A Hybrid Approach to Conjunctive Partial Deduction

Germán Vidal

Technical University of Valencia

Int'l Symp. on Logic-Based Program Synthesis and Transformation
LOPSTR 2010

July 23-25, 2010
Castle of Hagenberg, Austria
Introduction

Partial evaluation

- **input** program and part of input data (*static* data)
- **output** specialized (*residual*) program

Partial evaluator

- constructs a finite representation of all possible computations
- extracts *resultants* from transitions

Optimization comes from

- compressing paths in the graph (linear speedups for loops)
- renaming of expressions (removes unnecessary symbols)
Introduction

Partial evaluation

- **input** program and part of input data (*static* data)
- **output** specialized (*residual*) program

Partial evaluator

- constructs a finite representation of all possible computations
- extracts *resultants* from transitions

Optimization comes from

- compressing paths in the graph (linear speedups for loops)
- renaming of expressions (removes unnecessary symbols)
Introduction

Partial evaluation

- **input** program and part of input data (static data)
- **output** specialized (residual) program

Partial evaluator

- constructs a finite representation of all possible computations
- extracts **resultants** from transitions

Optimization comes from

- compressing paths in the graph (linear speedups for loops)
- renaming of expressions (removes unnecessary symbols)
Conjunctive partial deduction

**Input** logic program $P$ and a query $Q_0$

**Initialization** $S = \{Q_0\}$  $S = \{Q_0, Q_3, Q_4, Q_5\}$  $S = \{Q_0, Q_3, Q_4, Q_5, Q_6\}$

The set is kept finite using
- generalization
- splitting

(instance of $Q_0$)
Conjunctive partial deduction

**Input** logic program $P$ and a query $Q_0$

**Initialization** $S = \{Q_0\}$ $S = \{Q_0, Q_3, Q_4, Q_5\}$ $S = \{Q_0, Q_3, Q_4, Q_5, Q_6\}$

The set is kept finite using
- generalization
- splitting

(instance of $Q_0$)
**Conjunctive partial deduction**

**Input** logic program $P$ and a query $Q_0$

**Initialization**

$\mathcal{S} = \{ Q_0 \}$

$\mathcal{S} = \{ Q_0, Q_3, Q_4, Q_5 \}$

$\mathcal{S} = \{ Q_0, Q_3, Q_4, Q_5, Q_6 \}$

The set is kept finite using

- generalization
- splitting

(instance of $Q_0$)
Conjunctive partial deduction

**Input** logic program \( P \) and a query \( Q_0 \)

**Initialization**
\[
S = \{ Q_0 \} \quad S = \{ Q_0, Q_3, Q_4, Q_5 \} \quad S = \{ Q_0, Q_3, Q_4, Q_5, Q_6 \}
\]

The set is kept finite using
- generalization
- splitting

(instance of \( Q_0 \))
Conjunctive partial deduction

**Input** logic program \( P \) and a query \( Q_0 \)

**Initialization**
\[
S = \{ Q_0 \} \quad S = \{ Q_0, Q_3, Q_4, Q_5 \} \quad S = \{ Q_0, Q_3, Q_4, Q_5, Q_6 \}
\]

The set is kept finite using
- generalization
- splitting

(instance of \( Q_0 \))
Conjunctive partial deduction

**Input** logic program $P$ and a query $Q_0$

**Initialization** $S = \{ Q_0 \}$ $S = \{ Q_0, Q_3, Q_4, Q_5 \}$ $S = \{ Q_0, Q_3, Q_4, Q_5, Q_6 \}$

The set is kept finite using
- generalization
- splitting

(instance of $Q_0$)
Conjunctive partial deduction

**Input** logic program $P$ and a query $Q_0$

**Initialization** $S = \{ Q_0 \}$ $S = \{ Q_0, Q_3, Q_4, Q_5 \}$ $S = \{ Q_0, Q_3, Q_4, Q_5, Q_6 \}$

The set is kept finite using
- generalization
- splitting

(instance of $Q_0$)
This work

Original motivation:

- **paralzelizing** partial evaluation?
- run time groundness and sharing information is essential

Current approaches not useful because

- run time information is not available (only PE time info)
- usual operations (instance and splitting) do not preserve groundness and sharing

Our approach:

- hybrid control issues (combines static analysis and online tests)
- run time groundness information available
- good starting point for paralelizing partial evaluation
This work

Original motivation:

- parallelizing partial evaluation?
- run time groundness and sharing information is essential

