formally, given a program* P*, an extended query* $Q = (A_1, anc_1, \pi_1), \ldots, (A_n, anc_n, \pi_n)$*, and a computation rule* $\mathcal{R}$*, we say that* $\leftarrow Q \rightsquigarrow_{P,\mathcal{R},\sigma} \leftarrow Q'$ *is an* extended SLD resolution step *for* Q *with* P *and* $\mathcal{R}$ *if the following conditions hold:*[4]

- $\mathcal{R}(Q) = (A_i, anc_i, \pi_i)$*,* $1 \leq i \leq n$*, is the selected extended atom,*
- $H \leftarrow B_1, \ldots, B_m$ *is a renamed apart clause of* P*,*
- $\sigma = mgu(A_i, H)$ *is the computed most general unifier, and*
- $Q' = (\leftarrow (A_1, anc_1, \pi_1), \ldots, (A_{i-1}, anc_{i-1}, \pi_{i-1}),$
  $(B_1, anc'_1, \pi'_1), \ldots, (B_m, anc'_m, \pi'_m),$
  $(A_{i+1}, anc_{i+1}, \pi_{i+1}), \ldots, (A_n, anc_n, \pi_n))\sigma$
  where $anc'_j = anc_i \cup \{A_i\sigma_i\}$*,* σ_i *is a renaming substitution, and* $\pi'_j = propagate(A_i\sigma, \pi_i, B_j\sigma)$*,* $j = 1, \ldots, m$*.*

[3] Let us note that *propagate* does not take variable sharing into account, thus it returns just an approximation of the ground variables.

[4] We often omit P, $\mathcal{R}$ and/or σ in the notation of an extended SLD resolution step when they are clear from the context.

Trivially, extended SLD resolution is a conservative extension of SLD resolution: given extended queries Q, Q', we have that $\leftarrow Q \leadsto_\sigma \leftarrow Q'$ implies $\leftarrow \mathit{query}(Q) \hookrightarrow_\sigma \leftarrow \mathit{query}(Q')$.

In the following, we say that two (extended) queries Q and Q' are *variants*, denoted by $Q \approx Q'$, if there is a renaming substitution σ such that $Q\sigma = Q'$.

Now, we introduce our unfolding strategy based on the notions of embedding ordering and covering ancestors (analogously to, e.g., [2]):

Definition 6 (unfolding strategy, $\overset{\rhd}{\leadsto}$). *Given extended queries Q, Q', we have $\leftarrow Q \overset{\rhd}{\leadsto}_\sigma \leftarrow Q'$ if the following conditions hold:*

- *the extended SLD resolution step $\leftarrow Q \leadsto_\sigma \leftarrow Q'$ holds, where the selected extended atom (A, anc, π) is the leftmost extended atom such that there is no atom $B \in \mathit{anc}$ with $A \trianglerighteq B$ (if any);*
- *either (A, anc, π) is the leftmost extended atom of Q or A is left-terminating for π (according to the left-termination analysis done in the pre-processing stage).*

It is not difficult to prove (e.g., from the results in [2]) that our unfolding strategy guarantees that derivations are always terminating.

In the following, we will distinguish the following kinds of non-unfoldable extended queries:

- *failing* extended queries, in which no selected atom matches the head of a clause;
- *stalled* extended queries, in which every selected atom embeds one of its covering ancestors. In this case, we say that *stalled*(Q) holds.

For instance, the extended query $(\mathtt{head}([\,], \mathtt{H}), \{\,\}, \{\,\})$ is failing w.r.t. the usual definition of predicate `head`:

```
head([H|_], H).
```

On the other hand, the extended query $(\mathtt{head(X)}, \{\mathtt{head(Y)}\}, \{\})$ is stalled since $\mathtt{head(X)} \trianglerighteq \mathtt{head(Y)}$.

As mentioned in the introduction, in the context of CPD, avoiding the unfolding of atoms that embed some ancestor is not enough to ensure the termination of the process. In general, some form of splitting of queries is also needed. In our calculus, we consider two different forms of splitting. The first one is based on the notion of *independence* and allows us to split up a query when it includes two subsequences that do not share variables and, thus, can be evaluated independently (at run time) without any serious loss of accuracy.

