Towards scalable partial evaluation of declarative programs

Germán Vidal
Technical University of Valencia, Spain

{ *Joint work with Michael Leuschel, U. Düsseldorf, Germany* }

LOPSTR–PPDP 2009

September 9, 2009
Coimbra, Portugal

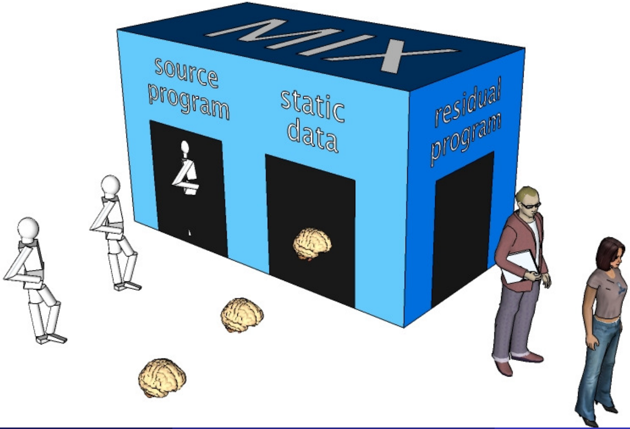
Outline

1. introduction
   - partial evaluation
   - applications
   - internals
2. termination analysis
   - introduction
   - construction of size-change graphs
   - computation of composition closure
   - identification of program loops
3. a fully automatic BTA
   - local termination
   - ensuring global termination
4. Concluding remarks

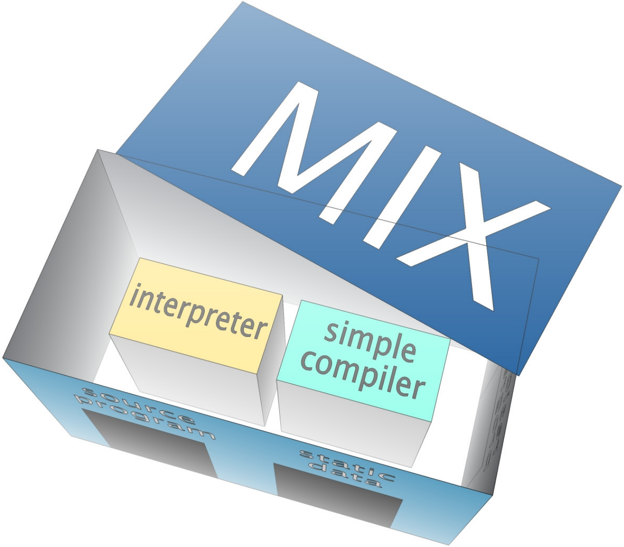
What is partial evaluation?

Definition

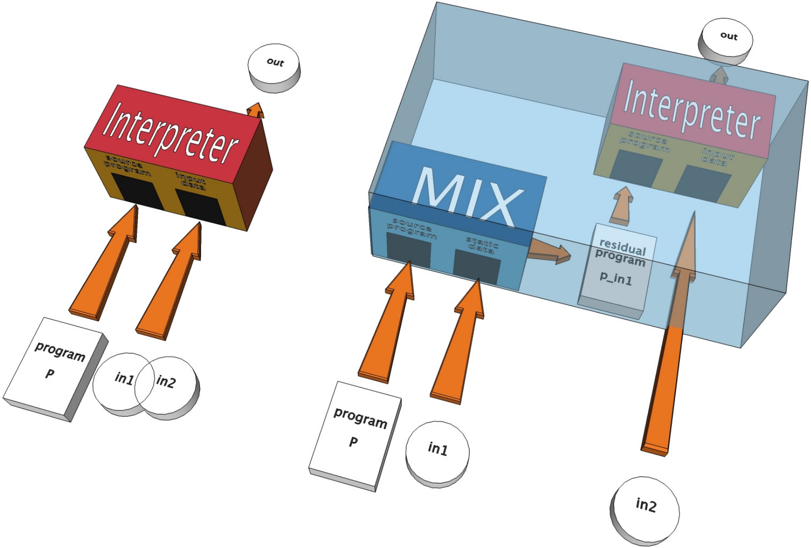
A partial evaluator takes a program and **part** *of its input data (static data) and returns a* **new***, specialized program*



What is partial evaluation?

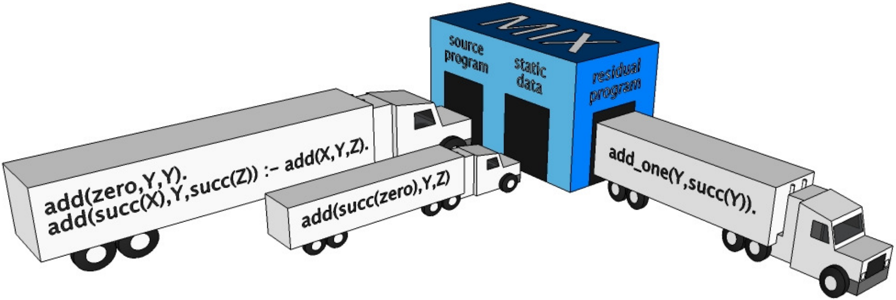


Correctness of partial evaluation

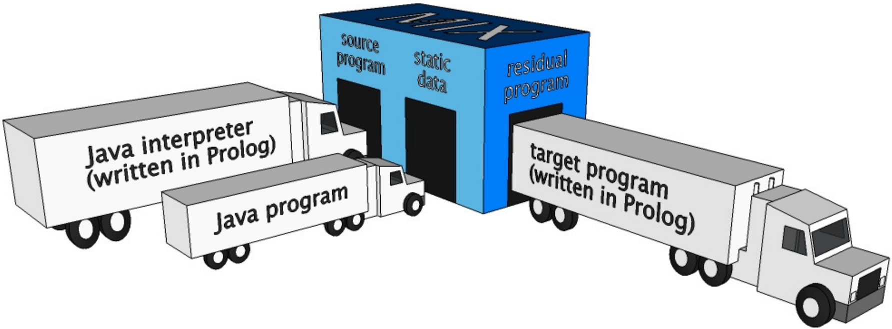


Some applications of partial evaluation

Program specialization

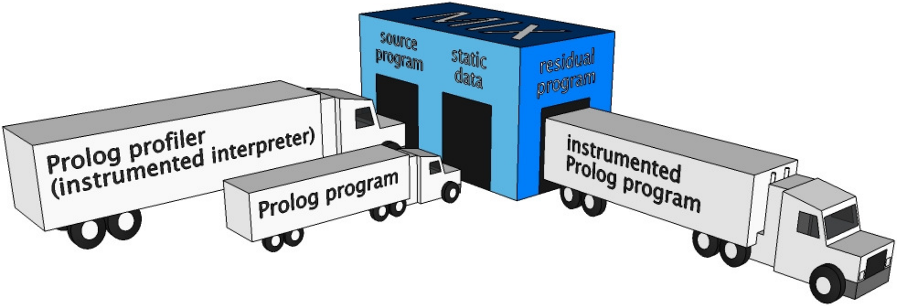


(De)compilation



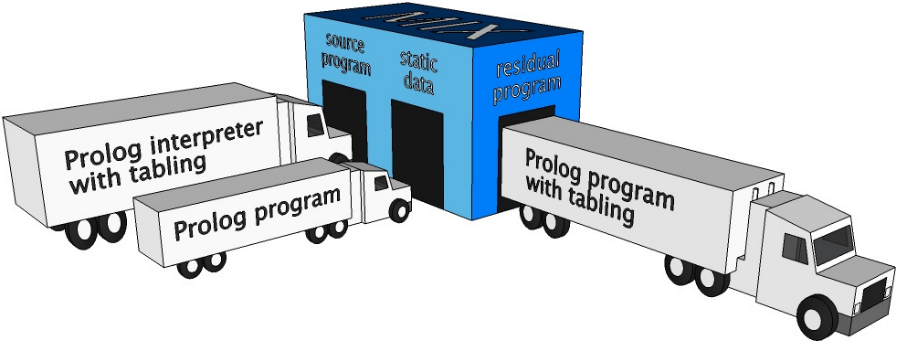
(e.g., interpretive decompilation [Albert et al., PADL 2007])

Program instrumentation



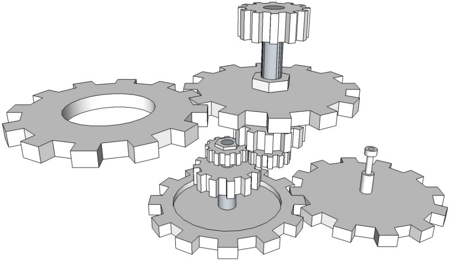
(e.g., semantics directed program execution monitoring [Kishon & Hudak, JFP 1995])

Optimized evaluation



(e.g., driving in the jungle [Secher, PADO 2001])

