Deciding Entailments in Inductive Separation Logic with Tree Automata

Radu Iosif Adam Rogalewicz Tomáš Vojnar

VERIMAG, Université Joseph Fourier/CNRS, Grenoble, France
FIT, Brno University of Technology, Czech Republic

Vienna University of Technology

June 2015

Introduction

- A procedure for checking entailments in a fragment of separation logic with inductive predicates based on a reduction to checking inclusion on tree automata.
- Separation logic (SL)
  - among the most popular formalisms for reasoning about heaps,
  - allows for local reasoning
    - handling separately disjoint sub-heaps,
  - used in many tools: Space Invader, Slayer, Xisa, Predator, S2, ...

Introduction

- A procedure for checking entailments in a fragment of separation logic with inductive predicates based on a reduction to checking inclusion on tree automata.

- Separation logic (SL)
  - among the most popular formalisms for reasoning about heaps,
  - allows for local reasoning
    - handling separately disjoint sub-heaps,
  - used in many tools: Space Invader, Slayer, Xisa, Predator, S2, ...

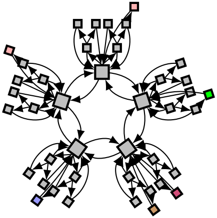
- Reasoning about heaps and dynamic linked data structures
  - crucial for many program analysis tasks,
  - notoriously difficult
    - dealing with infinite sets of complex graphs,
  - still under heavy research.

Introduction

- A procedure for checking entailments in a fragment of separation logic with inductive predicates based on a reduction to checking inclusion on tree automata.

- Separation logic (SL)
  - among the most popular formalisms for reasoning about heaps,
  - allows for local reasoning
    - handling separately disjoint sub-heaps,
  - used in many tools: Space Invader, Slayer, Xisa, Predator, S2, ...

- Reasoning about heaps and dynamic linked data structures
  - crucial for many program analysis tasks,
  - notoriously difficult
    - dealing with infinite sets of complex graphs,
  - still under heavy research.



Separation Logic

- Considered basic formulae of SL:

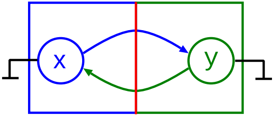
$$
\begin{array}{lcll}
\varphi & ::= & \exists x_1, ..., x_n \,.\, \Pi \wedge \Sigma & \\
\Pi & ::= & x_1 = x_2 \mid x = \textbf{nil} \mid \Pi_1 \wedge \Pi_2 & \text{pure part} \\
\Sigma & ::= & \textbf{emp} \mid x \mapsto (x_1, \ldots, x_n) \mid \Sigma_1 * \Sigma_2 & \text{spatial part}
\end{array}
$$

Separation Logic

- Considered basic formulae of SL:

$$\begin{array}{lcll}
\varphi & ::= & \exists x_1, ..., x_n \;.\; \Pi \wedge \Sigma & \\
\Pi & ::= & x_1 = x_2 \mid x = \textbf{nil} \mid \Pi_1 \wedge \Pi_2 & \text{pure part} \\
\Sigma & ::= & \textbf{emp} \mid x \mapsto (x_1, \ldots, x_n) \mid \Sigma_1 * \Sigma_2 & \text{spatial part}
\end{array}$$

- For example: $x \mapsto (y, u) * y \mapsto (v, x) \wedge \mathbf{u = nil} \wedge \mathbf{v = nil}$

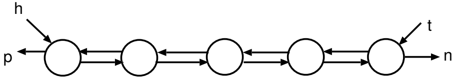


Inductive Definitions

- A system $\mathcal{P}$ of inductive definitions is an indexed set
  - $\{\ P_i(\mathbf{x}) \ \equiv \ \bigvee_j R_{i,j}(\mathbf{x}) \ \}_{i \in \{1,\ldots,n\}}$, $n \geq 1$.
- $R_{i,j}$ are rules of a predicate P_i:
  - $R_{i,j}(\mathbf{x}) \equiv \exists \mathbf{z} \ . \ \Sigma * P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_m}(\mathbf{y}_m) \ \wedge \ \Pi$

Inductive Definitions

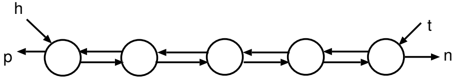
- A system $\mathcal{P}$ of inductive definitions is an indexed set
  - $\{ \ P_i(\mathbf{x}) \ \equiv \ \bigvee_j R_{i,j}(\mathbf{x}) \ \}_{i \in \{1,\ldots,n\}}$, $n \geq 1$.
- $R_{i,j}$ are rules of a predicate P_i:
  - $R_{i,j}(\mathbf{x}) \equiv \exists \mathbf{z} \ . \ \Sigma * P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_m}(\mathbf{y}_m) \ \wedge \ \Pi$
- For example:
  $\mathtt{DLL}(h, p, t, n) \equiv h \mapsto (n, p) \ \wedge \ h = t \mid \exists x. \ h \mapsto (x, p) * \mathtt{DLL}(x, h, t, n)$



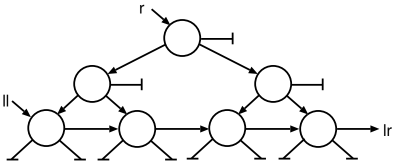
Inductive Definitions

- A system $\mathcal{P}$ of inductive definitions is an indexed set
  - $\{ \ P_i(\mathbf{x}) \ \equiv \ \bigvee_j R_{i,j}(\mathbf{x}) \ \}_{i \in \{1,\ldots,n\}}$, $n \geq 1$.
- $R_{i,j}$ are rules of a predicate P_i:
  - $R_{i,j}(\mathbf{x}) \equiv \exists \mathbf{z} \ . \ \Sigma * P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_m}(\mathbf{y}_m) \ \wedge \ \Pi$
- For example:

$$\mathtt{DLL}(h,p,t,n) \equiv h \mapsto (n,p) \ \wedge \ h = t \mid \exists x.\ h \mapsto (x,p) * \mathtt{DLL}(x,h,t,n)$$



$$\begin{aligned}
&\mathtt{TLL}(r, ll, lr) \equiv \\
&\quad r \mapsto (\mathbf{nil}, \mathbf{nil}, lr) \ \wedge \ r = ll \ \mid \\
&\quad \exists x, y, z.\ r \mapsto (x, y, \mathbf{nil}) \ * \\
&\qquad\qquad \mathtt{TLL}(x, ll, z) \ * \\
&\qquad\qquad \mathtt{TLL}(y, z, lr)
\end{aligned}$$