Current approaches not useful because

- run time information is not available (only PE time info)
- usual operations (instance and splitting) do not preserve groundness and sharing

Our approach:

- hybrid control issues (combines static analysis and online tests)
- run time groundness information available
- good starting point for parallelizing partial evaluation
This work

Original motivation:

- parallelizing partial evaluation?
- run time groundness and sharing information is essential

Current approaches not useful because

- run time information is not available (only PE time info)
- usual operations (instance and splitting) do not preserve groundness and sharing

Our approach:

- hybrid control issues (combines static analysis and online tests)
- run time groundness information available
- good starting point for parallelizing partial evaluation
This work

Original motivation:
- parallelizing partial evaluation?
- run time groundness and sharing information is essential

Current approaches not useful because
- run time information is not available (only PE time info)
- usual operations (instance and splitting) do not preserve groundness and sharing

Our approach:
- hybrid control issues (combines static analysis and online tests)
- run time groundness information available
- good starting point for parallelizing partial evaluation
Lightweight CPD

1 Pre-processing
   - call and success pattern analysis
   - left-termination analysis
   - identification of non-regular predicates

2 Partial evaluation
   - non-leftmost unfolding statically determined
   - only a limited form of splitting (statically determined)
   - no generalization (but might give up)

3 Post-processing
   - initially one-step renamed resultants
   - post-unfolding transition compression to avoid intermediate calls
Lightweight CPD

1. Pre-processing
   - call and success pattern analysis
   - left-termination analysis
   - identification of non-regular predicates

2. Partial evaluation
   - non-leftmost unfolding statically determined
   - only a limited form of splitting (statically determined)
   - no generalization (but might give up)

3. Post-processing
   - initially one-step renamed resultants
   - post-unfolding transition compression to avoid intermediate calls
Static analyses

Call and success pattern analysis (e.g., [Leuschel and Vidal, LOPSTR’08])

- for each predicate $p/n$, we get a set of patterns $p/n : \text{in} \mapsto \text{out}$
- e.g., append/3 : $\{1, 2\} \mapsto \{1, 2, 3\}$

\[
\begin{align*}
\text{append}([\ ], Y, Y). \\
\text{append}([X|R], Y, [X|S]) & : -\text{append}(R, Y, S).
\end{align*}
\]

Left-termination analysis

- determines if $p/n$ terminates for call pattern $\text{in}$ with Prolog’s leftmost selection strategy
- e.g., append/3 left-terminates for call pattern $\{1\}$
- e.g., append/3 doesn’t left-terminate for call pattern $\{2\}$
**Static analyses**

Call and success pattern analysis (e.g., [Leuschel and Vidal, LOPSTR’08])

- for each predicate $p/n$, we get a set of patterns $p/n : \text{in} \leftrightarrow \text{out}$
- e.g., append/3 : $\{1, 2\} \leftrightarrow \{1, 2, 3\}$

```prolog
append([], Y, Y).
append([X|R], Y, [X|S]) : ~append(R, Y, S).
```

Left-termination analysis

- determines if $p/n$ terminates for call pattern $\text{in}$ with Prolog’s leftmost selection strategy
- e.g., append/3 left-terminates for call pattern $\{1\}$
- e.g., append/3 doesn’t left-terminate for call pattern $\{2\}$
Strongly regular programs

Extends B-stratifiable programs [Hruza and Stepánek, TPLP 2004]:

- first, the call graph of the program is built
- predicate \( p/n \) is strongly regular if there is no

\[
p(t_1, \ldots, t_n) \leftarrow \text{body}
\]

such that \( \text{body} \) contains two atoms in the same SCC as \( p/n \)
- a logic program is strongly regular if all predicates are

**Property:** SRP cannot produce infinitely growing conjunctions at PE time

Identifying non-regular predicates will become useful to decide how to split queries at partial evaluation time
Strongly regular programs

Extends B-stratifiable programs [Hruza and Stepánek, TPLP 2004]:

- first, the call graph of the program is built
- predicate $p/n$ is strongly regular if there is no

$$p(t_1, \ldots, t_n) \leftarrow \text{body}$$

such that $\text{body}$ contains two atoms in the same SCC as $p/n$
- a logic program is strongly regular if all predicates are

**Property:** SRP cannot produce infinitely growing conjunctions at PE time

Identifying non-regular predicates will become useful to decide how to split queries at partial evaluation time
Strongly regular programs

Extends B-stratifiable programs [Hruza and Stepánek, TPLP 2004]:

- first, the call graph of the program is built
- predicate \( p/n \) is strongly regular if there is no
  \[
  p(t_1, \ldots, t_n) \leftarrow \text{body}
  \]
  such that \( \text{body} \) contains two atoms in the same SCC as \( p/n \)
- a logic program is strongly regular if all predicates are

**Property:** SRP cannot produce infinitely growing conjunctions at PE time

Identifying non-regular predicates will become useful to decide how to split queries at partial evaluation time
Strongly regular programs

Extends B-stratifiable programs [Hruza and Stepánek, TPLP 2004]:
- first, the call graph of the program is built
- predicate $p/n$ is strongly regular if there is no

$$p(t_1, \ldots, t_n) \leftarrow \text{body}$$

such that $\text{body}$ contains two atoms in the same SCC as $p/n$
- a logic program is strongly regular if all predicates are

**Property:** SRP cannot produce infinitely growing conjunctions at PE time

Identifying non-regular predicates will become useful to decide how to split queries at partial evaluation time
Strongly regular programs

Extends B-stratifiable programs [Hruza and Stepánek, TPLP 2004]:

- first, the call graph of the program is built
- predicate $p/n$ is strongly regular if there is no
  $$p(t_1, \ldots, t_n) \leftarrow \text{body}$$

  such that $\text{body}$ contains two atoms in the same SCC as $p/n$
- a logic program is strongly regular if all predicates are

**Property:** SRP cannot produce infinitely growing conjunctions at PE time

Identifying non-regular predicates will become useful to decide how to split queries at partial evaluation time
Strongly regular programs

Extends B-stratifiable programs [Hruza and Stepánek, TPLP 2004]:

- first, the call graph of the program is built
- predicate \( p/n \) is strongly regular if there is no

\[
p(t_1, \ldots, t_n) \leftarrow \text{body}
\]

such that \( \text{body} \) contains two atoms in the same SCC as \( p/n \)
- a logic program is strongly regular if all predicates are

**Property:** SRP cannot produce infinitely growing conjunctions at PE time

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

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

- 3 SCCs: \{applast/3\}, \{append/3\} and \{last/2\}
- no clause violates the strongly regular condition

Example (not strongly regular)

\[
\text{flipflip}(XT, YT) : -\text{flip}(XT, TT), \text{flip}(TT, YT).
\]
\[
\text{flip}(\text{leaf}(X), \text{leaf}(X)).
\]
\[
\text{flip}(\text{tree}(L, I, R), \text{tree}(FR, I, FL)) : -\text{flip}(L, FL), \text{flip}(R, FR).
\]

- 2 SCCs: \{flipflip/2\} and \{flip/2\}
- the second clause of \text{flip}/2 violates the strongly regular condition
Example (strongly regular)

\[
\text{applast}(L, X, \text{Last}) : \neg \text{append}(L, [X], \text{LX}), \text{last}((\text{Last}, \text{LX}).
\]
\[
\text{last}(X, [X]).
\]
\[
\text{last}(X, [H|T]) : \neg \text{last}(X, T).
\]
\[
\text{append}([], L, L).
\]
\[
\text{append}([H|L1], L2, [H|L3]) : \neg \text{append}(L1, L2, L3).
\]

- 3 SCCs: \{\text{applast}/3\}, \{\text{append}/3\} and \{\text{last}/2\}
- no clause violates the strongly regular condition

Example (not strongly regular)

\[
\text{flipflip}(XT, YT) : \neg \text{flip}(XT, TT), \text{flip}(TT, YT).
\]
\[
\text{flip}(\text{leaf}(X), \text{leaf}(X)).
\]
\[
\text{flip}(\text{tree}(L, I, R), \text{tree}(FR, I, FL)) : \neg \text{flip}(L, FL), \text{flip}(R, FR).
\]

- 2 SCCs: \{\text{flipflip}/2\} and \{\text{flip}/2\}
- the second clause of \text{flip}/2 violates the strongly regular condition
Lightweight CPD

1. Pre-processing
   - call and success pattern analysis
   - left-termination analysis
   - identification of non-regular predicates

2. Partial evaluation
   - non-leftmost unfolding statically determined
   - only a limited form of splitting (statically determined)
   - no generalization (but might give up)

3. Post-processing
   - initially one-step renamed resultants
   - post-unfolding transition compression to avoid intermediate calls
Lightweight CPD

1 Pre-processing
   - call and success pattern analysis
   - left-termination analysis
   - identification of non-regular predicates