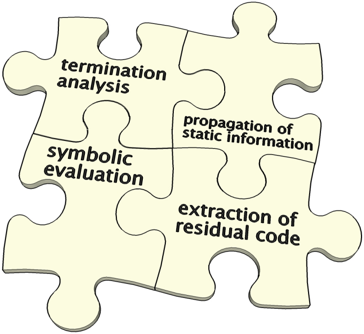
Internals of a partial evaluator



Elements of partial evaluation

to ensure the finiteness of partial evaluation

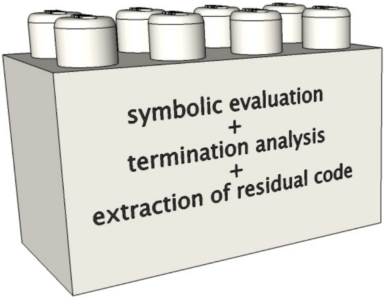
to evaluate expressions with missing information (variables)

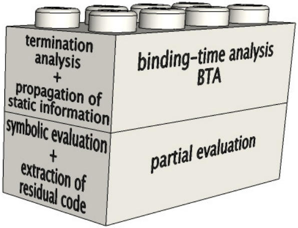


to propagate known information through the entire program

to extract the residual program from partial computations

Online vs offline





- more accurate
- less efficient

- less accurate
- more efficient

Binding-time analysis & partial evaluation

The BTA annotates the source program:

- every call is annotated with **unfold**/**memo**
- every parameter is annotated with **static**/**dynamic**

partial evaluation phase

1. take a call and unfold it as much as possible (following the unfold/memo annotations) — local control
2. for every call in the leaves — global control
   - generalize arguments marked as dynamic
   - add it to the set of calls to be partially evaluated
3. go to step (1)

Safeness of BTA annotations

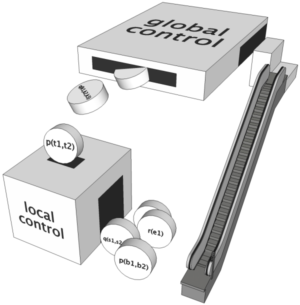
The annotations inferred by the BTA are safe if

- **local termination** is ensured

  (i.e., no call is infinitely unfolded)
- **global termination** is ensured

  (i.e., no infinitely many calls are partially evaluated)

- all parameters marked as **static** are actually known at partial evaluation time



Example

source program

$$
\begin{array}{l}
incList([\,], I, [\,]).\\
incList([H|T], I, [HI|TI]) \leftarrow add(I, H, HI),\\
\qquad\qquad\qquad\qquad\qquad\quad incList(T, I, TI).\\
add(zero, Y, Y).\\
add(succ(X), Y, succ(Z)) \leftarrow add(X, Y, Z).
\end{array}
$$

static data

$incList(L, succ(zero), L')$

output of the BTA

$$
\begin{array}{l}
incList(dynamic, static, dynamic)\\
add(static, dynamic, dynamic)
\end{array}
$$

partial evaluation

$incList(L, succ(zero), L')$

$\{L/[\,],\ L'/[\,]\}$ → $\square$

$\{L/[H|T],\ L'/[HI|TI]\}$ → $add(succ(zero), H, HI),\ incList(T, succ(zero), TI)$

$\downarrow \{HI/succ(HI')\}$

$add(zero, H, HI'),\ incList(T, succ(zero), TI)$

$\downarrow \{HI'/H\}$

$incList(T, succ(zero), TI)$

residual program

$incList_{one}([\,], [\,]).\qquad incList_{one}([H|T], [succ(H)|TI]) \leftarrow incList_{one}(T, TI).$

Scheme of a simple BTA

1. initially all calls marked as **unfold**
2. propagation of static information
3. termination analysis (for a particular selection or evaluation strategy)
4. if termination of a call cannot be ensured given the inferred static information, then change annotation to **memo** and go to step (2)

program

$p(X,Y) \quad \leftarrow \quad q(X,Z),\ r(Z,Y).$
$\ldots$

initialization

$p(static, dynamic)$
$p \mapsto unfold,\ q \mapsto unfold$

propagation of static info

$p(X,Z) \quad \leftarrow \quad q(X,Y),\ r(Y,Z).$
$\ldots$

$p(X,Z) \quad \leftarrow \quad q(X,Y),\ r(Y,Z).$
$\ldots$

termination analysis

$p \mapsto unfold,\ r \mapsto unfold$
$q \mapsto memo$

$p \mapsto unfold$
$q \mapsto memo,\ r \mapsto memo$

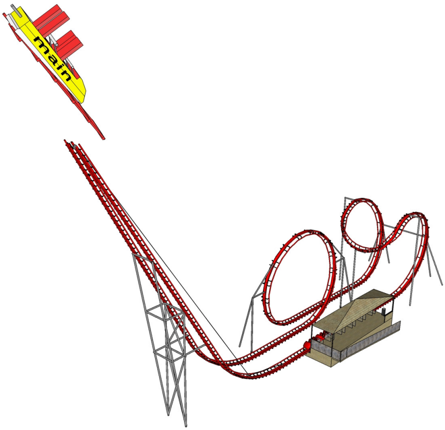
Our approach

1. keep the termination analysis and the propagation of static information independent
2. use a **strong** termination analysis
   [independent of selection or evaluation strategy]
3. implement efficient algorithms (e.g., hashing)
4. improve accuracy as much as possible
   - order of evaluation partially known
   - online annotations
   - user annotations

Remainder of the talk... designing a fully atomatic BTA

- termination analysis
  - size-change graphs
  - composition closure
- program annotation

termination analysis



Termination and quasi-termination

Terminating computation

- finite number of states
- E.g., $nat(s(s(0))) \rightarrow nat(s(0)) \rightarrow nat(0) \rightarrow \Box$

with

$$\begin{array}{l} nat(0). \\ nat(s(X)) \quad \leftarrow \quad nat(X). \end{array}$$

Quasi-terminating computation

- finite number of **different** states
- E.g., $nat(X) \rightarrow_{\{X/s(X')\}} nat(X') \rightarrow_{\{X'/s(X'')\}} nat(X'') \rightarrow \ldots$

Termination analysis & program annotations

Termination

- if a call is terminating
  annotate it with unfold, otherwise with memo

Quasi-termination

- if a call is quasi-terminating
  annotate all its arguments with static
  otherwise annotate the (possibly) increasing arguments with memo
  (will be generalized in the global level)
- E.g., $reverse(L, L') \rightarrow rev(L, [\,], L') \rightarrow_{\{L/[H|T]\}} rev(T, [H], L') \rightarrow \ldots$

  with

  $reverse(L, L') \leftarrow rev(L, [\,], L').$
  $rev([\,], A, A).$
  $rev([H|T], A, L). \leftarrow rev(T, [H|A], L).$

  therefore

  $rev(static, dynamic, static)$

Size-change analysis of logic programs

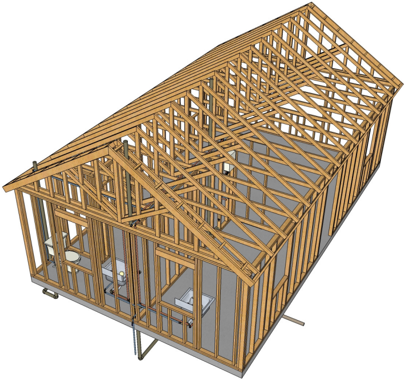
Main features

- adapted from [Lee, Jones, Ben-Amram, POPL 2001], [Lindenstrauss, Sagiv, ICLP 1997], etc
- consider both strong termination and quasi-termination
- appropriate for BTA and partial evaluation

size-change analysis

1. construction of size-change graphs
2. computation of composition closure

Construction of size-change graphs



Construction of size-change graphs

Size-change graphs are used to trace size changes of predicate arguments from one call to another

We construct a size-change graph for every call in the program

$$
\begin{array}{ll}
\mathbf{incList}([\,], \_, [\,]). & \mathbf{add}(\mathbf{0}, \mathbf{Y}, \mathbf{Y}). \\
\mathbf{incList}([\mathbf{X}|\mathbf{R}], \mathbf{I}, \mathbf{L}) \leftarrow \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, \mathbf{L}). & \mathbf{add}(\mathbf{s}(\mathbf{X}), \mathbf{Y}, \mathbf{s}(\mathbf{Z})) \leftarrow \mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}). \\
\mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, [\mathbf{XI}|\mathbf{RI}]) \leftarrow \mathbf{add}(\mathbf{I}, \mathbf{X}, \mathbf{XI}), & \\
\qquad \mathbf{incList}(\mathbf{R}, \mathbf{I}, \mathbf{RI}). &
\end{array}
$$