Restrictions

- One points-to predicate per rule:
  - YES: $R(x) \equiv \exists q.\ x \mapsto y * R(y)$,
  - NO: $R_1(x) \equiv \exists y, z, q.\ x \mapsto (y, z) * y \mapsto (q, x) * R_2(y) * R_3(q)$,
  - NO: $R_2(y, z) \equiv \mathit{emp} \wedge y = z$.

Restrictions

- One points-to predicate per rule:
  - YES: $R(x) \equiv \exists q.\ x \mapsto y * R(y)$,
  - NO: $R_1(x) \equiv \exists y, z, q.\ x \mapsto (y, z) * y \mapsto (q, x) * R_2(y) * R_3(q)$,
  - NO: $R_2(y, z) \equiv emp \wedge y = z$.

- Equalities restricted to allocated variables:
  - YES: $Q(x, y) \equiv \exists q.\ x \mapsto q \wedge x = y * R(q)$,
  - NO: $Q(x, y, z) \equiv \exists q.\ x \mapsto q \wedge y = z * R(q)$.

Restrictions

- One points-to predicate per rule:
  - YES: $R(x) \equiv \exists q.\ x \mapsto y * R(y)$,
  - NO: $R_1(x) \equiv \exists y, z, q.\ x \mapsto (y, z) * y \mapsto (q, x) * R_2(y) * R_3(q)$,
  - NO: $R_2(y, z) \equiv emp \wedge y = z$.
- Equalities restricted to allocated variables:
  - YES: $Q(x, y) \equiv \exists q.\ x \mapsto q \wedge x = y * R(q)$,
  - NO: $Q(x, y, z) \equiv \exists q.\ x \mapsto q \wedge y = z * R(q)$.
- Local edges only,
  - mapping (up to direction) to edges of a spanning tree,

Restrictions

- One points-to predicate per rule:
  - YES: $R(x) \equiv \exists q.\ x \mapsto y * R(y)$,
  - NO: $R_1(x) \equiv \exists y, z, q.\ x \mapsto (y, z) * y \mapsto (q, x) * R_2(y) * R_3(q)$,
  - NO: $R_2(y, z) \equiv emp \wedge y = z$.
- Equalities restricted to allocated variables:
  - YES: $Q(x, y) \equiv \exists q.\ x \mapsto q \wedge x = y * R(q)$,
  - NO: $Q(x, y, z) \equiv \exists q.\ x \mapsto q \wedge y = z * R(q)$.
- Local edges only,
  - mapping (up to direction) to edges of a spanning tree,
- Connected systems only.

Lifting the Restrictions

- Non-local and/or disconnected systems: incompleteness.

Lifting the Restrictions

- Non-local and/or disconnected systems: incompleteness.
- Rules with more points-to predicates can be (automatically) split.
  - E.g., $R_1(x) \equiv \exists y, z, q. x \mapsto (y, z) * y \mapsto (q, x) * R_2(y) * R_3(q)$ splits to:
    - $R_{1.1}(x) \;\equiv\; \exists y, z.\ x \mapsto (y, z) * R_{1.2}(x, y) * R_2(y)$ and
    - $R_{1.2}(x, y) \;\equiv\; \exists q.\ y \mapsto (q, x) * R_3(q)$.
  - Can lead to non-local edges or a disconnected system.

Lifting the Restrictions

- Non-local and/or disconnected systems: incompleteness.

- Rules with more points-to predicates can be (automatically) split.
  - E.g., $R_1(x) \equiv \exists y, z, q. x \mapsto (y, z) * y \mapsto (q, x) * R_2(y) * R_3(q)$ splits to:
    - $R_{1.1}(x) \;\equiv\; \exists y, z.\; x \mapsto (y, z) * R_{1.2}(x, y) * R_2(y)$ and
    - $R_{1.2}(x, y) \;\equiv\; \exists q.\; y \mapsto (q, x) * R_3(q)$.
  - Can lead to non-local edges or a disconnected system.

- Empty rules can be inlined.
  - E.g., for $Q_1(x, y) \equiv \exists z. x \mapsto (z) * Q_2(y, z)$, $Q_2(y, z) \equiv emp \wedge y = z$,
    - Inlining gives $Q(x, y) ::= x \mapsto y$.
  - Inlining can lead to forbidden equalities.

Lifting the Restrictions

- Non-local and/or disconnected systems: incompleteness.
- Rules with more points-to predicates can be (automatically) split.
  - E.g., $R_1(x) \equiv \exists y, z, q. x \mapsto (y, z) * y \mapsto (q, x) * R_2(y) * R_3(q)$ splits to:
    - $R_{1.1}(x) \;\equiv\; \exists y, z.\; x \mapsto (y, z) * R_{1.2}(x, y) * R_2(y)$ and
    - $R_{1.2}(x, y) \;\equiv\; \exists q.\; y \mapsto (q, x) * R_3(q)$.
  - Can lead to non-local edges or a disconnected system.
- Empty rules can be inlined.
  - E.g., for $Q_1(x, y) \equiv \exists z. x \mapsto (z) * Q_2(y, z)$, $Q_2(y, z) \equiv emp \wedge y = z$,
    - Inlining gives $Q(x, y) ::= x \mapsto y$.
  - Inlining can lead to forbidden equalities.
- General equalities can be removed:
  - tracking explicitly different combinations of equalities,
  - leads to an exponential blowup.