In the following, given an extended query $Q = (A_1, \_, \pi_1), \ldots, (A_n, \_, \pi_n)$ where "_" denotes an irrelevant argument, the set $\mathit{fvars}(Q)$ of free variables of Q is defined as follows: $\mathit{fvars}(Q) = \mathcal{V}ar(Q) \setminus \{X \in \mathcal{V}ar(\pi_i(A_i)) \mid i = 1, \ldots, n\}$.

Now, we introduce our splitting operations. We note that both of them only perform a partitioning of the considered query in such a way that only consecutive atoms are considered and the order is not changed. More flexible splitting operations where atoms are mixed or duplicated are also possible (see [5]).

Definition 7 (independent splitting, $\mathsf{i\text{-}split}$). *Let Q be an extended query. We say that the set of extended queries $\{Q_1, Q_2, Q_3\}$ is an independent splitting of Q, denoted by $\{Q_1, Q_2, Q_3\} \in \mathsf{i\text{-}split}(Q)$, if the following conditions hold:*

- *$Q = Q_1, Q_2, Q_3$;*
- *both Q_1 and Q_2 contain at least one extended atom (but Q_3 might be empty);*
- *there are no free variables (at run time) in common between Q_1 and Q_2, i.e., $\mathit{fvars}(Q_1) \cap \mathit{fvars}(Q_2) = \emptyset$.*

In general, there might be more than one independent splitting for a given extended query. In our approach, we try to minimize the length of Q_3 (i.e., so that Q_3 is empty in the optimal case).[5]

E.g., given the extended query

$Q = (\mathtt{append}(\mathtt{X},\mathtt{Y},\mathtt{L_1}), \_, [1,2]), (\mathtt{append}(\mathtt{X},\mathtt{Z},\mathtt{L_2}), \_, [1,2]), (\mathtt{append}(\mathtt{L_1},\mathtt{L_2},\mathtt{R}), \_, [\,])$

we have $\{Q_1, Q_2, Q_3\} \in \mathsf{i\text{-}split}(Q)$ with

$Q_1 = (\mathit{append}(X, Y, L_1), \_, [1,2])$
$Q_2 = (\mathit{append}(X, Z, L_2), \_, [1,2])$
$Q_3 = (\mathit{append}(L_1, L_2, R), \_, [\,])$

since $\mathit{fvars}(Q_1) = \{L_1\}$, $\mathit{fvars}(Q_2) = \{L_2\}$ and their intersection is empty (moreover, since $\mathit{fvars}(Q_3) = \{L_1, L_2, R\}$ this is the only possible independent splitting). Observe that variable independence is only guaranteed at run time (e.g., Q_1 and Q_2 share variable X at partial evaluation time).

Our second form of splitting often involves a more serious loss of accuracy and is only used when termination cannot be guaranteed otherwise.

Definition 8 (regular splitting, $\mathsf{r\text{-}split}$). *Let Q be an extended query. We say that the set of extended queries $\{Q_1, \ldots, Q_n\}$, $n \geq 1$, is a regular splitting of Q, denoted by $\{Q_1, \ldots, Q_n\} \in \mathsf{r\text{-}split}(Q)$, if the following conditions hold:*

- *$Q = Q_1, \ldots, Q_n$;*
- *every query Q_i contain at least one extended atom;*
- *every query Q_i contains at most one call to a non-regular predicate (according to the analysis performed in the pre-processing stage).*

In general, there might be more than one regular splitting for a given query. Here, we let $\mathsf{r\text{-}split}(Q)$ return any of them.[6]

[5] In [5], the notion of *maximally connected subconjunction* is introduced for a similar purpose. However, while the purpose of [5] is keep the longest possible conjunctions, our aim is to minimize the loss of run-time variable sharing.