Here, we construct four size-change graphs:

$$\mathbf{incList} \mapsto \mathbf{iList}, \quad \mathbf{iList} \mapsto \mathbf{add}, \quad \mathbf{iList} \mapsto \mathbf{incList}, \quad \mathbf{add} \mapsto \mathbf{add}$$

Construction of size-change graphs

Size-change graphs are used to trace size changes of predicate arguments from one call to another

We construct a size-change graph for every call in the program

$$
\begin{array}{lll}
\mathbf{incList}([\,], \_, [\,]). & & \mathbf{add}(\mathbf{0}, \mathbf{Y}, \mathbf{Y}). \\
\mathbf{incList}([\mathbf{X}|\mathbf{R}], \mathbf{I}, \mathbf{L}) & \leftarrow \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, \mathbf{L}). & \mathbf{add}(\mathbf{s}(\mathbf{X}), \mathbf{Y}, \mathbf{s}(\mathbf{Z})) \leftarrow \mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}). \\
\mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, [\mathbf{XI}|\mathbf{RI}]) & \leftarrow \mathbf{add}(\mathbf{I}, \mathbf{X}, \mathbf{XI}), & \\
 & \quad\ \mathbf{incList}(\mathbf{R}, \mathbf{I}, \mathbf{RI}). &
\end{array}
$$

Here, we construct four size-change graphs:

$$\mathbf{incList} \mapsto \mathbf{iList}, \quad \mathbf{iList} \mapsto \mathbf{add}, \quad \mathbf{iList} \mapsto \mathbf{incList}, \quad \mathbf{add} \mapsto \mathbf{add}$$

Construction of size-change graphs

Size-change graphs are used to trace size changes of predicate arguments from one call to another

We construct a size-change graph for every call in the program

$$
\begin{array}{lll}
\mathbf{incList}([\,], \_, [\,]). & & \mathbf{add}(\mathbf{0}, \mathbf{Y}, \mathbf{Y}). \\
\mathbf{incList}([\mathbf{X}|\mathbf{R}], \mathbf{I}, \mathbf{L}) & \leftarrow \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, \mathbf{L}). & \mathbf{add}(\mathbf{s}(\mathbf{X}), \mathbf{Y}, \mathbf{s}(\mathbf{Z})) \leftarrow \mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}). \\
\mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, [\mathbf{XI}|\mathbf{RI}]) & \leftarrow \mathbf{add}(\mathbf{I}, \mathbf{X}, \mathbf{XI}), & \\
 & \quad\ \mathbf{incList}(\mathbf{R}, \mathbf{I}, \mathbf{RI}). &
\end{array}
$$

Here, we construct four size-change graphs:

$$\mathbf{incList} \mapsto \mathbf{iList}, \quad \mathbf{iList} \mapsto \mathbf{add}, \quad \mathbf{iList} \mapsto \mathbf{incList}, \quad \mathbf{add} \mapsto \mathbf{add}$$

Construction of size-change graphs

Size-change graphs are used to trace size changes of predicate arguments from one call to another

We construct a size-change graph for every call in the program

$$\begin{array}{lll} \mathbf{incList}([\,], \_, [\,]). & & \mathbf{add}(0, \mathbf{Y}, \mathbf{Y}). \\ \mathbf{incList}([\mathbf{X}|\mathbf{R}], \mathbf{I}, \mathbf{L}) & \leftarrow \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, \mathbf{L}). & \mathbf{add}(s(\mathbf{X}), \mathbf{Y}, s(\mathbf{Z})) \leftarrow \mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}). \\ \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, [\mathbf{XI}|\mathbf{RI}]) & \leftarrow \mathbf{add}(\mathbf{I}, \mathbf{X}, \mathbf{XI}), & \\ & \quad \mathbf{incList}(\mathbf{R}, \mathbf{I}, \mathbf{RI}). & \end{array}$$

Here, we construct four size-change graphs:

$$\mathbf{incList} \mapsto \mathbf{iList}, \quad \mathbf{iList} \mapsto \mathbf{add}, \quad \mathbf{iList} \mapsto \mathbf{incList}, \quad \mathbf{add} \mapsto \mathbf{add}$$

Construction of size-change graphs

Size-change graphs are used to trace size changes of predicate arguments from one call to another

We construct a size-change graph for every call in the program

$$
\begin{array}{lll}
\mathbf{incList}([\,], \_, [\,]). & & \mathbf{add}(\mathbf{0}, \mathbf{Y}, \mathbf{Y}). \\
\mathbf{incList}([\mathbf{X}|\mathbf{R}], \mathbf{I}, \mathbf{L}) & \leftarrow \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, \mathbf{L}). & \mathbf{add}(\mathbf{s}(\mathbf{X}), \mathbf{Y}, \mathbf{s}(\mathbf{Z})) \leftarrow \mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}). \\
\mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, [\mathbf{XI}|\mathbf{RI}]) & \leftarrow \mathbf{add}(\mathbf{I}, \mathbf{X}, \mathbf{XI}), & \\
 & \quad\; \mathbf{incList}(\mathbf{R}, \mathbf{I}, \mathbf{RI}). &
\end{array}
$$

Here, we construct four size-change graphs:

$$\mathbf{incList} \mapsto \mathbf{iList}, \quad \mathbf{iList} \mapsto \mathbf{add}, \quad \mathbf{iList} \mapsto \mathbf{incList}, \quad \mathbf{add} \mapsto \mathbf{add}$$

Size-change graphs are parameterized by a **reduction pair** $(\gtrsim, \succ)$ [Thiemann, Giesl, AAECC 2005]:

1. $\gtrsim$ is a quasi-order [reflexive & transitive]
2. $\succ$ is a well-founded order [irreflexive & transitive]
3. $\gtrsim$ and $\succ$ are closed under substitutions [$s \gtrsim t \;\Rightarrow\; \sigma(s) \gtrsim \sigma(t)$]
4. they are compatible

   (i.e., $\gtrsim \circ \succ \subseteq \succ$ and $\succ \circ \gtrsim \subseteq \succ$ but $\gtrsim \subseteq \succ$ is not necessary)

which can be induced from a **symbolic norm** $||\;||$:

$$s \succ t \;\Leftrightarrow\; ||s|| > ||t|| \quad \text{and} \quad s \gtrsim t \;\Leftrightarrow\; ||s|| \geqslant ||t||$$

Size-change graphs are parameterized by a **reduction pair** $(\succsim, \succ)$ [Thiemann, Giesl, AAECC 2005]:

1. $\succsim$ is a quasi-order [reflexive & transitive]
2. $\succ$ is a well-founded order [irreflexive & transitive]
3. $\succsim$ and $\succ$ are closed under substitutions [$s \succsim t \;\Rightarrow\; \sigma(s) \succsim \sigma(t)$]
4. they are compatible

   (i.e., $\succsim \circ \succ \subseteq \succ$ and $\succ \circ \succsim \subseteq \succ$ but $\succsim \subseteq \succ$ is not necessary)

which can be induced from a **symbolic norm** $|| \; ||$:

$$s \succ t \;\Leftrightarrow\; ||s|| > ||t|| \quad \text{and} \quad s \succsim t \;\Leftrightarrow\; ||s|| \geqslant ||t||$$

Symbolic norms

symbolic term-size norm

$$||t||_{ts} = \begin{cases} n + \sum_{i=0}^{n} ||t_i||_{ts} & \text{if } t = f(t_1,\ldots,t_n),\ n \geqslant 0 \\ t & \text{if } t \text{ is a variable} \end{cases}$$

E.g., $||\mathbf{f(a,b)}||_{ts} = \mathbf{2}$, but $||\mathbf{f(X,Y)}||_{ts} = \mathbf{2+X+Y}$

symbolic list-length norm

$$||t||_{ll} = \begin{cases} 1 + ||Xs||_{ll} & \text{if } t = [X|Xs] \\ t & \text{if } t \text{ is a variable} \\ 0 & \text{otherwise} \end{cases}$$

E.g., $||\mathbf{[1,X]}||_{ll} = \mathbf{2}$, but $||\mathbf{[1|X]}||_{ll} = \mathbf{1+X}$

Symbolic norms

symbolic term-size norm

$$||t||_{ts} = \begin{cases} n + \sum_{i=0}^{n} ||t_i||_{ts} & \text{if } t = f(t_1, \ldots, t_n),\ n \geqslant 0 \\ t & \text{if } t \text{ is a variable} \end{cases}$$

E.g., $||\mathbf{f(a,b)}||_{ts} = \mathbf{2}$, but $||\mathbf{f(X,Y)}||_{ts} = \mathbf{2 + X + Y}$

symbolic list-length norm

$$||t||_{ll} = \begin{cases} 1 + ||Xs||_{ll} & \text{if } t = [X|Xs] \\ t & \text{if } t \text{ is a variable} \\ 0 & \text{otherwise} \end{cases}$$

E.g., $||\mathbf{[1,X]}||_{ll} = \mathbf{2}$, but $||\mathbf{[1|X]}||_{ll} = \mathbf{1 + X}$