2 Partial evaluation
   - non-leftmost unfolding statically determined
   - only a limited form of splitting (statically determined)
   - no generalization (but might give up)

3 Post-processing
   - initially one-step renamed resultants
   - post-unfolding transition compression to avoid intermediate calls
Partial evaluation: global level

Global state:

\[ \langle\langle\{qs_1, \ldots, qs_n\}, gs\rangle\rangle \]

where

- \(\{qs_1, \ldots, qs_n\}\) is a set of queries (with call patterns)
- \(gs\) is the set of already partially evaluated queries

Initial global state: \(\langle\langle\{qs\}, \emptyset\rangle\rangle\)

Transition system

(restart)

\[ \forall qs' \in gs. \quad qs_i \supseteq qs', \quad i \in \{1, \ldots, n\} \]

\[ \langle\langle\{qs_1, \ldots, qs_n\}, gs\rangle\rangle \rightarrow \langle qs_i, [], \{qs_i\} \cup gs \rangle \]

(stop)

\[ \exists qs' \in gs. \quad qs_i \supseteq qs', \quad i \in \{1, \ldots, n\} \]

\[ \langle\langle\{qs_1, \ldots, qs_n\}, gs\rangle\rangle \rightarrow qs_i \langle\langle\rangle\rangle \]
Partial evaluation: global level

Global state:

$$\langle\langle\{qs_1, \ldots, qs_n\}, gs\rangle\rangle$$

where
- $$\{qs_1, \ldots, qs_n\}$$ is a set of queries (with call patterns)
- $$gs$$ is the set of already partially evaluated queries

Initial global state: $$\langle\langle\{qs\}, \emptyset\rangle\rangle$$

Transition system

(restart)  $$\forall qs' \in gs. \; qs_i \supseteq qs', \; i \in \{1, \ldots, n\}$$

$$\langle\langle\{qs_1, \ldots, qs_n\}, gs\rangle\rangle \rightarrow \langle qs_i, [\ ], \{qs_i\} \cup gs\rangle$$

(stop)  $$\exists qs' \in gs. \; qs_i \supseteq qs', \; i \in \{1, \ldots, n\}$$

$$\langle\langle\{qs_1, \ldots, qs_n\}, gs\rangle\rangle \rightarrow qs_i \langle\langle\rangle\rangle$$
Partial evaluation: global level

Global state:

⟨⟨{qs₁, ..., qsₙ}, gs⟩⟩

where

- \{qs₁, ..., qsₙ\} is a set of queries (with call patterns)
- gs is the set of already partially evaluated queries

Initial global state:  ⟨⟨{qs}, ∅⟩⟩

Transition system

(restart)

\[
\forall qs' \in gs. \forall i \in \{1, ..., n\} \frac{\langle\langle\{qs₁, ..., qsₙ\}, gs\rangle\rangle \rightarrow \langleqs_i, [], \{qs_i\} \cup gs\rangle}{\langle\langle\{qs₁, ..., qsₙ\}, gs\rangle\rangle \rightarrow qs_i \langle\langle\rangle\rangle}
\]

(stop)
Partial evaluation: global level

Global state:

$$\langle\langle\{qs_1, \ldots, qs_n\}, gs\rangle\rangle$$

where

- $$\{qs_1, \ldots, qs_n\}$$ is a set of queries (with call patterns)
- $$gs$$ is the set of already partially evaluated queries

Initial global state: $$\langle\langle\{qs\}, \emptyset\rangle\rangle$$

Transition system

(restart)

$$\forall qs' \in gs. \ \forall i \in \{1, \ldots, n\}$$
$$\langle\langle\{qs_1, \ldots, qs_n\}, gs\rangle\rangle \rightarrow \langle qs_i, [], \{qs_i\} \cup gs\rangle$$

(stop)

$$\exists qs' \in gs. \ \forall i \in \{1, \ldots, n\}$$
$$\langle\langle\{qs_1, \ldots, qs_n\}, gs\rangle\rangle \rightarrow qs_i \langle\langle\rangle\rangle$$
Partial evaluation: global level

Global state:

\[\langle\langle \{qs_1, \ldots, qs_n\}, gs\rangle\rangle\]

where

- \(\{qs_1, \ldots, qs_n\}\) is a set of queries (with call patterns)
- \(gs\) is the set of already partially evaluated queries

Initial global state: \(\langle\langle \{qs\}, \emptyset\rangle\rangle\)

