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

- $\boxed{\textbf{input}}$ program and part of input data (static data)
- $\boxed{\textbf{output}}$ 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

- $\boxed{\textbf{input}}$ program and part of input data (static data)
- $\boxed{\textbf{output}}$ 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

- $\boxed{\textbf{input}}$ program and part of input data (static data)
- $\boxed{\textbf{output}}$ 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

Conjunctive partial deduction

Input logic program P and a query Q_0

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

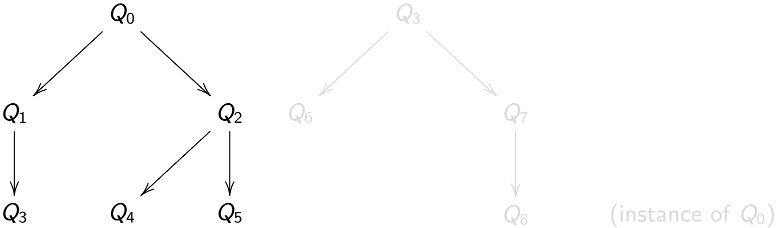
The set is kept finite using

- generalization
- splitting

Conjunctive partial deduction

$\boxed{\text{Input}}$ logic program P and a query Q_0

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



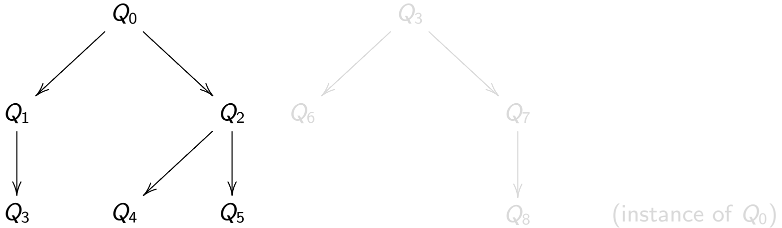
The set is kept finite using

- generalization
- splitting

Conjunctive partial deduction

$\boxed{\text{Input}}$ logic program P and a query Q_0

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



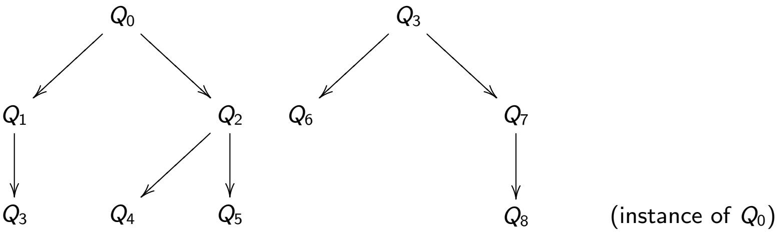
The set is kept finite using

- generalization
- splitting

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

Conjunctive partial deduction

Input logic program P and a query Q_0

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

$$
\begin{tikzcd}
 & Q_0 \arrow[dl] \arrow[dr] & & & Q_3 \arrow[dl] \arrow[dr] & & \\
Q_1 \arrow[d] & & Q_2 \arrow[dl] \arrow[d] & Q_6 & & Q_7 \arrow[d] & \\
Q_3 & Q_4 & Q_5 & & & Q_8 & \text{(instance of } Q_0\text{)}
\end{tikzcd}
$$

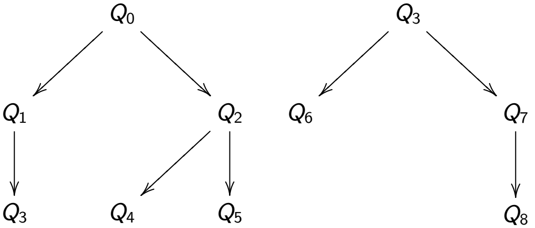
The set is kept finite using

- generalization
- splitting

Conjunctive partial deduction

Input logic program P and a query Q_0

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



(instance of Q_0)

The set is kept finite using

- generalization
- splitting

This work

Original motivation:

- paralelizing 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:

- paralelizing 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:

- paralelizing 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:

- paralelizing 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**

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 $p/n : in \mapsto out$
- e.g., $\mathtt{append}/3 : \{1, 2\} \mapsto \{1, 2, 3\}$

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

Left-termination analysis

- determines if p/n terminates for call pattern *in* with Prolog's leftmost selection strategy
- e.g., $\mathtt{append}/3$ left-terminates for call pattern $\{1\}$
- e.g., $\mathtt{append}/3$ 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 $p/n : in \mapsto out$
- e.g., $\mathtt{append}/3 : \{1,2\} \mapsto \{1,2,3\}$