Size-change graph (parametric w.r.t. $(\succsim, \succ)$)

We have a size-change graph for every pair

$$p(s_1, \ldots, s_n) \;/\; q(t_1, \ldots, t_m)$$

such that there is a program clause

$$p(s_1, \ldots, s_n) \leftarrow \ldots q(t_1, \ldots, t_m) \ldots$$

- **Nodes**
  - output nodes: $\{1_p, \ldots, n_p\}$
  - input nodes: $\{1_q, \ldots, m_q\}$
- **Edges**
  - if $s_i \succ t_j$ then $i_p \xrightarrow{\succ} j_q$
  - else if $s_i \succsim t_j$ then $i_p \xrightarrow{\succsim} j_q$ [otherwise there is no edge]

Example: construction of size-change graphs

$$\begin{array}{llll}
\mathbf{incList}([\,], _, [\,]). & & \mathbf{add}(\mathbf{0}, \mathbf{Y}, \mathbf{Y}). & \\
\mathbf{incList}([\mathbf{X}|\mathbf{R}], \mathbf{I}, \mathbf{L}) & \leftarrow \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, \mathbf{L}). & \mathbf{add}(\mathbf{s}(\mathbf{X}), \mathbf{Y}, \mathbf{s}(\mathbf{Z})) & \leftarrow \mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}). \\
\mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, [\mathbf{XI}|\mathbf{RI}]) & \leftarrow \mathbf{add}(\mathbf{I}, \mathbf{X}, \mathbf{XI}), & & \\
 & \quad\ \mathbf{incList}(\mathbf{R}, \mathbf{I}, \mathbf{RI}). & &
\end{array}$$

Example: construction of size-change graphs

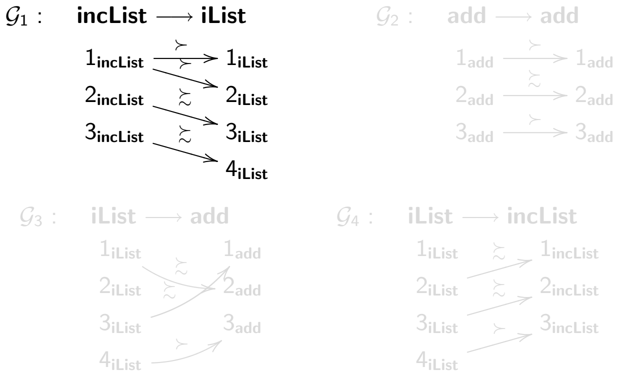
incList([ ], _, [ ]).
incList([X|R], I, L) ← iList(X, R, I, L).
iList(X, R, I, [XI|RI]) ← add(I, X, XI),
incList(R, I, RI).

add(0, Y, Y).
add(s(X), Y, s(Z)) ← add(X, Y, Z).

Example: construction of size-change graphs

incList([ ], _, [ ]).
incList([X|R], I, L) ← iList(X, R, I, L).
iList(X, R, I, [XI|RI]) ← add(I, X, XI),
incList(R, I, RI).

add(0, Y, Y).
add(s(X), Y, s(Z)) ← add(X, Y, Z).



Example: construction of size-change graphs

$\mathsf{incList}([\,], \_, [\,]).$

$\mathsf{incList}([X|R], I, L) \leftarrow \mathsf{iList}(X, R, I, L).$

$\mathsf{iList}(X, R, I, [XI|RI]) \leftarrow \mathsf{add}(I, X, XI), \mathsf{incList}(R, I, RI).$

$\mathbf{add(0, Y, Y).}$

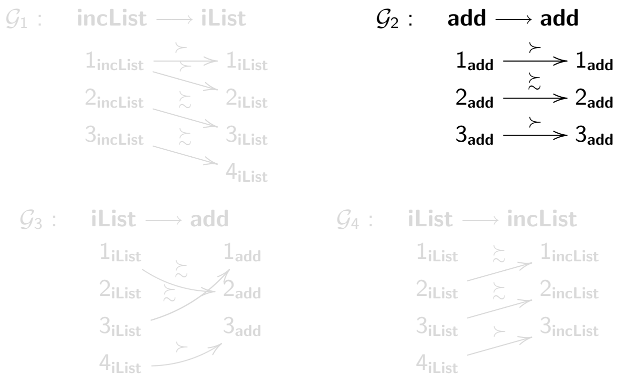
$\mathsf{add}(s(X), Y, s(Z)) \leftarrow \mathsf{add}(X, Y, Z).$

Example: construction of size-change graphs

incList([ ], _, [ ]).
incList([X|R], I, L) ← iList(X, R, I, L).

iList(X, R, I, [XI|RI]) ← add(I, X, XI),
incList(R, I, RI).

add(0, Y, Y).
add(s(X), Y, s(Z)) ← add(X, Y, Z).

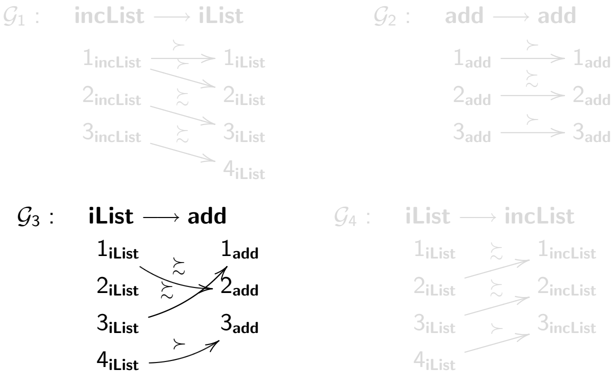


Example: construction of size-change graphs

incList([ ], _, [ ]).
incList([X|R], I, L) ← iList(X, R, I, L).

add(0, Y, Y).
add(s(X), Y, s(Z)) ← add(X, Y, Z).

iList(X, R, I, [XI|RI]) ← add(I, X, XI),
incList(R, I, RI).

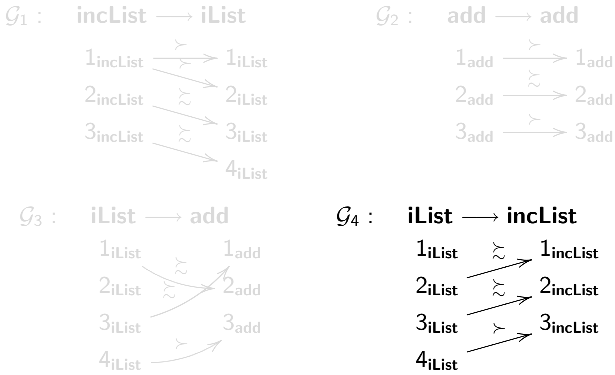


Example: construction of size-change graphs

$\mathbf{incList}([\,], \_, [\,]).$ $\quad$ $\mathbf{add}(0, \mathbf{Y}, \mathbf{Y}).$

$\mathbf{incList}([\mathbf{X}|\mathbf{R}], \mathbf{I}, \mathbf{L}) \leftarrow \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, \mathbf{L}).$ $\quad$ $\mathbf{add}(s(\mathbf{X}), \mathbf{Y}, s(\mathbf{Z})) \leftarrow \mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}).$

$\mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, [\mathbf{XI}|\mathbf{RI}]) \leftarrow \mathbf{add}(\mathbf{I}, \mathbf{X}, \mathbf{XI}),$
$\qquad \mathbf{incList}(\mathbf{R}, \mathbf{I}, \mathbf{RI}).$



Example: construction of size-change graphs

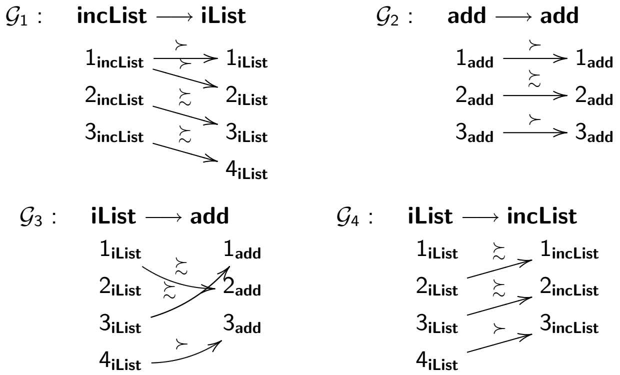
$\mathbf{incList}([\,], \_, [\,]).$

$\mathbf{incList}([\mathbf{X}|\mathbf{R}], \mathbf{I}, \mathbf{L}) \leftarrow \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, \mathbf{L}).$