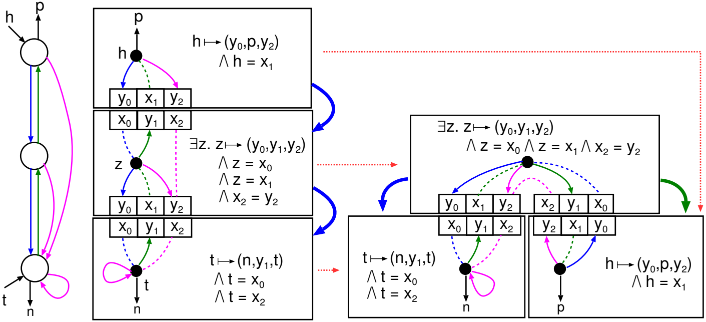
Basic Idea of the Procedure

- We reduce checking of $\varphi \models \psi$ to checking $L(A_\varphi) \subseteq L(A_\psi)$ where:

Basic Idea of the Procedure

- We reduce checking of $\varphi \models \psi$ to checking $L(A_\varphi) \subseteq L(A_\psi)$ where:
  - TA A_φ / A_ψ recognize unfolding trees of inductive definitions of φ / ψ,

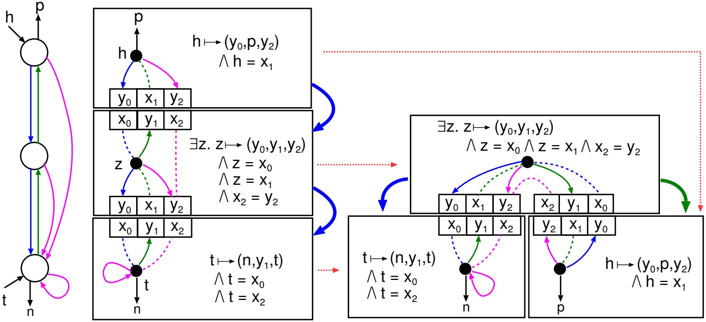
$TDLL(h, p, t, n) \equiv h \mapsto (n, p, t) \ \wedge \ h = t \ | \ \exists z. \ h \mapsto (z, p, t) * TDLL(z, h, t, n)$



Basic Idea of the Procedure

- We reduce checking of $\varphi \models \psi$ to checking $L(A_\varphi) \subseteq L(A_\psi)$ where:
  - TA A_φ / A_ψ recognize unfolding trees of inductive definitions of φ/ψ,
  - Rotation closure: dealing with possibly different spanning trees.

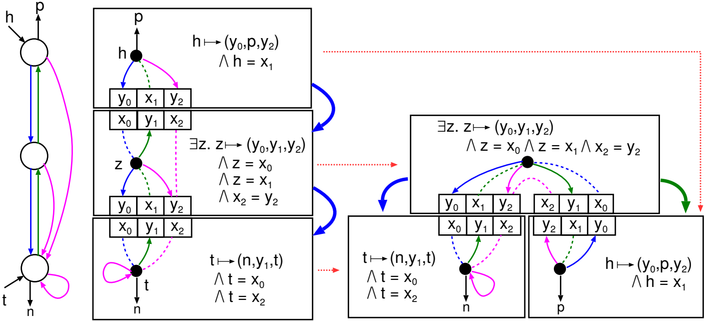
$$TDLL(h,p,t,n) \equiv h \mapsto (n,p,t) \ \wedge \ h = t \ \mid \ \exists z.\ h \mapsto (z,p,t) * TDLL(z,h,t,n)$$



Basic Idea of the Procedure

- We reduce checking of $\varphi \models \psi$ to checking $L(A_\varphi) \subseteq L(A_\psi)$ where:
    - TA A_φ / A_ψ recognize unfolding trees of inductive definitions of φ/ψ,
    - Rotation closure: dealing with possibly different spanning trees.
    - Alphabet – tiles: small graphs of the neighbourhood of allocated nodes.

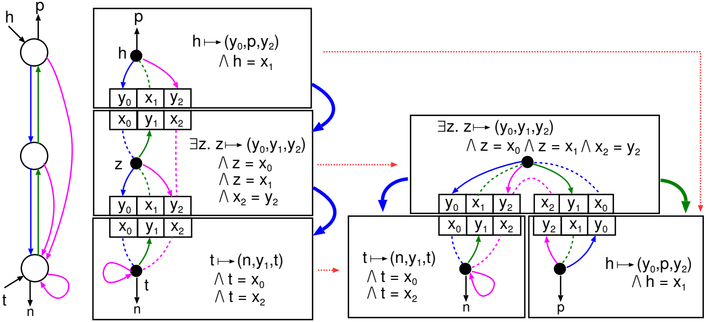
$$TDLL(h,p,t,n) \equiv h \mapsto (n,p,t) \ \wedge \ h = t \ \mid \ \exists z.\ h \mapsto (z,p,t) * TDLL(z,h,t,n)$$



Basic Idea of the Procedure

- We reduce checking of $\varphi \models \psi$ to checking $L(A_\varphi) \subseteq L(A_\psi)$ where:
  - TA A_φ / A_ψ recognize unfolding trees of inductive definitions of φ / ψ,
  - Rotation closure: dealing with possibly different spanning trees.
  - Alphabet – tiles: small graphs of the neighbourhood of allocated nodes.
  - Local edges: tree edges – composition of neighbouring tiles.