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

Left-termination analysis

- determines if p/n terminates for call pattern *in* with Prolog's leftmost selection strategy
- e.g., $\mathtt{append}/3$ left-terminates for call pattern $\{1\}$
- e.g., $\mathtt{append}/3$ 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

$$p(t_1, \ldots, t_n) \leftarrow body$$

  such that *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

$$p(t_1, \ldots, t_n) \leftarrow body$$

  such that *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

$$p(t_1, \ldots, t_n) \leftarrow body$$

such that *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

$$p(t_1, \ldots, t_n) \leftarrow body$$

  such that *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

$$p(t_1, \ldots, t_n) \leftarrow body$$

  such that *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

$$p(t_1, \ldots, t_n) \leftarrow body$$

  such that *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)

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

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

Example (not strongly regular)

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

- 2 SCCs: $\{\mathtt{flipflip}/2\}$ and $\{\mathtt{flip}/2\}$
- the second clause of $\mathtt{flip}/2$ violates the strongly regular condition

Example (strongly regular)

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

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

Example (not strongly regular)

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

- 2 SCCs: $\{\mathtt{flipflip}/2\}$ and $\{\mathtt{flip}/2\}$
- the second clause of $\mathtt{flip}/2$ 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

$$\text{(restart)}\quad \frac{\nexists qs' \in gs.\ qs_i \trianglerighteq qs',\ \ i \in \{1,\ldots,n\}}{\langle\langle\{qs_1,\ldots,qs_n\},\ gs\rangle\rangle \to \langle qs_i, [\,], \{qs_i\} \cup gs\rangle}$$

$$\text{(stop)}\quad \frac{\exists qs' \in gs.\ qs_i \trianglerighteq qs',\ \ i \in \{1,\ldots,n\}}{\langle\langle\{qs_1,\ldots,qs_n\},\ \ gs\rangle\rangle \to_{qs_i} \langle\langle\ \rangle\rangle}$$

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

$$\text{(restart)}\quad \frac{\nexists qs' \in gs.\ qs_i \trianglerighteq qs',\ \ i \in \{1, \ldots, n\}}{\langle\langle\{qs_1, \ldots, qs_n\},\ gs\rangle\rangle \to \langle qs_i, [\,], \{qs_i\} \cup gs\rangle}$$

$$\text{(stop)}\quad \frac{\exists qs' \in gs.\ qs_i \trianglerighteq qs',\ \ i \in \{1, \ldots, n\}}{\langle\langle\{qs_1, \ldots, qs_n\},\ \ gs\rangle\rangle \to_{qs_i} \langle\langle\ \rangle\rangle}$$

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

$$\text{(restart)}\quad \frac{\not\exists qs' \in gs.\ qs_i \trianglerighteq qs',\ \ i \in \{1,\ldots,n\}}{\langle\langle\{qs_1,\ldots,qs_n\},\ gs\rangle\rangle \to \langle qs_i, [\,], \{qs_i\} \cup gs\rangle}$$

$$\text{(stop)}\quad \frac{\exists qs' \in gs.\ qs_i \trianglerighteq qs',\ \ i \in \{1,\ldots,n\}}{\langle\langle\{qs_1,\ldots,qs_n\},\ \ gs\rangle\rangle \to_{qs_i} \langle\langle\ \rangle\rangle}$$

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

$$\text{(restart)} \quad \frac{\nexists qs' \in gs.\ qs_i \trianglerighteq qs',\ \ i \in \{1, \ldots, n\}}{\langle\langle \{qs_1, \ldots, qs_n\},\ gs\rangle\rangle \to \langle qs_i, [\,], \{qs_i\} \cup gs\rangle}$$

$$\text{(stop)} \quad \frac{\exists qs' \in gs.\ qs_i \trianglerighteq qs',\ \ i \in \{1, \ldots, n\}}{\langle\langle \{qs_1, \ldots, qs_n\},\ \ gs\rangle\rangle \to_{qs_i} \langle\langle\ \rangle\rangle}$$