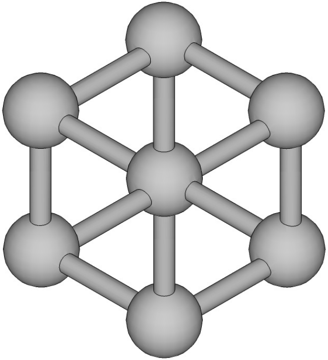
$\mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, [\mathbf{XI}|\mathbf{RI}]) \leftarrow \mathbf{add}(\mathbf{I}, \mathbf{X}, \mathbf{XI}), \mathbf{incList}(\mathbf{R}, \mathbf{I}, \mathbf{RI}).$

$\mathbf{add}(\mathbf{0}, \mathbf{Y}, \mathbf{Y}).$

$\mathbf{add}(\mathbf{s}(\mathbf{X}), \mathbf{Y}, \mathbf{s}(\mathbf{Z})) \leftarrow \mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}).$



Computation of composition closure



Composition of size-change graphs

If $G = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1)$ and
$H = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, w_r\}, E_2)$ are size-change graphs,

then their composition is

$$G \cdot H = (\{1_p, \ldots, n_p\}, \{1_r, \ldots, w_r\}, E)$$

where

1. $(i_p \longrightarrow k_r) \in E$ iff $(i_p \longrightarrow j_q) \in E_1$ and $(j_q \longrightarrow k_r) \in E_2$
2. if one of the edges is labeled with $\succ$ so is the new edge
3. otherwise, it is labeled with $\succsim$

Composition of size-change graphs

If $G = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1)$ and
$H = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, w_r\}, E_2)$ are size-change graphs,

then their composition is

$$G \cdot H = (\{1_p, \ldots, n_p\}, \{1_r, \ldots, w_r\}, E)$$

where

1. $(i_p \longrightarrow k_r) \in E$ iff $(i_p \longrightarrow j_q) \in E_1$ and $(j_q \longrightarrow k_r) \in E_2$
2. if one of the edges is labeled with $\succ$ so is the new edge
3. otherwise, it is labeled with $\succsim$

Composition of size-change graphs

If $G = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1)$ and
$H = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, w_r\}, E_2)$ are size-change graphs,

then their composition is

$$G \cdot H = (\{1_p, \ldots, n_p\}, \{1_r, \ldots, w_r\}, E)$$

where

1. $(i_p \longrightarrow k_r) \in E$ iff $(i_p \longrightarrow j_q) \in E_1$ and $(j_q \longrightarrow k_r) \in E_2$
2. if one of the edges is labeled with $\succ$ so is the new edge
3. otherwise, it is labeled with $\succsim$

Computing the composition closure

Naive procedure

1. initialize $\mathcal{M}$ with the size-change graphs of the program
2. repeat
   - for every pair of graphs $G, H \in \mathcal{M}$, add $G \cdot H$ to $\mathcal{M}$

   until no new graphs are added to $\mathcal{M}$

Drawback

- computationally expensive (exponential time)

Computing the composition closure

Naive procedure

1. initialize $\mathcal{M}$ with the size-change graphs of the program
2. repeat
   - for every pair of graphs $G, H \in \mathcal{M}$, add $G \cdot H$ to $\mathcal{M}$

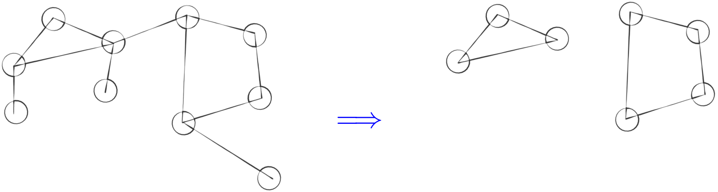
   until no new graphs are added to $\mathcal{M}$

Drawback

- computationally expensive (exponential time)

Ideas for improvement

Not all size-change graphs are needed (only those in a loop)

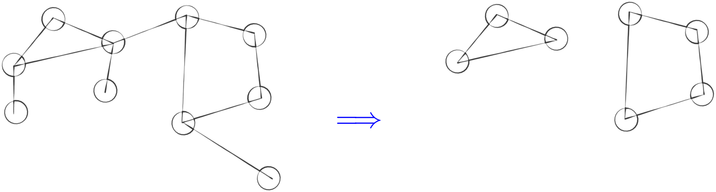


If a graph is "stronger" than another graph, it can be safely deleted

Computing the the compositions starting from a **single** predicate for each loop suffices in our context (BTA)

Ideas for improvement

Not all size-change graphs are needed (only those in a loop)

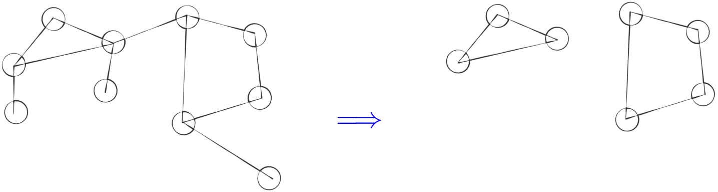


If a graph is "stronger" than another graph, it can be safely deleted

Computing the the compositions starting from a **single** predicate for each loop suffices in our context (BTA)

Ideas for improvement

Not all size-change graphs are needed (only those in a loop)



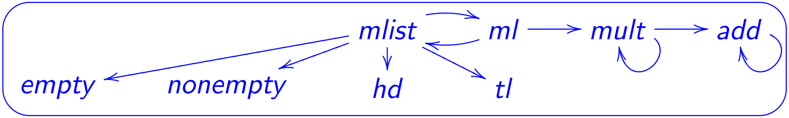
If a graph is "stronger" than another graph, it can be safely deleted

Computing the the compositions starting from a **single** predicate for each loop suffices in our context (BTA)

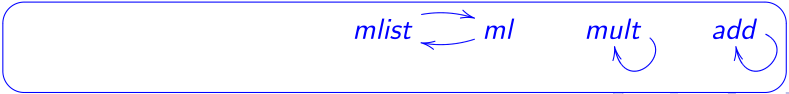
Identifying the program loops

(c_1) $mlist(L, I, [\,]) \leftarrow empty(L).$
(c_2) $mlist(L, I, LI) \leftarrow nonempty(L), hd(L, X), tl(L, R), ml(X, R, I, LI).$
(c_3) $ml(X, R, I, [XI|RI]) \leftarrow mult(X, I, XI),\ mlist(R, I, RI).$
(c_4) $mult(0, Y, 0).$ (c_5) $mult(s(X), Y, Z) \leftarrow mult(X, Y, Z1),\ add(Z1, Y, Z).$
(c_6) $add(X, 0, X).$ (c_7) $add(X, s(Y), s(Z)) \leftarrow add(X, Y, Z).$
(c_8) $hd([X|\_], X).$ (c_9) $empty([\,]).$
(c_{10}) $tl([\_|R], R).$ (c_{11}) $nonempty([\_|\_]).$

1. construct the call graph of the program



2. compute its strongly connected components (SCC)
3. delete trivial SCCs (a single non-recursive node)
4. delete edges between SCCs



Improved algorithm for composition closure

hazlo mas informal!!! (di las cosas con palabras...)

1. **Input:** a program P and a cover set $S \in CS(P)$
2. **Initialisation:**
   $i := 0; \quad \mathcal{M}_i := i\_sc\_graphs(P, S); \quad SC := sc\_graphs(P)$
3. **repeat**
   - $\mathcal{M}_{add} := \varnothing; \mathcal{M}_{del} := \varnothing$
   - for all $\mathcal{G}_1 \in \mathcal{M}_i$ and $\mathcal{G}_2 \in SC$ such that $\mathcal{G}_1 \bullet \mathcal{G}_2$ is defined
     1. $\mathcal{G} := \mathcal{G}_1 \bullet \mathcal{G}_2$
     2. **if** $\nexists \mathcal{H} \in (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}$ such that $\mathcal{G} \sqsubseteq \mathcal{H}$ or $\mathcal{H} \sqsubseteq \mathcal{G}$
        then $\mathcal{M}_{add} := \mathcal{M}_{add} \cup \{\mathcal{G}\}$
     3. **if** $\exists \mathcal{H} \in (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}$ such that $\mathcal{G} \sqsubseteq \mathcal{H}$ **then** $\mathcal{M}_{add} :=$
        $\mathcal{M}_{add} \cup \{\mathcal{G}\}$ and $\mathcal{M}_{del} := \mathcal{M}_{del} \cup \{\mathcal{H} \in (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}) \mid \mathcal{G} \sqsubseteq \mathcal{H}\}$
   - $\mathcal{M}_{i+1} := (\mathcal{M}_i \cup \mathcal{M}_{add}) \setminus \mathcal{M}_{del}$
   - $i := i + 1$

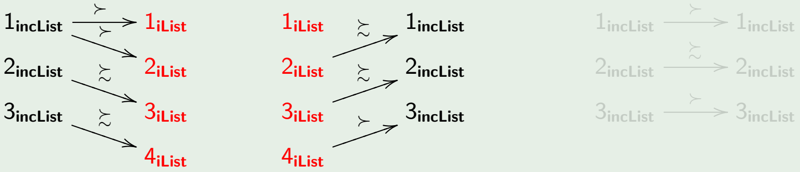
   until $\mathcal{M}_i = \mathcal{M}_{i+1}$