[6] In the implemented partial evaluator, we just traverse Q from left to right and start a new subsequence every time a call to a non-regular predicate is found.

$$\textsf{(success)} \qquad \frac{Q = \mathit{true}}{\langle Q, \mathit{memo}\rangle \longrightarrow \langle\ \rangle}$$

$$\textsf{(variant)} \qquad \frac{\exists Q' \in \mathit{memo}.\ \mathit{query}(Q) \approx \mathit{query}(Q')}{\langle Q, \mathit{memo}\rangle \longrightarrow \langle\ \rangle}$$

$$\textsf{(failure)} \qquad \frac{\neg \mathit{stalled}(Q)\ \wedge\ \not\exists Q'.\ \leftarrow Q \overset{\rhd}{\leadsto}_\sigma \leftarrow Q'}{\langle Q, \mathit{memo}\rangle \longrightarrow \langle\ \rangle}$$

$$\textsf{(embedding)} \qquad \frac{\mathit{stalled}(Q)\ \wedge\ \exists Q' \in \mathit{memo}.\ \mathit{query}(Q) \trianglerighteq \mathit{query}(Q')}{\langle Q, \mathit{memo}\rangle \overset{emb}{\longrightarrow} \langle\ \rangle}$$

$$\textsf{(unfold)} \qquad \frac{\exists Q'.\ \leftarrow Q \overset{\rhd}{\leadsto}_\sigma \leftarrow Q'}{\langle Q, \mathit{memo}\rangle \overset{unf}{\longrightarrow}_\sigma \langle Q', \mathit{memo} \cup \{Q\}\rangle}$$

$$\textsf{(i-split)} \qquad \frac{\mathit{stalled}(Q)\ \wedge\ \not\exists Q' \in \mathit{memo}.\ \mathit{query}(Q) \trianglerighteq \mathit{query}(Q')\ \wedge\ Q'' \in \textsf{i-split}(Q)}{\langle Q, \mathit{memo}\rangle \overset{is}{\longrightarrow} \langle Q'', \mathit{memo} \cup \{Q\}\rangle}$$

$$\textsf{(r-split)} \qquad \frac{\mathit{stalled}(Q)\ \wedge\ \not\exists Q' \in \mathit{memo}.\ \mathit{query}(Q) \trianglerighteq \mathit{query}(Q')\ \wedge\ Q'' \in \textsf{r-split}(Q)}{\langle Q, \mathit{memo}\rangle \overset{rs}{\longrightarrow} \langle Q'', \mathit{memo} \cup \{Q\}\rangle}$$

Fig. 2. Partial evaluation semantics

Let us consider again the programs of Example 1. In this case, it is easy to check that $\{Q\} \in \textsf{r-split}(Q)$ for

$$Q = (\texttt{append}(\texttt{L}, [\texttt{X}], \texttt{LX}), \_, \_), (\texttt{last}(\texttt{Last}, \texttt{LX}), \_, \_)$$

(i.e., no regular splitting is necessary), but $\{Q_1, Q_2\} \in \textsf{r-split}(Q)$ for

$$Q = (\texttt{flip}(\texttt{L}, \texttt{FL}), \_, \_), (\texttt{flip}(\texttt{R}, \texttt{FR}), \_, \_)$$

with $Q_1 = (\texttt{flip}(\texttt{L}, \texttt{FL}), \_, \_)$ and $Q_2 = (\texttt{flip}(\texttt{R}, \texttt{FR}), \_, \_)$, since `flip/2` is non-regular.

We formalize our CPD algorithm by means of a (labelled) state transition system. In our context, *states* are pairs of the form $\langle q, \mathit{memo}\rangle$ where q is an extended query and *memo* is a set of extended queries (the queries already partially evaluated).

The rules of the partial evaluation calculus are shown in Fig. 2. Let us briefly describe these rules:

– The first two rules, success and variant, should be self-explanatory: both empty queries and variants of previously partially evaluated queries are just discarded. Note that, in contrast to traditional partial evaluators, an instance of a previously partially evaluated query is not discarded (i.e., it is not considered *closed*). The reason for this decision is to enforce the preservation of run-time information—like variable sharing and groundness—as much as possible.