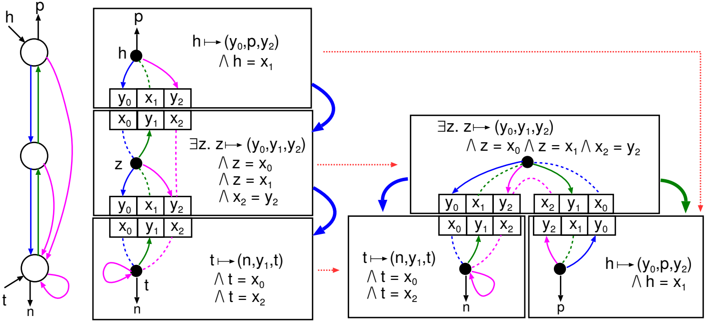
$$TDLL(h,p,t,n) \equiv h \mapsto (n,p,t) \ \wedge \ h = t \ \mid \ \exists z.\ h \mapsto (z,p,t) * TDLL(z,h,t,n)$$



Basic Idea of the Procedure

- We reduce checking of $\varphi \models \psi$ to checking $L(A_\varphi) \subseteq L(A_\psi)$ where:
  - TA A_φ / A_ψ recognize unfolding trees of inductive definitions of φ/ψ,
  - Rotation closure: dealing with possibly different spanning trees.
  - Alphabet – tiles: small graphs of the neighbourhood of allocated nodes.
  - Local edges: tree edges – composition of neighbouring tiles.
  - Non-local edges: sequences of equalities passed through tiles.

$$TDLL(h,p,t,n) \equiv h \mapsto (n,p,t) \ \wedge \ h = t \ \mid \ \exists z.\ h \mapsto (z,p,t) * TDLL(z,h,t,n)$$



$h \mapsto (y_0,p,y_2) \wedge h = x_1$

$\exists z.\ z \mapsto (y_0,y_1,y_2) \wedge z = x_0 \wedge z = x_1 \wedge x_2 = y_2$

$t \mapsto (n,y_1,t) \wedge t = x_0 \wedge t = x_2$

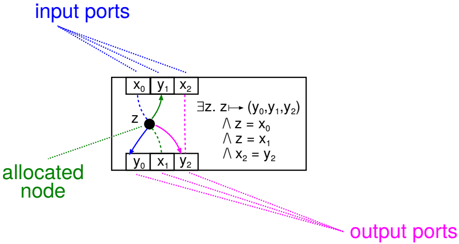
$\exists z.\ z \mapsto (y_0,y_1,y_2) \wedge z = x_0 \wedge z = x_1 \wedge x_2 = y_2$

$t \mapsto (n,y_1,t) \wedge t = x_0 \wedge t = x_2$

$h \mapsto (y_0,p,y_2) \wedge h = x_1$

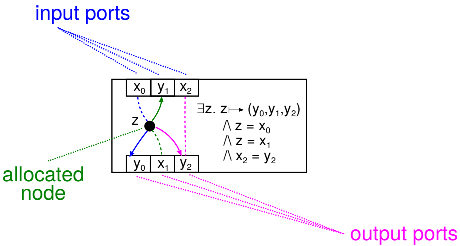
Tiles

- Tiles: small graphs of the neighbourhood of allocated nodes.



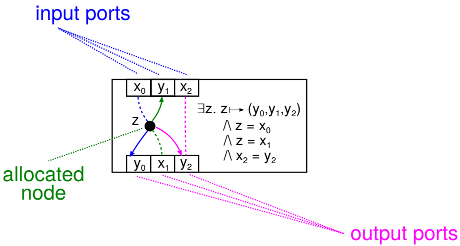
Tiles

- Tiles: small graphs of the neighbourhood of allocated nodes.
  - A single allocated node.



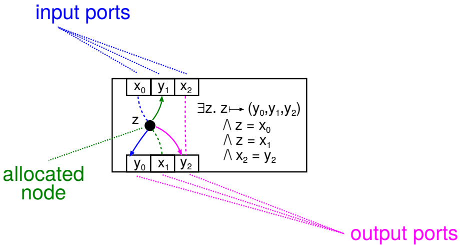
Tiles

- Tiles: small graphs of the neighbourhood of allocated nodes.
  - A single allocated node.
  - A single vector of input ports:
    - towards the root of the unfolding tree.



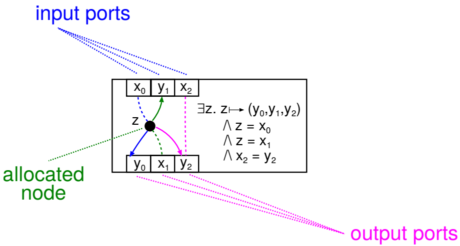
Tiles

- Tiles: small graphs of the neighbourhood of allocated nodes.
  - A single allocated node.
  - A single vector of input ports:
    - towards the root of the unfolding tree.
  - Possibly multiple vectors of output ports:
    - towards the leaves of the unfolding tree.



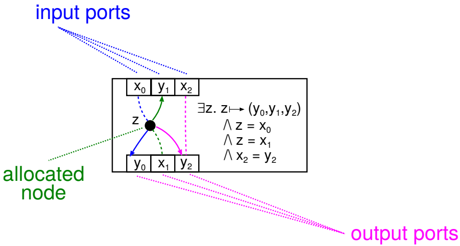
Tiles

- Tiles: small graphs of the neighbourhood of allocated nodes.
  - A single allocated node.
  - A single vector of input ports:
    - towards the root of the unfolding tree.
  - Possibly multiple vectors of output ports:
    - towards the leaves of the unfolding tree.
  - Two kinds of edges:
    - points-to edges: solid lines from the allocated node,
    - equality edges: dotted lines.