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

$$(\text{restart}) \quad \frac{\not\exists qs' \in gs.\ qs_i \trianglerighteq qs',\ \ i \in \{1, \ldots, n\}}{\langle\langle \{qs_1, \ldots, qs_n\},\ gs \rangle\rangle \rightarrow \langle qs_i, [\,], \{qs_i\} \cup gs \rangle}$$

$$(\text{stop}) \quad \frac{\exists qs' \in gs.\ qs_i \trianglerighteq qs',\ \ i \in \{1, \ldots, n\}}{\langle\langle \{qs_1, \ldots, qs_n\},\ \ gs \rangle\rangle \rightarrow_{qs_i} \langle\langle\ \rangle\rangle}$$

Partial evaluation: local level

Local states:

$$\langle qs, ls, gs \rangle$$

where

- *qs* is a query (with call paterns)
- *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 paterns)
- *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 paterns)
- *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 paterns)
- 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 $\mathrm{p(a), q(X)}$ and the program

$$\begin{array}{l}\mathrm{p(b).}\\ \mathrm{q(X) :- q(X).}\end{array}$$

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

In our context, $\mathrm{q(X)}$ 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 $\mathtt{p(a), q(X)}$ and the program

$$\begin{array}{l}\mathtt{p(b).}\\ \mathtt{q(X) :- q(X).}\end{array}$$

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

In our context, $\mathtt{q(X)}$ 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 $\mathtt{p(a), q(X)}$ and the program

$$\begin{array}{l}\mathtt{p(b).}\\ \mathtt{q(X) :- q(X).}\end{array}$$

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

In our context, $\mathtt{q(X)}$ 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

$$qs = \mathrm{append}(\mathrm{X}, \mathrm{Y}, \mathrm{L}_1), \mathrm{append}(\mathrm{X}, \mathrm{Z}, \mathrm{L}_2), \mathrm{append}(\mathrm{L}_1, \mathrm{L}_2, \mathrm{R})$$

the independent splitting of qs returns

$$\begin{array}{rcl} qs_1 & = & \mathrm{append}(\mathrm{X}, \mathrm{Y}, \mathrm{L}_1) \\ qs_2 & = & \mathrm{append}(\mathrm{X}, \mathrm{Z}, \mathrm{L}_2) \\ qs_3 & = & \mathrm{append}(\mathrm{L}_1, \mathrm{L}_2, \mathrm{R}) \end{array}$$

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

$$qs = \mathtt{append}(\mathtt{X}, \mathtt{Y}, \mathtt{L}_1), \mathtt{append}(\mathtt{X}, \mathtt{Z}, \mathtt{L}_2), \mathtt{append}(\mathtt{L}_1, \mathtt{L}_2, \mathtt{R})$$

the independent splitting of qs returns

$$\begin{array}{rcl} qs_1 & = & \mathtt{append}(\mathtt{X}, \mathtt{Y}, \mathtt{L}_1) \\ qs_2 & = & \mathtt{append}(\mathtt{X}, \mathtt{Z}, \mathtt{L}_2) \\ qs_3 & = & \mathtt{append}(\mathtt{L}_1, \mathtt{L}_2, \mathtt{R}) \end{array}$$

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

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

is

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

since $\mathtt{flip}/2$ 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

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

is

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

since $\texttt{flip}/2$ is non-regular

Partial evaluation: local level

(variant) $$\frac{\exists qs' \in ls.\ qs \approx qs'}{\langle qs, ls, gs\rangle \stackrel{v}{\Rightarrow} \langle \diamond, ls, gs\rangle}$$

(independent splitting) $$\frac{\text{i-split}(qs) = \langle qs_1, qs_2, qs_3\rangle}{\langle qs, ls, gs\rangle \stackrel{i}{\Rightarrow} \langle\langle\{qs_1, qs_2, qs_3\},\ gs\rangle\rangle}$$

(unfold) $$\frac{\text{unfold}(qs) = qs'}{\langle qs, ls, gs\rangle \stackrel{u}{\Rightarrow}_\sigma \langle qs', \{qs\} \cup ls, gs\rangle}$$

(regular splitting) $$\frac{\text{r-split}(qs) = \langle qs_1, \ldots, qs_n\rangle}{\langle qs, ls, gs\rangle \stackrel{r}{\Rightarrow} \langle\langle\{qs_1, \ldots, qs_n\},\ gs\rangle\rangle}$$

Partial evaluation: local level

(variant) $$\frac{\exists qs' \in ls.\ qs \approx qs'}{\langle qs, ls, gs\rangle \ \overset{v}{\Rightarrow}\ \langle \diamond, ls, gs\rangle}$$