– The next two rules are similar but consider different cases. Rule failure is used when a query is not stalled but the selected atom does not match the head of any clause. Rule embedding is considered when the query is stalled because all atoms embed some covering ancestor. Although both cases terminate the current derivation, we need to consider them separately because we will produce a resultant only in the second case.
– The next rule, *unfold*, just performs an extended SLD resolution step using the unfolding strategy of Def. 6. Observe that this rule is non-deterministic since the underlying extended SLD resolution is also non-deterministic. Therefore, we construct a tree structure during partial evaluation.
– Finally, the last two rules perform the splitting of stalled queries that do not embed any partially evaluated query. These rules are tried in the textual order, so that when the first rule i-split is applicable, we discard the second rule r-split. We consider that only one (independent or regular) splitting is applicable. However, these rules are still non-deterministic since they should initiate subderivations for every subconjunction.

Observe that we only label the relation $\longrightarrow$ in those cases in which a resultant is produced (see next section).

Example 2. Let us consider again the definition of applast/3 in Example 1. Then, the calculus of Fig. 2 computes (among others) the following derivation for $Q_0 =$ $(\mathsf{applast(L, X, Last)}, \{\,\}, \{1\})$ (the unfolded atom is underlined):[7]

$$
\begin{array}{ll}
\langle Q_0, \{\,\}\rangle & \\
\xrightarrow{unf}_{\{\}} & \langle(\underline{\mathsf{append(L,[X],LX)}}, \{\mathsf{applast(L,X,Last)}\}, \{1\}), \\
& \ (\mathsf{last(Last,LX)}, \{\mathsf{applast(L,X,Last)}\}, \{2\}), \\
& \{\mathsf{applast(L,X,Last)}\}\rangle \\
\xrightarrow{unf}_{\{L/[H|T],LX/[A|S]\}} & \langle(\mathsf{append(T,[X],S)}, \{\mathsf{applast(L,X,Last)}\}, \{1\}), \\
& \ (\underline{\mathsf{last(Last,[A|S])}}, \{\mathsf{applast(L,X,Last)}\}, \{2\}), \\
& \{\mathsf{applast(L,X,Last)}, \mathsf{append(L,[X],LX)}\}\rangle \\
\xrightarrow{unf}_{\{\,\}} & \langle(\mathsf{append(T,[X],S)}, \{\mathsf{applast(L,X,Last)}\}, \{1\}), \\
& \ (\mathsf{last(Last,S)}, \{\mathsf{applast(L,X,Last)}\}, \{2\}), \\
& \{\mathsf{applast(L,X,Last)}, \mathsf{append(L,[X],LX)}, \mathsf{last(Last,[A|S])}\}\rangle \\
\xrightarrow{variant} & \langle\,\rangle
\end{array}
$$

4 Post-Processing Stage

Once the partial evaluation stage terminates, we produce renamed, residual rules associated to the transitions of the partial evaluation semantics as follows:

[7] Observe that both append/3 and `last`/2 are terminating when the first argument is ground, so unfolding non-leftmost atoms is admissible.

- For every unfolding step $\langle Q, memo\rangle \xrightarrow{unf}_{\sigma} \langle Q', memo'\rangle$, we produce a binary clause of the form $ren(query(Q))\sigma \leftarrow ren(query(Q'))$., where $ren(query(Q))$ and $ren(query(Q'))$ are *renamings* (atoms with fresh predicate symbols) of queries $query(Q)$ and $query(Q')$, respectively.
  Actually, we observed that our scheme returns binary residual programs often (in particular, when the source programs are B-stratifiable [6]).
- For every embedding step $\langle Q, memo\rangle \xrightarrow{emb} \langle\ \rangle$, we produce a residual rule of the form $ren(Q) \leftarrow Q$., i.e., we give up the specialization of this query and just call the predicates of the original program.
- Finally, for every branching performed with the rules for independent or regular splitting in which Q is decomposed into the set of queries $\{Q_1, \ldots, Q_n\}$, we produce a residual rule of the form $ren(Q) \leftarrow ren(Q_1), \ldots, ren(Q_n)$.

We do not present the details of the renaming function here since it is a standard renaming as introduced in, e.g., [5].