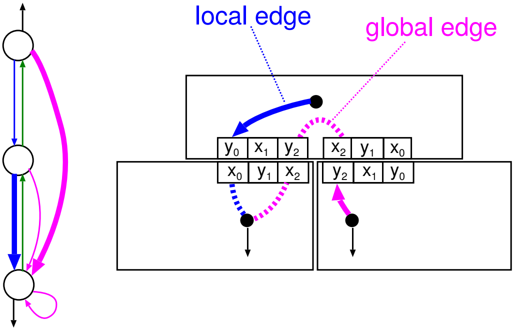
Tiles

- Tiles: small graphs of the neighbourhood of allocated nodes.
  - A single allocated node.
  - A single vector of input ports:
    - towards the root of the unfolding tree.
  - Possibly multiple vectors of output ports:
    - towards the leaves of the unfolding tree.
  - Two kinds of edges:
    - points-to edges: solid lines from the allocated node,
    - equality edges: dotted lines.
  - Can be described by a simple SL formula.



Tile Composition

- Local edges:
  - correspond to edges of unfolding trees,
  - composition of a single points-to and a single equality edge.
- Global edges:
  - span multiple tree edges,
  - composition of a single points-to and ≥ 2 equality edges.



Specialization

- Top-most tiles have no input ports.
  - Parameters of top-most predicate calls are replaced by free variables.
  - For that, a specialised version of the top-level predicate is created.

- For example:
  - when $\mathtt{DLL}(a, b, c, d)$ is used on the top level,
    - $\mathtt{DLL}(h, p, t, n) \;\equiv\; \exists x.\ h \mapsto (x, p) * \mathtt{DLL}(x, h, t, n) \;\mid\; h \mapsto (n, p) \wedge h = t,$

Specialization

- Top-most tiles have no input ports.
  - Parameters of top-most predicate calls are replaced by free variables.
  - For that, a specialised version of the top-level predicate is created.

- For example:
  - when $\mathtt{DLL}(a, b, c, d)$ is used on the top level,
    - $\mathtt{DLL}(h, p, t, n) \ \equiv \ \exists x.\ h \mapsto (x, p) * \mathtt{DLL}(x, h, t, n) \ \mid \ h \mapsto (n, p) \wedge h = t,$
  - the top call is transformed to $\mathtt{DLL}'()$,
    - $\mathtt{DLL}'() \ \equiv \ \exists x.\ a \mapsto (x, b) * \mathtt{DLL}(x, a, c, d) \ \mid \ a \mapsto (d, b) \wedge a = c.$

Translation to Tree Automata

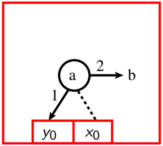
- A system of inductive definitions $\mathcal{P}$ is translated to a TA $A_{\mathcal{P}}$:
    - Each predicate P maps to a single TA state q_P.
    - Predicates with no parameters become final states (for bottom-up TA).
    - Each predicate rule is translated to a TA rule.

Translation to Tree Automata

- A system of inductive definitions $\mathcal{P}$ is translated to a TA $A_{\mathcal{P}}$:
  - Each predicate P maps to a single TA state q_P.
  - Predicates with no parameters become final states (for bottom-up TA).
  - Each predicate rule is translated to a TA rule.

- For example:
  - $\mathbf{DLL}'() \equiv \exists x.\ a \mapsto (x, b) * \mathbf{DLL}(x, a, c, d) \quad \rightsquigarrow \quad q_{\mathbf{DLL}} \xrightarrow{T_1^A} q_{\mathbf{DLL}'}$

T_1^A



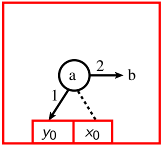
Translation to Tree Automata

- A system of inductive definitions $\mathcal{P}$ is translated to a TA $A_{\mathcal{P}}$:
  - Each predicate P maps to a single TA state q_P.
  - Predicates with no parameters become final states (for bottom-up TA).
  - Each predicate rule is translated to a TA rule.

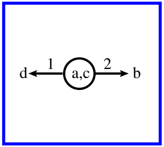
- For example:
  - $\mathbf{DLL}'() \equiv \exists x.\ a \mapsto (x, b) * \mathbf{DLL}(x, a, c, d) \quad \rightsquigarrow \quad q_{\mathbf{DLL}} \xrightarrow{T_1^A} q_{\mathbf{DLL}'}$

    $\mid a \mapsto (d, b) \wedge a = c \quad \rightsquigarrow \quad \xrightarrow{T_1^B} q_{\mathbf{DLL}'}$

T_1^A



T_1^B

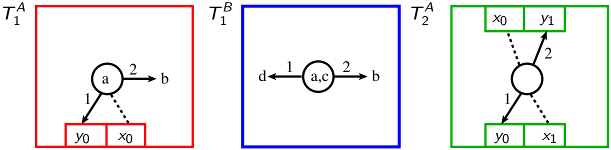


Translation to Tree Automata

- A system of inductive definitions $\mathcal{P}$ is translated to a TA $A_{\mathcal{P}}$:
  - Each predicate P maps to a single TA state q_P.
  - Predicates with no parameters become final states (for bottom-up TA).
  - Each predicate rule is translated to a TA rule.

- For example:
  - $\mathbf{DLL}'() \equiv \exists x.\ a \mapsto (x,b) * \mathbf{DLL}(x,a,c,d) \quad \rightsquigarrow \quad q_{\mathbf{DLL}} \xrightarrow{T_1^A} q_{\mathbf{DLL}'}$

    $\mid a \mapsto (d,b) \wedge a = c \quad \rightsquigarrow \quad \xrightarrow{T_1^B} q_{\mathbf{DLL}'}$
  - $\mathbf{DLL}(h,p,t,n) \equiv \exists x.\ h \mapsto (x,p) * \mathbf{DLL}(x,h,t,n) \quad \rightsquigarrow \quad q_{\mathbf{DLL}} \xrightarrow{T_2^A} q_{\mathbf{DLL}}$