(independent splitting) $$\frac{\text{i-split}(qs) = \langle qs_1, qs_2, qs_3\rangle}{\langle qs, ls, gs\rangle \overset{i}{\Rightarrow} \langle\langle\{qs_1, qs_2, qs_3\},\ gs\rangle\rangle}$$

(unfold) $$\frac{\text{unfold}(qs) = qs'}{\langle qs, ls, gs\rangle \ \overset{u}{\Rightarrow}_\sigma\ \langle qs', \{qs\} \cup ls, gs\rangle}$$

(regular splitting) $$\frac{\text{r-split}(qs) = \langle qs_1, \ldots, qs_n\rangle}{\langle qs, ls, gs\rangle \overset{r}{\Rightarrow} \langle\langle\{qs_1, \ldots, qs_n\},\ gs\rangle\rangle}$$

Partial evaluation: local level

$$\text{(variant)}\quad \frac{\exists qs' \in ls.\ qs \approx qs'}{\langle qs, ls, gs\rangle \stackrel{v}{\Rightarrow} \langle \diamond, ls, gs\rangle}$$

$$\text{(independent splitting)}\quad \frac{\text{i-split}(qs) = \langle qs_1, qs_2, qs_3\rangle}{\langle qs, ls, gs\rangle \stackrel{i}{\Rightarrow} \langle\langle\{qs_1, qs_2, qs_3\},\ gs\rangle\rangle}$$

$$\text{(unfold)}\quad \frac{\text{unfold}(qs) = qs'}{\langle qs, ls, gs\rangle \stackrel{u}{\Rightarrow}_\sigma \langle qs', \{qs\} \cup ls, gs\rangle}$$

$$\text{(regular splitting)}\quad \frac{\text{r-split}(qs) = \langle qs_1, \ldots, qs_n\rangle}{\langle qs, ls, gs\rangle \stackrel{r}{\Rightarrow} \langle\langle\{qs_1, \ldots, qs_n\},\ gs\rangle\rangle}$$

Partial evaluation: local level

(variant) $$\frac{\exists qs' \in ls.\ qs \approx qs'}{\langle qs, ls, gs\rangle \ \stackrel{v}{\Rightarrow}\ \langle \diamond, ls, gs\rangle}$$

(independent splitting) $$\frac{\text{i-split}(qs) = \langle qs_1, qs_2, qs_3\rangle}{\langle qs, ls, gs\rangle \stackrel{i}{\Rightarrow} \langle\langle\{qs_1, qs_2, qs_3\},\ \ gs\rangle\rangle}$$

(unfold) $$\frac{\text{unfold}(qs) = qs'}{\langle qs, ls, gs\rangle \ \stackrel{u}{\Rightarrow}_\sigma \langle qs', \{qs\} \cup ls, gs\rangle}$$

(regular splitting) $$\frac{\text{r-split}(qs) = \langle qs_1, \ldots, qs_n\rangle}{\langle qs, ls, gs\rangle \stackrel{r}{\Rightarrow} \langle\langle\{qs_1, \ldots, qs_n\},\ \ gs\rangle\rangle}$$

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

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

- 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 $ren(qs_i) \leftarrow qs_i$

Post-processing

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

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

- 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 $ren(qs_i) \leftarrow qs_i$

Post-processing

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

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

- 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 $ren(qs_i) \leftarrow qs_i$

Experimental results

A prototype has been implemented ($\approx$ 1000 lines, SWI Prolog)

(left-termination and SRP analysis still missing)

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

benchmark	*advisor*	*applast*	*depth*	*doubleapp*	*ex_depth*	*flip*	*matchapp*	*regexp.r1*
original	4	58	24	50	24	34	374	73
residual	0	29	1	34	15	47	23	10

benchmark	*regexp.r2*	*regexp.r3*	*relative*	*rev_acc_type*	*rotateprune*	*transpose*
original	28	41	96	35	32	58
residual	8	12	3	34	45	0

Experimental results

A prototype has been implemented ($\approx$ 1000 lines, SWI Prolog)

(left-termination and SRP analysis still missing)

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

benchmark	*advisor*	*applast*	*depth*	*doubleapp*	*ex_depth*	*flip*	*matchapp*	*regexp.r1*
original	4	58	24	50	24	34	374	73
residual	0	29	1	34	15	47	23	10

benchmark	*regexp.r2*	*regexp.r3*	*relative*	*rev_acc_type*	*rotateprune*	*transpose*
original	28	41	96	35	32	58
residual	8	12	3	34	45	0

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 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 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 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)