For instance, for the derivation of Example 2, we produce the following residual rules:

```
applast__1(L,X,Last) ← append_last__2(L,X,LX,Last).
append_last__2([H|T],X,[A|S],Last) ← append_last__3(T,X,A,S,Last).
append_last__3(T,X,A,S,Last) ← append_last__2(T,X,S,Last).
```

Besides the extraction of renamed, residual clauses, we also apply a simple post-unfolding transformation. Basically, we unfold all *intermediate* predicates, i.e., predicates that are only called from one program point. This is very effective for reducing the size of the residual program and is very easy to implement. For instance, in the above clauses, `append_last__3` is just an intermediate predicate and can be removed:

```
applast__1(L,X,Last) ← append_last__2(L,X,LX,Last).
append_last__2([H|T],X,[A|S],Last) ← append_last__2(T,X,S,Last).
```

5 Experimental Evaluation

A prototype implementation of the partial evaluator described so far has been developed. It consists of approx. 1000 lines of SWI Prolog code (including the call and success pattern analysis, comments, etc). The only missing component is the left-termination analysis (so the preservation of finite failures is not yet ensured). A web interface to our tool is publicly available at

```
http://german.dsic.upv.es/lite.html
```

We have tested it by running the benchmarks of the DPPD library [9] that do not contain cut nor negation. Basic built-in's are considered in a simple way: they are evaluated when enough information is provided at partial evaluation time, or left untouched otherwise. Nevertheless, the design of the partial evaluator is not stable yet and some decisions (especially those related with the treatment of built-in's) can be reconsidered in the near future.

Table 1 summarizes our experimental results for some selected benchmarks; for every benchmark, we show the number of inferences[8] (using the `time/1` utility of SWI Prolog) of some run time queries (columns `r1`, `r2`, etc) in the original (row **original**) and partially evaluated programs with both the CPD system ECCE [10] (row **ecce**) and our prototype implementation (row **lite**). All information regarding the source programs, the partial deduction queries, the partially evaluated programs, the run time queries, etc, can be found in the DPPD library [9] and in the web page mentioned above. We do not show the run-times of the partial evaluator since it was really fast: it took less than 0.1 seconds in all but two examples—*liftsolve.app* and *relative*, where it took 0.380 and 0.318 seconds, respectively—with an average running time of 0.042 seconds.

Table 1. Experimental evaluation

benchmark	*advisor*		*applast*	*contains*	*contains.kmp*	*depth*	*doubleapp*		
	`r1`	`r2`	`r1`	`r1`	`r1`	`r1`	`r1`	`r2`	`r3`
original	4	6	58	75	104	24	3	11	50
ecce	0	0	29	18	21	0	2	7	33
lite	0	1	29	18	21	1	3	8	34

benchmark	*ex_depth**			*flip**		*grammar**	*liftsolve.app*		*matchapp*			*match**
	`r1`	`r2`	`r3`	`r1`	`r2`	`r1`	`r1`	`r2`	`r1`	`r2`	`r3`	`r1`
original	24	32	17	6	18	86	16	81	167	206	374	53
ecce	6	9	7	3	9	5	0	1	14	16	23	52
lite	14	0	12	2	24	23	0	1	14	16	23	52

benchmark	*maxlength**			*regexp.r1*			*regexp.r2*			*regexp.r3*			*relative*	*transpose*
	`r1`	`r2`	`r3`	`r1`	`r2`	`r3`	`r1`	`r2`	`r3`	`r1`	`r2`	`r3`	`r1`	`r1`
original	35	3	72	73	85	14	28	27	24	41	39	63	96	58
ecce	26	1	54	10	11	2	10	10	7	14	14	21	0	0
lite	34	0	71	10	11	2	8	8	5	14	14	21	1	0

A detailed experimental comparison between ECCE and our new approach is beyond the scope of this paper. Nevertheless, we note that 5 out of 18 benchmarks in Table 1 produced residual programs which are not closed (i.e., which contained

[8] We do not show actual run-times since they do not add significant new information over the number of inferences. Basically, when the number of inferences coincide, the programs obtained by ECCE and our approach are almost identical. In the remaining cases, where non-closed programs are obtained, the partially evaluated programs obtained by our approach resemble the original non-specialized ones though some inference steps are saved, but the run-times are not significantly changed.

calls to the predicates of the original program because rule embedding was applied during the partial evaluation process). These benchmarks are marked with an asterisk in Table 1. In general, we observed that around 40% of the benchmarks produced non-closed residual programs.