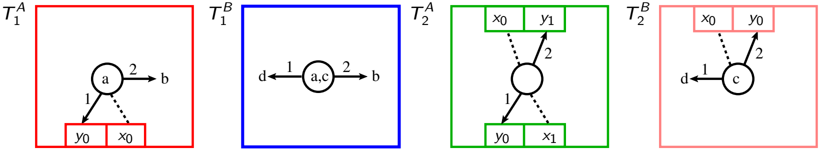


Translation to Tree Automata

- A system of inductive definitions $\mathcal{P}$ is translated to a TA $A_{\mathcal{P}}$:
  - Each predicate P maps to a single TA state q_P.
  - Predicates with no parameters become final states (for bottom-up TA).
  - Each predicate rule is translated to a TA rule.
- For example:
  - $\mathbf{DLL}'() \equiv \exists x.\ a \mapsto (x, b) * \mathbf{DLL}(x, a, c, d) \quad \rightsquigarrow \quad q_{\mathbf{DLL}} \xrightarrow{T_1^A} q_{\mathbf{DLL}'}$

    $\quad | \ a \mapsto (d, b) \wedge a = c \quad \rightsquigarrow \quad \xrightarrow{T_1^B} q_{\mathbf{DLL}'}$
  - $\mathbf{DLL}(h, p, t, n) \equiv \exists x.\ h \mapsto (x, p) * \mathbf{DLL}(x, h, t, n) \quad \rightsquigarrow \quad q_{\mathbf{DLL}} \xrightarrow{T_2^A} q_{\mathbf{DLL}}$

    $\quad | \ h \mapsto (x, p) \wedge h = t \quad \rightsquigarrow \quad \xrightarrow{T_2^B} q_{\mathbf{DLL}}$



Entailment Checking

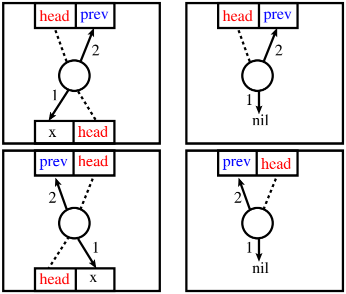
- The described translation from inductive SL definitions to TA gives an incomplete entailment checking procedure:
  - $\mathcal{L}(A_{\mathcal{P}_1}) \subseteq \mathcal{L}(A_{\mathcal{P}_2}) \Rightarrow \mathcal{P}_1 \models \mathcal{P}_2$
- An *EXPTIME* upper bound.
- To get a complete procedure, one has to tackle:
  - Canonical tiling of the system of predicates.
  - Possibly different spanning trees of the same structure.

(Quasi-)Canonical Tiles

- Different orderings of predicate parameters give different tiles, e.g., for a slightly simplified DLL predicate:

$$\mathtt{DLL}_A(head, prev) \equiv$$
$$\exists x.\ head \mapsto (x, prev) * \mathtt{DLL}(x, head)$$
$$\mid\ head \mapsto (\mathbf{nil}, prev)$$

$$\mathtt{DLL}_B(prev, head) \equiv$$
$$\exists x.\ head \mapsto (x, prev) * \mathtt{DLL}(head, x)$$
$$\mid\ head \mapsto (\mathbf{nil}, prev)$$

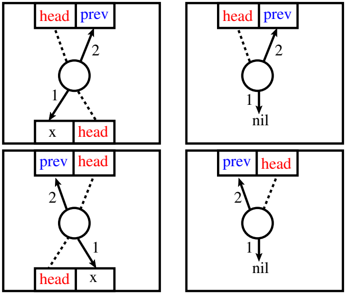


(Quasi-)Canonical Tiles

- Different orderings of predicate parameters give different tiles, e.g., for a slightly simplified DLL predicate:

$\mathtt{DLL}_A(\mathit{head}, \mathit{prev}) \equiv$
$\exists x.\ \mathit{head} \mapsto (x, \mathit{prev}) * \mathtt{DLL}(x, \mathit{head})$
$\mid \mathit{head} \mapsto (\mathbf{nil}, \mathit{prev})$

$\mathtt{DLL}_B(\mathit{prev}, \mathit{head}) \equiv$
$\exists x.\ \mathit{head} \mapsto (x, \mathit{prev}) * \mathtt{DLL}(\mathit{head}, x)$
$\mid \mathit{head} \mapsto (\mathbf{nil}, \mathit{prev})$



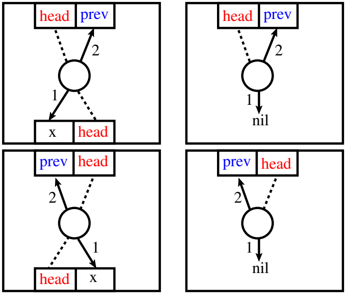
- Order the vectors of input/output ports as follows:
  1. Ports corresponding to forward local edges ordered wrt. selectors.

(Quasi-)Canonical Tiles

- Different orderings of predicate parameters give different tiles, e.g., for a slightly simplified DLL predicate:

$$\mathtt{DLL}_A(head, prev) \equiv$$
$$\exists x.\ head \mapsto (x, prev) * \mathtt{DLL}(x, head)$$
$$\mid\ head \mapsto (\mathbf{nil}, prev)$$

$$\mathtt{DLL}_B(prev, head) \equiv$$
$$\exists x.\ head \mapsto (x, prev) * \mathtt{DLL}(head, x)$$
$$\mid\ head \mapsto (\mathbf{nil}, prev)$$



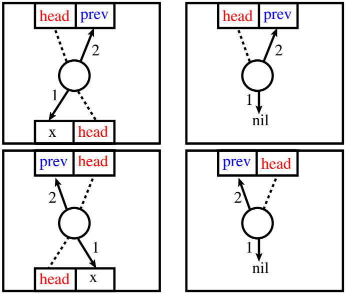
- Order the vectors of input/output ports as follows:
  1. Ports corresponding to forward local edges ordered wrt. selectors.
  2. Ports corresponding to backward local edges ordered wrt. selectors.