Transition system

\[
\begin{align*}
\text{(restart)} & \quad \forall qs' \in gs. \; qs_i \supseteq qs', \; i \in \{1, \ldots, n\} \\
& \frac{}{\langle\langle \{qs_1, \ldots, qs_n\}, gs\rangle\rangle \rightarrow \langle qs_i, [\;], \{qs_i\} \cup gs\rangle}
\end{align*}
\]

\[
\begin{align*}
\text{(stop)} & \quad \exists qs' \in gs. \; qs_i \supseteq qs', \; i \in \{1, \ldots, n\} \\
& \frac{}{\langle\langle \{qs_1, \ldots, qs_n\}, gs\rangle\rangle \rightarrow qs_i \langle\langle \rangle\rangle}
\end{align*}
\]
Partial evaluation: local level

Local states:

\[ \langle qs, ls, gs \rangle \]

where

- \( qs \) is a query (with call patterns)
- \( ls \) is the local stack (queries already processed in the local level)
- \( gs \) is the global stack (queries already processed in the global level)
Partial evaluation: local level

Local states:

\[ \langle qs, ls, gs \rangle \]

where

- \( qs \) is a query (with call patterns)
- \( ls \) is the local stack (queries already processed in the local level)
- \( gs \) is the global stack (queries already processed in the global level)
Partial evaluation: local level

**Local states:**

\[ \langle qs, ls, gs \rangle \]

where

- *qs* is a query (with call patterns)
- *ls* is the *local stack* (queries already processed in the local level)
- *gs* is the *global stack* (queries already processed in the global level)
Partial evaluation: local level

Local states:

\[ \langle qs, ls, gs \rangle \]

where

- \( qs \) is a query (with call patterns)
- \( ls \) is the *local stack* (queries already processed in the local level)
- \( gs \) is the *global stack* (queries already processed in the global level)
Definition (unfoldable atom)

- it doesn’t embed any previous call
- leftmost atom or left-terminating for the associated call pattern

(to ensure correctness w.r.t. finite failures, instead of requiring weakly fair SLD trees [De Schreye et al, JLP 99])

For instance, given the query \( p(a), q(X) \) and the program

\[
\begin{align*}
p(b). \\
q(X) : -q(X).
\end{align*}
\]

the derivation \( p(a), q(X) \leadsto p(a), q(X) \) is not weakly fair
(thus \( pq(X) : -pq(X). \) is not a legal resultant)

In our context, \( q(X) \) is not unfoldable (not left-terminating)
Definition (unfoldable atom)

- it doesn’t embed any previous call
- leftmost atom or left-terminating for the associated call pattern
  (to ensure correctness w.r.t. finite failures, instead of requiring weakly fair SLD trees [De Schreye et al, JLP 99])

For instance, given the query \( p(a), q(X) \) and the program

\[
\begin{align*}
 p(b). \\
 q(X) : -q(X).
\end{align*}
\]

the derivation \( p(a), q(X) \rightarrow p(a), q(X) \) is not weakly fair (thus \( pq(X) : -pq(X) \) is not a legal resultant)

In our context, \( q(X) \) is not unfoldable (not left-terminating)
Definition (unfoldable atom)

- it doesn’t embed any previous call
- leftmost atom or left-terminating for the associated call pattern
  (to ensure correctness w.r.t. finite failures, instead of requiring weakly fair SLD trees [De Schreye et al, JLP 99])

For instance, given the query $p(a), q(X)$ and the program

$$
\begin{align*}
p(b). \\
q(X) : \neg q(X).
\end{align*}
$$

the derivation $p(a), q(X) \leadsto p(a), q(X)$ is not weakly fair
(thus $pq(X) : \neg pq(X)$. is not a legal resultant)

In our context, $q(X)$ is not unfoldable (not left-terminating)
Splitting

Definition (independent splitting)

Given a query $qs$, we have that $qs_1, qs_2, qs_3$ is an independent splitting if

- $qs = qs_1, qs_2, qs_3$
- $qs_1$ and $qs_2$ do not share variables (according to call patterns)

For instance, given the query

$$qs = \text{append}(X, Y, L_1), \text{append}(X, Z, L_2), \text{append}(L_1, L_2, R)$$

the independent splitting of $qs$ returns

$$qs_1 = \text{append}(X, Y, L_1)$$
$$qs_2 = \text{append}(X, Z, L_2)$$
$$qs_3 = \text{append}(L_1, L_2, R)$$
Splitting