It is worthwhile to note that, when a closed program is obtained, the improvement achieved by our partial evaluator is similar to that obtained by the state-of-the-art system ECCE. However, when a non-closed residual program is obtained the efficiency of the partially evaluated program is rather variable (though no significant slowdown w.r.t. the original programs was produced).

In summary, our technique was only effective in 60% of the considered benchmarks. Nevertheless, this situation could be improved in a number of ways. First, one can add more accurate run-time information (e.g., from a sharing and freeness analysis) so that regular splitting—which usually involves a significant loss of accuracy—is avoided as much as possible. Also, one could introduce a limited form of generalization so that our technique gives up (and thus returns calls to the original predicates) in fewer cases. These topics are subject on ongoing research.

6 Concluding Remarks and Future Work

We have developed a lightweight approach to conjunctive partial deduction that combines features from both online and offline styles of partial evaluation. The resulting scheme is conceptually simpler than existing approaches (thus making it more amenable to predicting the result of partial evaluation) and introduces for the first time the use of run time information to assist the splitting of conjunctions.

The correctness of our approach would not be difficult to prove. In general, it can be seen as an instance of the CPD framework [5]. The main difference comes from the fact that our "one-step" unfoldings do not generally fulfill the weakly fair condition of [5] (which is required for ensuring correctness w.r.t. finite failures). This is solved in our approach by requiring non-leftmost unfolding to be only applicable over terminating calls.

As for future work, our main line of research involves the addition of (run-time) variable sharing information. Besides improving the accuracy of splitting, the combination of (run-time) sharing and groundness information can be very useful at partial evaluation time in order to produce residual programs where some sequential conjunctions are automatically replaced by *concurrent* conjunctions. Some preliminary experiments in this direction (using the `concurrent/3` predicate of SWI Prolog) have shown promising results.

Acknowledgements

We gratefully acknowledge the anonymous referees as well as the participants of LOPSTR 2010 for many useful comments and suggestions.

References

1. A. Ben-Amram and M. Codish. A SAT-Based Approach to Size Change Termination with Global Ranking Functions. In C.R. Ramakrishnan and Jakob Rehof, editors, *Proc. of the 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, pages 46–55. Springer LNCS 5028, 2008.
2. M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding during Partial Deduction of Logic Programs. In V. Saraswat and K. Ueda, editors, *Proc. 1991 Int'l Symp. on Logic Programming*, pages 117–131, 1991.
3. N.H. Christensen and R. Glück. Offline Partial Evaluation Can Be as Accurate as Online Partial Evaluation. *ACM Transactions on Programming Languages and Systems*, 26(1):191–220, 2004.
4. M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
5. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.
6. J. Hruza and P. Stepánek. Speedup of logic programs by binarization and partial deduction. *TPLP*, 4(3):355–380, 2004.
7. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
8. M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, pages 379–403. Springer LNCS 2566, 2002.
9. M. Leuschel. The DPPD (Dozens of Problems for Partial Deduction) Library of Benchmarks, 2007. Available at URL: `http://www.ecs.soton.ac.uk/~mal/systems/dppd.html`.
10. M. Leuschel, D. Elphick, M. Varea, S. Craig, and M. Fontaine. The Ecce and Logen Partial Evaluators and Their Web Interfaces. In *Proc. of PEPM'06*, pages 88–94. IBM Press, 2006.
11. M. Leuschel and G. Vidal. Fast Offline Partial Evaluation of Large Logic Programs. In *Proc. of the 18th Int'l Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2008)*, pages 119–134. Springer LNCS 5438, 2009.
12. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
13. Z. Somogyi. A System of Precise Modes for Logic Programs. In E.Y. Shapiro, editor, *Proc. of Third Int'l Conf. on Logic Programming*, pages 769–787. The MIT Press, Cambridge, MA, 1986.