(Quasi-)Canonical Tiles

- Different orderings of predicate parameters give different tiles, e.g., for a slightly simplified DLL predicate:

$$\begin{aligned}&\mathtt{DLL}_A(\mathit{head}, \mathit{prev}) \equiv\\ &\exists x.\ \mathit{head} \mapsto (x, \mathit{prev}) * \mathtt{DLL}(x, \mathit{head})\\ &\mid\ \mathit{head} \mapsto (\mathbf{nil}, \mathit{prev})\end{aligned}$$

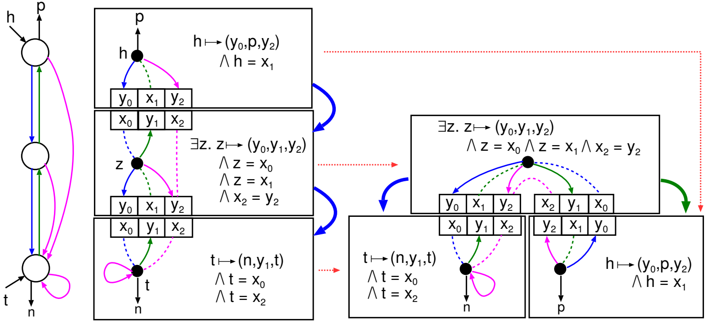
$$\begin{aligned}&\mathtt{DLL}_B(\mathit{prev}, \mathit{head}) \equiv\\ &\exists x.\ \mathit{head} \mapsto (x, \mathit{prev}) * \mathtt{DLL}(\mathit{head}, x)\\ &\mid\ \mathit{head} \mapsto (\mathbf{nil}, \mathit{prev})\end{aligned}$$



- Order the vectors of input/output ports as follows:
  1. Ports corresponding to forward local edges ordered wrt. selectors.
  2. Ports corresponding to backward local edges ordered wrt. selectors.
  3. Ports corresponding to non-local edges,
     - not ordered: leading to quasi-canonicity in this case.

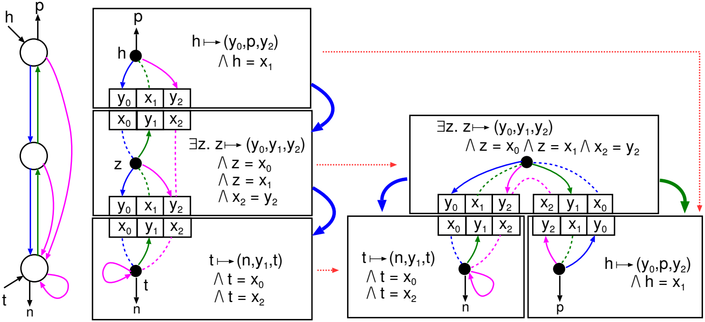
Different Spanning Trees

- Data structures can be represented using different spanning trees:



Different Spanning Trees

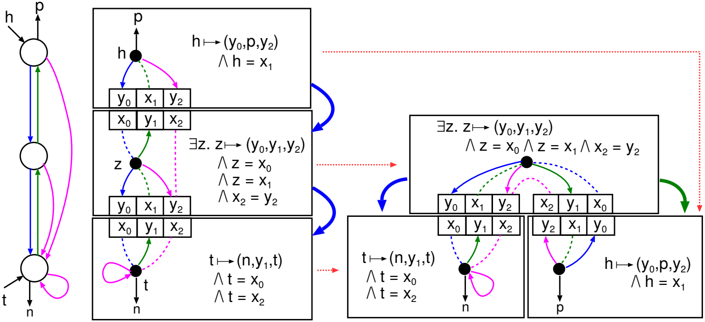
- Data structures can be represented using different spanning trees:



- The different spanning trees are often equal up to rotation.
  - A mapping which preserves neighbouring nodes of each node.

Different Spanning Trees

- Data structures can be represented using different spanning trees:



- The different spanning trees are often equal up to rotation.
  - A mapping which preserves neighbouring nodes of each node.
- The above always holds for systems with local edges.

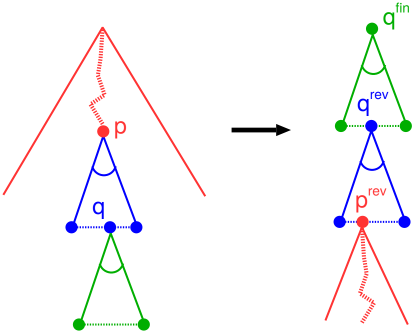
Rotation Closure on TA

- Dealing with different spanning trees:
  1. Generate a TA for one kind of spanning trees.
  2. Close the TA under rotation.

Rotation Closure on TA

- Dealing with different spanning trees:
  1. Generate a TA for one kind of spanning trees.
  2. Close the TA under rotation.

- Rotation closure is easy to implement on TA:
  - $T(p_1, \ldots, p_m) \to q$ changes to $T_{new}(p_1, \ldots, q^{rev}, \ldots p_m) \to q^{fin}$.
  - $T(q_1, \ldots, q, \ldots, q_n) \to p$ changes to $T_{new}(q_1, \ldots, p^{rev}, \ldots q_n) \to q^{rev}$.



Completeness of the Entailment Check

- For local, connected inductive systems, the described procedure with canonization and rotation closure is sound and complete, i.e.,
  - $\mathcal{L}(A_{\mathcal{P}_1}) \subseteq \mathcal{L}(A^r_{\mathcal{P}_2}) \Leftrightarrow \mathcal{P}_1 \models \mathcal{P}_2$.
- *EXPTIME* upper bound.
- Quasi-canonization and rotation closure improve completeness for systems with non-local edges also.