**Definition (independent splitting)**

Given a query $qs$, we have that $qs_1, qs_2, qs_3$ is an independent splitting if

- $qs = qs_1, qs_2, qs_3$
- $qs_1$ and $qs_2$ do not share variables (according to call patterns)

For instance, given the query

$$qs = \text{append}(X, Y, L_1), \text{append}(X, Z, L_2), \text{append}(L_1, L_2, R)$$

the independent splitting of $qs$ returns

$$qs_1 = \text{append}(X, Y, L_1)$$
$$qs_2 = \text{append}(X, Z, L_2)$$
$$qs_3 = \text{append}(L_1, L_2, R)$$
Definition (regular splitting)

Given a query \( qs \), we have that \( qs_1, \ldots, qs_n \) is a regular splitting if

- \( qs = qs_1, \ldots, qs_n \)
- every \( qs_i \) contains at most one non-regular predicate

For instance, the regular splitting of

\[
\text{flip}(L, FL), \text{flip}(R, FR)
\]

is

\[
qs_1 = \text{flip}(L, FL) \\
qs_2 = \text{flip}(R, FR)
\]

since \( \text{flip}/2 \) is non-regular.
Definition (regular splitting)

Given a query $qs$, we have that $qs_1, \ldots, qs_n$ is a regular splitting if

- $qs = qs_1, \ldots, qs_n$
- every $qs_i$ contains at most one non-regular predicate

For instance, the regular splitting of

$$\text{flip}(L, FL), \text{flip}(R, FR)$$

is

$$qs_1 = \text{flip}(L, FL)$$
$$qs_2 = \text{flip}(R, FR)$$

since $\text{flip}/2$ is non-regular
Partial evaluation: local level

\[
\begin{align*}
\text{(variant)} & \quad \exists qs' \in ls. \, qs \approx qs' \\
& \quad \langle qs, ls, gs \rangle \, \Rightarrow \, \langle \diamond, ls, gs \rangle
\end{align*}
\]

\[
\begin{align*}
\text{(independent splitting)} & \quad \text{i-split}(qs) = \langle qs_1, qs_2, qs_3 \rangle \\
& \quad \langle qs, ls, gs \rangle \, \Rightarrow \, \langle \{ qs_1, qs_2, qs_3 \}, gs \rangle
\end{align*}
\]

\[
\begin{align*}
\text{(unfold)} & \quad \text{unfold}(qs) = qs' \\
& \quad \langle qs, ls, gs \rangle \, \Rightarrow \, \langle qs', \{ qs \} \cup ls, gs \rangle
\end{align*}
\]

\[
\begin{align*}
\text{(regular splitting)} & \quad \text{r-split}(qs) = \langle qs_1, \ldots, qs_n \rangle \\
& \quad \langle qs, ls, gs \rangle \, \Rightarrow \, \langle \{ qs_1, \ldots, qs_n \}, gs \rangle
\end{align*}
\]
Partial evaluation: local level

(variant) \[
\exists qs' \in ls. \ \forall qs \approx qs'
\]
\[
\langle qs, ls, gs \rangle \ \Rightarrow \ \langle \diamond, ls, gs \rangle
\]

(independent splitting) \[
i-split(qs) = \langle qs_1, qs_2, qs_3 \rangle
\]
\[
\langle qs, ls, gs \rangle \ \Rightarrow \ \langle \langle \{qs_1, qs_2, qs_3\}, gs \rangle \rangle
\]

(unfold) \[
unfold(qs) = qs'
\]
\[
\langle qs, ls, gs \rangle \ \Rightarrow \ \langle qs', \{qs\} \cup ls, gs \rangle
\]

(regular splitting) \[
r-split(qs) = \langle qs_1, \ldots, qs_n \rangle
\]
\[
\langle qs, ls, gs \rangle \ \Rightarrow \ \langle \langle \{qs_1, \ldots, qs_n\}, gs \rangle \rangle
\]
Partial evaluation: local level

\[ \exists q_s' \in l_s. \ q_s \approx q_s' \]
\[ \langle q_s, l_s, g_s \rangle \Rightarrow \langle \diamond, l_s, g_s \rangle \]

(i-splitting)
\[ \text{i-split}(q_s) = \langle q_{s1}, q_{s2}, q_{s3} \rangle \]
\[ \langle q_s, l_s, g_s \rangle \Rightarrow \langle \langle \{q_{s1}, q_{s2}, q_{s3}\}, g_s \rangle \rangle \]