Transitive closure:

- compute all possible concatenations of graphs

Example

$\mathcal{G}_1$:**incList** $\longrightarrow$ **iList** $\bullet$ $\mathcal{G}_4$:**iList** $\longrightarrow$ **incList** $=$ $\mathcal{G}_{14}$: **incList** $\longrightarrow$ **incList**



Transitive closure:

- compute all possible concatenations of graphs

Example

Identification of program loops

maximal graph

A size-change graph G is maximal if

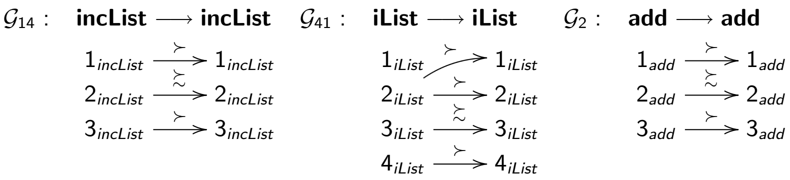
1. its input and output nodes are the same
2. it is idempotent, i.e., $G = G \cdot G$

maximal graph $\approx$ program loop

Example

$$\mathbf{incList}([\,], \_, [\,]).$$
$$\mathbf{incList}([\mathbf{X}|\mathbf{R}], \mathbf{I}, \mathbf{L}) \leftarrow \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, \mathbf{L}).$$
$$\mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, [\mathbf{XI}|\mathbf{RI}]) \leftarrow \mathbf{add}(\mathbf{I}, \mathbf{X}, \mathbf{XI}), \mathbf{incList}(\mathbf{R}, \mathbf{I}, \mathbf{RI}).$$

$$\mathbf{add}(\mathbf{0}, \mathbf{Y}, \mathbf{Y}).$$
$$\mathbf{add}(\mathbf{s}(\mathbf{X}), \mathbf{Y}, \mathbf{s}(\mathbf{Z})) \leftarrow \mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}).$$



Size-change terminating! (acc. to [Lee, Jones, Ben-Amram, POPL 2001])

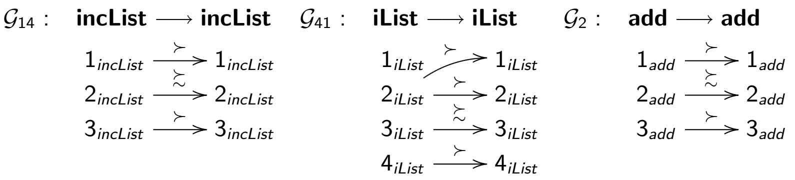
Too weak in logic programming...

$$\mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) \Rightarrow_{\{X \mapsto s(X')],\ Z \mapsto s(Z')\}} \mathbf{add}(\mathbf{X'}, \mathbf{Y}, \mathbf{Z'}) \Rightarrow_{\{X' \mapsto s(X''),\ Z' \mapsto s(Z'')\}} \mathbf{add}(\mathbf{X''}, \mathbf{Y}, \mathbf{Z''}) \Rightarrow \infty$$

Example

$$\begin{array}{ll}
\mathbf{incList}([\,], _, [\,]). & \mathbf{add}(\mathbf{0}, \mathbf{Y}, \mathbf{Y}). \\
\mathbf{incList}([\mathbf{X}|\mathbf{R}], \mathbf{I}, \mathbf{L}) \leftarrow \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, \mathbf{L}). & \mathbf{add}(\mathbf{s}(\mathbf{X}), \mathbf{Y}, \mathbf{s}(\mathbf{Z})) \leftarrow \mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}).
\end{array}$$

$$\begin{array}{l}
\mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, [\mathbf{XI}|\mathbf{RI}]) \leftarrow \mathbf{add}(\mathbf{I}, \mathbf{X}, \mathbf{XI}), \\
\qquad \mathbf{incList}(\mathbf{R}, \mathbf{I}, \mathbf{RI}).
\end{array}$$



Size-change terminating! (acc. to [Lee, Jones, Ben-Amram, POPL 2001])

Too weak in logic programming...

$$\mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) \Rightarrow_{\{X \mapsto s(X')], \ Z \mapsto s(Z')\}} \mathbf{add}(\mathbf{X}', \mathbf{Y}, \mathbf{Z}')$$

$$\Rightarrow_{\{X' \mapsto s(X''), \ Z' \mapsto s(Z'')\}} \mathbf{add}(\mathbf{X}'', \mathbf{Y}, \mathbf{Z}'') \Rightarrow \infty$$

Example

$\mathbf{incList}([\,], \_, [\,]).$

$\mathbf{incList}([\mathbf{X}|\mathbf{R}], \mathbf{I}, \mathbf{L}) \leftarrow \mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, \mathbf{L}).$

$\mathbf{iList}(\mathbf{X}, \mathbf{R}, \mathbf{I}, [\mathbf{XI}|\mathbf{RI}]) \leftarrow \mathbf{add}(\mathbf{I}, \mathbf{X}, \mathbf{XI}),$
$\quad \mathbf{incList}(\mathbf{R}, \mathbf{I}, \mathbf{RI}).$

$\mathbf{add}(\mathbf{0}, \mathbf{Y}, \mathbf{Y}).$

$\mathbf{add}(\mathbf{s}(\mathbf{X}), \mathbf{Y}, \mathbf{s}(\mathbf{Z})) \leftarrow \mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}).$

$\mathcal{G}_{14}$: **incList** $\longrightarrow$ **incList**

$1_{incList} \xrightarrow{\succ} 1_{incList}$
$2_{incList} \xrightarrow{\succsim} 2_{incList}$
$3_{incList} \xrightarrow{\succ} 3_{incList}$

$\mathcal{G}_{41}$: **iList** $\longrightarrow$ **iList**

$2_{iList} \xrightarrow{\succ} 1_{iList}$
$2_{iList} \xrightarrow{\succ} 2_{iList}$
$3_{iList} \xrightarrow{\succsim} 3_{iList}$
$4_{iList} \xrightarrow{\succ} 4_{iList}$

$\mathcal{G}_{2}$: **add** $\longrightarrow$ **add**

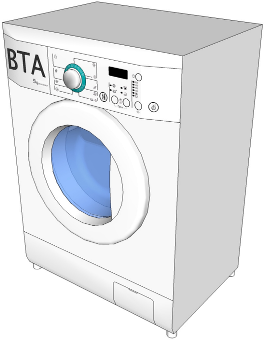
$1_{add} \xrightarrow{\succ} 1_{add}$
$2_{add} \xrightarrow{\succsim} 2_{add}$
$3_{add} \xrightarrow{\succ} 3_{add}$

Size-change terminating! (acc. to [Lee, Jones, Ben-Amram, POPL 2001])

Too weak in logic programming...

$$\mathbf{add}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) \Rightarrow_{\{\mathbf{X} \mapsto \mathbf{s}(\mathbf{X}')], \ \mathbf{Z} \mapsto \mathbf{s}(\mathbf{Z}')\}} \mathbf{add}(\mathbf{X}', \mathbf{Y}, \mathbf{Z}')$$
$$\Rightarrow_{\{\mathbf{X}' \mapsto \mathbf{s}(\mathbf{X}''), \ \mathbf{Z}' \mapsto \mathbf{s}(\mathbf{Z}'')\}} \mathbf{add}(\mathbf{X}'', \mathbf{Y}, \mathbf{Z}'') \Rightarrow \infty$$

A fully automatic BTA



Local termination

instantiated enough [Lindenstrauss, Sagiv, ICLP 1997]

Term t is instantiated enough w.r.t. $||\cdot||$ if $||t||$ is an integer

sufficient condition for termination

If every maximal graph for P contains at least one edge

$$i_p \xrightarrow{\succ} i_p$$

such that, in every possible call, $p(t_1, \ldots, t_n)$, the argument t_i is instantiated enough w.r.t. $||\cdot||$, then P is terminating

such that the i-th argument of p is classified as static, then P is terminating

and p is annotated with unfold (with memo otherwise)

Local termination

instantiated enough [Lindenstrauss, Sagiv, ICLP 1997]

Term t is instantiated enough w.r.t. $||\cdot||$ if $||t||$ is an integer

sufficient condition for termination

If every maximal graph for P contains at least one edge

$$i_p \xrightarrow{\succ} i_p$$

such that, in every possible call, $p(t_1, \ldots, t_n)$, the argument t_i is instantiated enough w.r.t. $||\cdot||$, then P is terminating

such that the i-th argument of p is classified as **static**, then P is terminating

and p is annotated with **unfold** (with **memo** otherwise)

Local termination

instantiated enough [Lindenstrauss, Sagiv, ICLP 1997]

Term t is instantiated enough w.r.t. $||\cdot||$ if $||t||$ is an integer

sufficient condition for termination

If every maximal graph for P contains at least one edge

$$i_p \xrightarrow{\succ} i_p$$

such that, in every possible call, $p(t_1, \ldots, t_n)$, the argument t_i is instantiated enough w.r.t. $||\cdot||$, then P is terminating

such that the i-th argument of p is classified as **static**, then P is terminating

and p is annotated with **unfold** (with **memo** otherwise)

Local termination

instantiated enough [Lindenstrauss, Sagiv, ICLP 1997]

Term t is instantiated enough w.r.t. $||\cdot||$ if $||t||$ is an integer

sufficient condition for termination

If every maximal graph for P contains at least one edge

$$i_p \xrightarrow{\succ} i_p$$

such that, in every possible call, $p(t_1, \ldots, t_n)$, the argument t_i is instantiated enough w.r.t. $||\cdot||$, then P is terminating

such that the i-th argument of p is classified as **static**, then P is terminating

and p is annotated with **unfold** (with **memo** otherwise)

Global termination

Must ensure quasi-termination (only finitely many different atoms)

Basic algorithm:

- for every predicate p and for every maximal graph for p, either
  - the previous condition hold or
  - there is an edge to every argument (no matter the label)
- and the considered symbolic norm is **bounded**

  i.e., the set $\{s \mid ||t|| \geqslant ||s||\}$ is finite for any term t

  $1_{add} \xrightarrow{\succ} 1_{add}$ with 1_{add} or 3_{add} static
  $2_{add} \xrightarrow{\succsim} 2_{add}$
  $3_{add} \xrightarrow{\succ} 3_{add}$

  $1_{iList} \xrightarrow{\succ} 1_{iList}$
  $2_{iList} \xrightarrow{\succ} 2_{iList}$
  $3_{iList} \xrightarrow{\succsim} 3_{iList}$
  $4_{iList} \xrightarrow{\succ} 4_{iList}$

- But the symbolic list-length norm is not bounded...

  $||[a]|| = ||[s(a)]|| = ||[s(s(a))]|| = \ldots = 1$

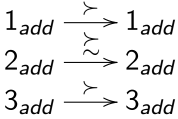
Global termination

Must ensure quasi-termination (only finitely many different atoms)

Basic algorithm:

- for every predicate p and for every maximal graph for p, either
  - the previous condition hold or
  - there is an edge to every argument (no matter the label)
- and the considered symbolic norm is **bounded**

  i.e., the set $\{s \mid ||t|| \geqslant ||s||\}$ is finite for any term t



with 1_{add} or 3_{add} static

- But the symbolic list-length norm is not bounded...

  $||[a]|| = ||[s(a)]|| = ||[s(s(a))]|| = \ldots = 1$

Global termination

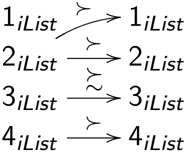
Must ensure quasi-termination (only finitely many different atoms)

Basic algorithm:

- for every predicate p and for every maximal graph for p, either
  - the previous condition hold or
  - there is an edge to every argument (no matter the label)
- and the considered symbolic norm is **bounded**

  i.e., the set $\{s \mid ||t|| \geqslant ||s||\}$ is finite for any term t

with 1_{add} or 3_{add} static



- But the symbolic list-length norm is not bounded...

  $||[a]|| = ||[s(a)]|| = ||[s(s(a))]|| = \ldots = 1$

Global termination

Must ensure quasi-termination (only finitely many different atoms)

Basic algorithm:

- for every predicate p and for every maximal graph for p, either
  - the previous condition hold or
  - there is an edge to every argument (no matter the label)
- and the considered symbolic norm is **bounded**

  i.e., the set $\{s \mid ||t|| \geqslant ||s||\}$ is finite for any term t

- But the symbolic list-length norm is not bounded...

  $||[a]|| = ||[s(a)]|| = ||[s(s(a))]|| = \ldots = 1$

Global termination

Must ensure quasi-termination (only finitely many different atoms)

Basic algorithm:

- for every predicate p and for every maximal graph for p, either
  - the previous condition hold or
  - there is an edge to every argument (no matter the label)
- and the considered symbolic norm is **bounded**

  i.e., the set $\{s \mid ||t|| \geqslant ||s||\}$ is finite for any term t

with 1_{add} or 3_{add} static

- But the symbolic list-length norm is not bounded...
  $||[a]|| = ||[s(a)]|| = ||[s(s(a))]|| = \ldots = 1$

annotations for global termination

given a predicate p and an argument i:

1. if every maximal graph for p fulfills the local termination condition, i_p is marked as **static**
2. otherwise, if $\exists$ a maximal graph with no input edge to i_p, then it is marked as **dynamic**

Furthermore, one should compute the lub w.r.t. the annotations computed by the algorithm for propagating static information

annotations for global termination

given a predicate *p* and an argument *i*:

1. if every maximal graph for *p* fulfills the local termination condition, i_p is marked as **static**
2. otherwise, if ∃ a maximal graph with no input edge to i_p, then it is marked as **dynamic**

Furthermore, one should compute the lub w.r.t. the annotations computed by the algorithm for propagating static information

annotations for global termination

given a predicate p and an argument i:

1. if every maximal graph for p fulfills the local termination condition, i_p is marked as **static**
2. otherwise, if $\exists$ a maximal graph with no input edge to i_p, then it is marked as **dynamic**

Furthermore, one should compute the lub w.r.t. the annotations computed by the algorithm for propagating static information

Example

incList([ ], _, [ ]).
incList([**X**|**R**], **I**, **L**) ← **iList**(**X**, **R**, **I**, **L**).

iList(**X**, **R**, **I**, [**XI**|**RI**]) ← **add**(**I**, **X**, **XI**),
incList(**R**, **I**, **RI**).

add(**0**, **Y**, **Y**).
add(**s**(**X**), **Y**, **s**(**Z**)) ← **add**(**X**, **Y**, **Z**).

$\mathcal{G}_{14}$: **incList** ⟶ **incList**

$1_{incList} \xrightarrow{\succ} 1_{incList}$
$2_{incList} \xrightarrow{\succsim} 2_{incList}$
$3_{incList} \xrightarrow{\succ} 3_{incList}$

$\mathcal{G}_{41}$: **iList** ⟶ **iList**

$2_{iList} \xrightarrow{\succ} 1_{iList}$
$2_{iList} \xrightarrow{\succ} 2_{iList}$
$3_{iList} \xrightarrow{\succsim} 3_{iList}$
$4_{iList} \xrightarrow{\succ} 4_{iList}$

$\mathcal{G}_{2}$: **add** ⟶ **add**

$1_{add} \xrightarrow{\succ} 1_{add}$
$2_{add} \xrightarrow{\succsim} 2_{add}$
$3_{add} \xrightarrow{\succ} 3_{add}$

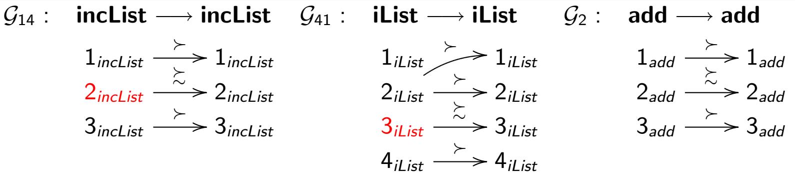
1. User's input: *incList(dynamic, static, dynamic)*
2. size-change analysis:
   - local termination: depends on the binding-times...
   - global termination: all arguments *static*
3. Propagation of BTs: *incList(D, S, D), iList(D, D, S, D), add(S, D, D)*
   - local termination: *incList* ↦ *memo*, *iList* ↦ *memo*, *add* ↦ *unfold*

Example

incList([ ], _, [ ]).
incList([**X**|**R**], **I**, **L**) ←**iList**(**X**, **R**, **I**, **L**).

iList(**X**, **R**, **I**, [**XI**|**RI**])←**add**(**I**, **X**, **XI**),
incList(**R**, **I**, **RI**).

add(**0**, **Y**, **Y**).
add(s(**X**), **Y**, s(**Z**))←**add**(**X**, **Y**, **Z**).