(unfold)
\[ \text{unfold}(q_s) = q_s' \]
\[ \langle q_s, l_s, g_s \rangle \Rightarrow \langle q_s', \{q_s\} \cup l_s, g_s \rangle \]

(regular splitting)
\[ \text{r-split}(q_s) = \langle q_{s1}, \ldots, q_{s_n} \rangle \]
\[ \langle q_s, l_s, g_s \rangle \Rightarrow \langle \langle \{q_{s1}, \ldots, q_{s_n}\}, g_s \rangle \rangle \]
Partial evaluation: local level

(variant) \[
\exists qs' \in ls. \; qs \approx qs' \\
\langle qs, ls, gs \rangle \Rightarrow \langle \Diamond, ls, gs \rangle
\]

(independent splitting) \[
i\text{-split}(qs) = \langle qs_1, qs_2, qs_3 \rangle \\
\langle qs, ls, gs \rangle \xrightarrow{i} \langle \langle \{qs_1, qs_2, qs_3\}, gs \rangle \rangle
\]

(unfold) \[
\text{unfold}(qs) = qs' \\
\langle qs, ls, gs \rangle \Rightarrow_{\sigma} \langle qs', \{qs\} \cup ls, gs \rangle
\]

(regular splitting) \[
r\text{-split}(qs) = \langle qs_1, \ldots, qs_n \rangle \\
\langle qs, ls, gs \rangle \Rightarrow \langle \langle \{qs_1, \ldots, qs_n\}, gs \rangle \rangle
\]
Lightweight CPD

1. Pre-processing
   - call and success pattern analysis
   - left-termination analysis
   - identification of non-regular predicates

2. Partial evaluation
   - non-leftmost unfolding statically determined
   - only a limited form of splitting (statically determined)
   - no generalization (but might give up)

3. Post-processing
   - initially one-step renamed resultants
   - post-unfolding transition compression to avoid intermediate calls
Lightweight CPD

1. Pre-processing
   - call and success pattern analysis
   - left-termination analysis
   - identification of non-regular predicates

2. Partial evaluation
   - non-leftmost unfolding statically determined
   - only a limited form of splitting (statically determined)
   - no generalization (but might give up)

3. Post-processing
   - initially one-step renamed resultants
   - post-unfolding transition compression to avoid intermediate calls
Post-processing

- For \( \langle qs, ls, gs \rangle \xrightarrow{u} \langle qs', ls', gs' \rangle \)
  we produce \( \text{ren}(qs) \sigma \leftarrow \text{ren}(qs') \)

- For \( \langle qs, ls, gs \rangle \xrightarrow{s} \langle \langle \{qs_1, \ldots, qs_n\}, \_ \rangle, \_ \rangle \), with \( s \in \{i, r\} \)
  we produce \( \text{ren}(qs) \leftarrow \text{ren}(qs_1), \ldots, \text{ren}(qs_n) \)

- For every global transition \( \langle \langle \{qs_1, \ldots, qs_n\}, \_ \rangle \rangle \rightarrow_{qs_i} \langle \langle \_ \rangle \rangle \)
  we produce a residual clause of the form \( \text{ren}(qs_i) \leftarrow qs_i \)
Post-processing

- For $\langle qs, ls, gs \rangle \xrightarrow{u} \sigma \langle qs', ls', gs' \rangle$
  
  we produce $\text{ren}(qs) \sigma \leftarrow \text{ren}(qs')$

- For $\langle qs, ls, gs \rangle \xrightarrow{s} \langle\langle \{qs_1, \ldots, qs_n\}, \_\rangle\rangle$, with $s \in \{i, r\}$
  
  we produce $\text{ren}(qs) \leftarrow \text{ren}(qs_1), \ldots, \text{ren}(qs_n)$

- For every global transition $\langle\langle \{qs_1, \ldots, qs_n\}, \_\rangle\rangle \rightarrow_{qs_i} \langle\_\rangle$
  
  we produce a residual clause of the form $\text{ren}(qs_i) \leftarrow qs_i$
Post-processing

For $\langle qs, ls, gs \rangle \xrightarrow{u}_\sigma \langle qs', ls', gs' \rangle$
we produce $\text{ren}(qs) \sigma \leftarrow \text{ren}(qs')$

For $\langle qs, ls, gs \rangle \xrightarrow{s} \langle \langle \{qs_1, \ldots, qs_n\}, \_ \rangle \rangle$, with $s \in \{i, r\}$
we produce $\text{ren}(qs) \leftarrow \text{ren}(qs_1), \ldots, \text{ren}(qs_n)$