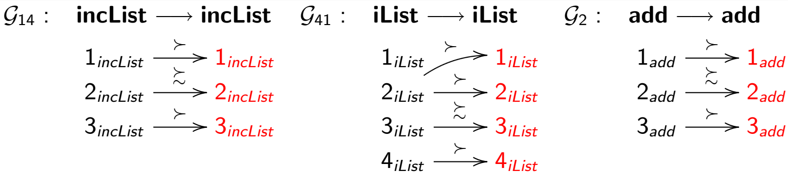
1. User's input: *incList(dynamic, static, dynamic)*
2. size-change analysis:
   - local termination: depends on the binding-times. . .
   - global termination: all arguments *static*
3. Propagation of BTs: *incList(D, S, D), iList(D, D, S, D), add(S, D, D)*
   - local termination: *incList* ↦ *memo*, *iList* ↦ *memo*, *add* ↦ *unfold*

Example

incList([ ], _, [ ]).
incList([**X**|**R**], **I**, **L**) ←**iList**(**X**, **R**, **I**, **L**).

iList(**X**, **R**, **I**, [**XI**|**RI**])←**add**(**I**, **X**, **XI**),
incList(**R**, **I**, **RI**).

add(**0**, **Y**, **Y**).
add(s(**X**), **Y**, s(**Z**))←**add**(**X**, **Y**, **Z**).



1. User's input: *incList*(*dynamic*, *static*, *dynamic*)
2. size-change analysis:
   - local termination: depends on the binding-times. . .
   - global termination: all arguments *static*
3. Propagation of BTs: *incList*(*D*, *S*, *D*), *iList*(*D*, *D*, *S*, *D*), *add*(*S*, *D*, *D*)
   - local termination: *incList* ↦ *memo*, *iList* ↦ *memo*, *add* ↦ *unfold*

Example

$\mathbf{incList}([\,], \_, [\,]).$
$\mathbf{incList}([X|R], I, L) \leftarrow \mathbf{iList}(X, R, I, L).$

$\mathbf{iList}(X, R, I, [XI|RI]) \leftarrow \mathbf{add}(I, X, XI),$
$\quad\quad \mathbf{incList}(R, I, RI).$

$\mathbf{add}(0, Y, Y).$
$\mathbf{add}(s(X), Y, s(Z)) \leftarrow \mathbf{add}(X, Y, Z).$

$\mathcal{G}_{14}$: **incList** $\longrightarrow$ **incList**

$1_{incList} \xrightarrow{\succ} 1_{incList}$
$2_{incList} \xrightarrow{\succsim} 2_{incList}$
$3_{incList} \xrightarrow{\succ} 3_{incList}$

$\mathcal{G}_{41}$: **iList** $\longrightarrow$ **iList**

$2_{iList} \xrightarrow{\succ} 1_{iList}$
$2_{iList} \xrightarrow{\succ} 2_{iList}$
$3_{iList} \xrightarrow{\succsim} 3_{iList}$
$4_{iList} \xrightarrow{\succ} 4_{iList}$

$\mathcal{G}_{2}$: **add** $\longrightarrow$ **add**

$1_{add} \xrightarrow{\succ} 1_{add}$
$2_{add} \xrightarrow{\succsim} 2_{add}$
$3_{add} \xrightarrow{\succ} 3_{add}$

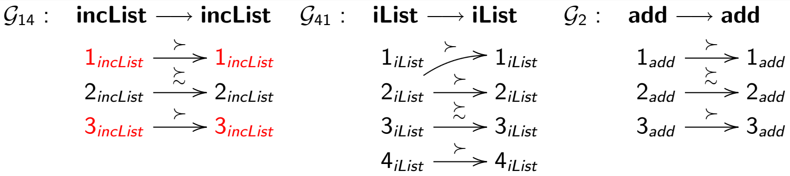
1. User's input: *incList(dynamic, static, dynamic)*
2. size-change analysis:
   - local termination: depends on the binding-times. . .
   - global termination: all arguments *static*
3. Propagation of BTs: *incList(D, S, D)*, *iList(D, D, S, D)*, *add(S, D, D)*
   - local termination: *incList* $\mapsto$ *memo*, *iList* $\mapsto$ *memo*, *add* $\mapsto$ *unfold*

Example

incList([ ], _, [ ]).
incList([**X**|**R**], **I**, **L**) ← **iList**(**X**, **R**, **I**, **L**).

iList(**X**, **R**, **I**, [**XI**|**RI**]) ← **add**(**I**, **X**, **XI**),
  incList(**R**, **I**, **RI**).

add(**0**, **Y**, **Y**).
add(**s**(**X**), **Y**, **s**(**Z**)) ← **add**(**X**, **Y**, **Z**).



1. User's input: *incList(dynamic, static, dynamic)*
2. size-change analysis:
   - local termination: depends on the binding-times...
   - global termination: all arguments *static*
3. Propagation of BTs: *incList(D, S, D)*, *iList(D, D, S, D)*, *add(S, D, D)*
   - local termination: *incList ↦ memo*, *iList ↦ memo*, *add ↦ unfold*

Example

incList([ ], _, [ ]).
incList([**X**|**R**], **I**, **L**) ←**iList**(**X**, **R**, **I**, **L**).

iList(**X**, **R**, **I**, [**XI**|**RI**])←**add**(**I**, **X**, **XI**),
incList(**R**, **I**, **RI**).

add(**0**, **Y**, **Y**).
add(**s**(**X**), **Y**, **s**(**Z**))←**add**(**X**, **Y**, **Z**).

$\mathcal{G}_{14}$: **incList** $\longrightarrow$ **incList**

$1_{incList} \xrightarrow{\succ} 1_{incList}$
$2_{incList} \xrightarrow{\succsim} 2_{incList}$
$3_{incList} \xrightarrow{\succ} 3_{incList}$

$\mathcal{G}_{41}$: **iList** $\longrightarrow$ **iList**

$2_{iList} \xrightarrow{\succ} 1_{iList}$
$2_{iList} \xrightarrow{\succ} 2_{iList}$
$3_{iList} \xrightarrow{\succsim} 3_{iList}$
$4_{iList} \xrightarrow{\succ} 4_{iList}$

$\mathcal{G}_{2}$: **add** $\longrightarrow$ **add**

$1_{add} \xrightarrow{\succ} 1_{add}$
$2_{add} \xrightarrow{\succsim} 2_{add}$
$3_{add} \xrightarrow{\succ} 3_{add}$

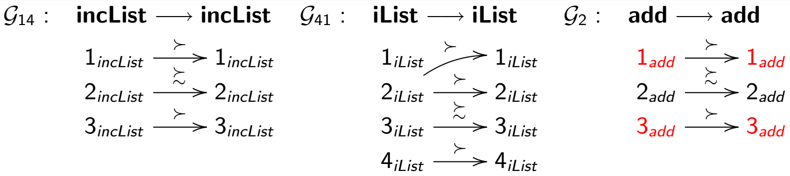
1. User's input: *incList(dynamic, static, dynamic)*
2. size-change analysis:
   - local termination: depends on the binding-times. . .
   - global termination: all arguments *static*
3. Propagation of BTs: *incList(D, S, D)*, *iList(D, D, S, D)*, *add(S, D, D)*
   - local termination: *incList* ↦ *memo*, *iList* ↦ *memo*, *add* ↦ *unfold*

Example

incList([ ], _, [ ]).
incList([**X**|**R**], **I**, **L**) ←**iList**(**X**, **R**, **I**, **L**).

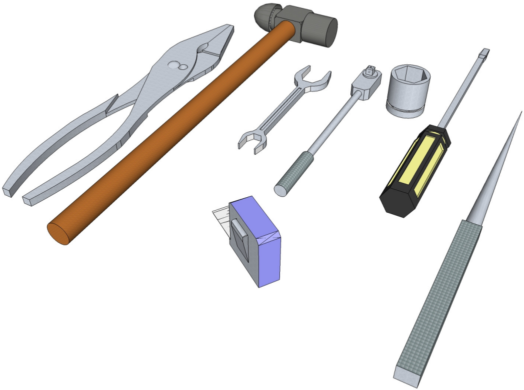
iList(**X**, **R**, **I**, [**XI**|**RI**])←**add**(**I**, **X**, **XI**),
incList(**R**, **I**, **RI**).

add(**0**, **Y**, **Y**).
add(s(**X**), **Y**, s(**Z**))←**add**(**X**, **Y**, **Z**).



1. User's input: *incList(dynamic, static, dynamic)*
2. size-change analysis:
   - local termination: depends on the binding-times. . .
   - global termination: all arguments *static*
3. Propagation of BTs: *incList(D, S, D)*, *iList(D, D, S, D)*, *add(S, D, D)*
   - local termination: *incList* ↦ *memo*, *iList* ↦ *memo*, *add* ↦ *unfold*

Some practical considerations



Concluding remarks

Very fast BTA, scales well to medium-sized Prolog programs

Ensures both local and global termination

Less accurate than previous BTA. . .

Much room for improvement:

- improve accuracy of size-change analysis
- hybrid approach: replace **memo** and **dynamic** with **online** and use online techniques for them

Concluding remarks

Very fast BTA, scales well to medium-sized Prolog programs

Ensures both local and global termination

Less accurate than previous BTA...

Much room for improvement:

- improve accuracy of size-change analysis
- hybrid approach: replace **memo** and **dynamic** with **online** and use online techniques for them