For every global transition $\langle \langle \{qs_1, \ldots, qs_n\}, \_ \rangle \rangle \rightarrow qs_i \langle \langle \_ \rangle \rangle$
we produce a residual clause of the form $\text{ren}(qs_i) \leftarrow qs_i$
Experimental results

A prototype has been implemented (≈ 1000 lines, SWI Prolog) (left-termination and SRP analysis still missing)

http://kaz.dsic.upv.es/lite.html

<table>
<thead>
<tr>
<th>benchmark</th>
<th>advisor</th>
<th>applast</th>
<th>depth</th>
<th>doubleapp</th>
<th>ex_depth</th>
<th>flip</th>
<th>matchapp</th>
<th>regexp.r1</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>4</td>
<td>58</td>
<td>24</td>
<td>50</td>
<td>24</td>
<td>34</td>
<td>374</td>
<td>73</td>
</tr>
<tr>
<td>residual</td>
<td>0</td>
<td>29</td>
<td>1</td>
<td>34</td>
<td>15</td>
<td>47</td>
<td>23</td>
<td>10</td>
</tr>
<tr>
<td>benchmark</td>
<td>regexp.r2</td>
<td>regexp.r3</td>
<td>relative</td>
<td>rev_acc_type</td>
<td>rotateprune</td>
<td>transpose</td>
<td></td>
<td></td>
</tr>
<tr>
<td>original</td>
<td>28</td>
<td>41</td>
<td>96</td>
<td>35</td>
<td>32</td>
<td>58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>residual</td>
<td>8</td>
<td>12</td>
<td>3</td>
<td>34</td>
<td>45</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experimental results

A prototype has been implemented (≈ 1000 lines, SWI Prolog) (left-termination and SRP analysis still missing)

http://kaz.dsic.upv.es/lite.html

<table>
<thead>
<tr>
<th>benchmark</th>
<th>advisor</th>
<th>applast</th>
<th>depth</th>
<th>doubleapp</th>
<th>ex_depth</th>
<th>flip</th>
<th>matchapp</th>
<th>regexp.r1</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>4</td>
<td>58</td>
<td>24</td>
<td>50</td>
<td>24</td>
<td>34</td>
<td>374</td>
<td>73</td>
</tr>
<tr>
<td>residual</td>
<td>0</td>
<td>29</td>
<td>1</td>
<td>34</td>
<td>15</td>
<td>47</td>
<td>23</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>benchmark</th>
<th>regexp.r2</th>
<th>regexp.r3</th>
<th>relative</th>
<th>rev_acc_type</th>
<th>rotateprune</th>
<th>transpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>28</td>
<td>41</td>
<td>96</td>
<td>35</td>
<td>32</td>
<td>58</td>
</tr>
<tr>
<td>residual</td>
<td>8</td>
<td>12</td>
<td>3</td>
<td>34</td>
<td>45</td>
<td>0</td>
</tr>
</tbody>
</table>
Summary and future work

New hybrid framework for CPD (correctness not difficult)

Well suited to preserve run time information (groundness and sharing)

Good candidate to develop a parallelizing partial evaluator

Future work

- deal with built-in’s and negation
- add (run time) variable sharing information
- produce parallel conjunctions in residual programs
  (preliminary experiments with concurrent/3 are promising)
Summary and future work

New hybrid framework for CPD (correctness not difficult)

Well suited to preserve run time information (groundness and sharing)

Good candidate to develop a paralelizing partial evaluator

Future work

- deal with built-in’s and negation
- add (run time) variable sharing information
- produce paralel conjunctions in residual programs
  (preliminary experiments with concurrent/3 are promising)
Summary and future work

New hybrid framework for CPD (correctness not difficult)

Well suited to preserve run time information (groundness and sharing)

Good candidate to develop a parallelizing partial evaluator

Future work

- deal with built-in’s and negation
- add (run time) variable sharing information
- produce parallel conjunctions in residual programs
  (preliminary experiments with concurrent/3 are promising)
Summary and future work

New hybrid framework for CPD (correctness not difficult)

Well suited to preserve run time information (groundness and sharing)

Good candidate to develop a paralelizing partial evaluator

Future work

- deal with built-in’s and negation
- add (run time) variable sharing information
- produce paralel conjuctions in residual programs
  (preliminary experiments with concurrent/3 are promising)