Implementation and Experimental Results

- Implemented in a tool called SLIDE:

  `http://www.fit.vutbr.cz/research/groups/verifit/tools/slide/`

- Tested successfully on a number of experiments:

Entailment $LHS \models RHS$	Answer	$\lvert A_{lhs} \rvert$	$\lvert A_{rhs} \rvert$	$\lvert A^{r}_{rhs} \rvert$
$\mathtt{DLL}(a, \mathbf{nil}, c, \mathbf{nil}) \models \mathtt{DLL}_{rev}(a, \mathbf{nil}, c, \mathbf{nil})$	True	2/4	2/4	5/8
$\mathtt{DLL}_{rev}(a, \mathbf{nil}, c, \mathbf{nil}) \models \mathtt{DLL}_{mid}(a, \mathbf{nil}, c, \mathbf{nil})$	True	2/4	4/8	12/18
$\mathtt{DLL}_{mid}(a, \mathbf{nil}, c, \mathbf{nil}) \models \mathtt{DLL}(a, \mathbf{nil}, c, \mathbf{nil})$	True	4/8	2/4	5/8
$\exists x, n, b.\ x \mapsto (n, b) * \mathtt{DLL}_{rev}(a, \mathbf{nil}, b, x) * \mathtt{DLL}(n, x, c, \mathbf{nil}) \models \mathtt{DLL}(a, \mathbf{nil}, c, \mathbf{nil})$	True	3/5	2/4	5/8
$\mathtt{DLL}(a, \mathbf{nil}, c, \mathbf{nil}) \models \exists x, n, b.\ x \mapsto (n, b) * \mathtt{DLL}_{rev}(a, \mathbf{nil}, b, x) * \mathtt{DLL}(n, x, c, \mathbf{nil})$	False	2/4	3/5	9/13
$\exists y, a.\ x \mapsto (y, \mathbf{nil}) * y \mapsto (a, x) * \mathtt{DLL}(a, y, c, \mathbf{nil}) \models \mathtt{DLL}(x, \mathbf{nil}, c, \mathbf{nil})$	True	3/4	2/4	5/8
$\mathtt{DLL}(x, \mathbf{nil}, c, \mathbf{nil}) \models \exists y, a.\ x \mapsto (\mathbf{nil}, y) * y \mapsto (a, x) * \mathtt{DLL}(a, y, c, \mathbf{nil})$	False	2/4	3/4	8/10
$\exists x, b. \mathtt{DLL}(x, b, c, \mathbf{nil}) * \mathtt{DLL}_{rev}(a, \mathbf{nil}, b, x) \models \mathtt{DLL}(a, \mathbf{nil}, c, \mathbf{nil})$	True	3/6	2/4	5/8
$\mathtt{DLL}(a, \mathbf{nil}, c, \mathbf{nil}) \models \mathtt{DLL}_{0+}(a, \mathbf{nil}, c, \mathbf{nil})$	True	2/4	2/4	5/8
$\mathtt{TREE}_{pp}(a, \mathbf{nil}) \models \mathtt{TREE}^{rev}_{pp}(a, \mathbf{nil})$	True	2/4	3/8	6/11
$\mathtt{TREE}^{rev}_{pp}(a, \mathbf{nil}) \models \mathtt{TREE}_{pp}(a, \mathbf{nil})$	True	3/8	2/4	5/10
$\mathtt{TLL}_{pp}(a, \mathbf{nil}, c, \mathbf{nil}) \models \mathtt{TLL}^{rev}_{pp}(a, \mathbf{nil}, c, \mathbf{nil})$	True	4/8	4/8	13/22
$\mathtt{TLL}^{rev}_{pp}(a, \mathbf{nil}, c, \mathbf{nil}) \models \mathtt{TLL}_{pp}(a, \mathbf{nil}, c, \mathbf{nil})$	True	4/8	4/8	13/22
$\exists l, r, z.\ a \mapsto (l, r, \mathbf{nil}, \mathbf{nil}) * \mathtt{TLL}(l, c, z) * \mathtt{TLL}(r, z, \mathbf{nil}) \models \mathtt{TLL}(a, c, \mathbf{nil})$	True	4/7	4/8	13/22
$\mathtt{TLL}(a, c, \mathbf{nil}) \models \exists l, r, z.\ a \mapsto (l, r, \mathbf{nil}, \mathbf{nil}) * \mathtt{TLL}(l, c, z) * \mathtt{TLL}(r, z, \mathbf{nil})$	False	4/8	4/7	13/21

- SLCOMP'14: 2nd (out of 3 participants) in the UDB devision.

Related Approaches and Summary

- Lists:
    - many entailment procedures,
        - recently, e.g., SPEN: graph homomorphisms, SAT, TA membership.
    - Often with hard-coded predicates and/or incomplete.
    - Special procedures in analysers like Space Invader, Predator, or Infer.

Related Approaches and Summary

- Lists:
  - many entailment procedures,
    - recently, e.g., SPEN: graph homomorphisms, SAT, TA membership.
  - Often with hard-coded predicates and/or incomplete.
  - Special procedures in analysers like Space Invader, Predator, or Infer.
- Trees:
  - GRIT: based on translation to SMT, more restricted than our approach.

Related Approaches and Summary

- Lists:
  - many entailment procedures,
    - recently, e.g., SPEN: graph homomorphisms, SAT, TA membership.
  - Often with hard-coded predicates and/or incomplete.
  - Special procedures in analysers like Space Invader, Predator, or Infer.
- Trees:
  - GRIT: based on translation to SMT, more restricted than our approach.
- User-defined predicates:
  - Sleek, Cyclist – incomplete procedures.

Related Approaches and Summary

- Lists:
    - many entailment procedures,
        - recently, e.g., SPEN: graph homomorphisms, SAT, TA membership.
    - Often with hard-coded predicates and/or incomplete.
    - Special procedures in analysers like Space Invader, Predator, or Infer.
- Trees:
    - GRIT: based on translation to SMT, more restricted than our approach.
- User-defined predicates:
    - Sleek, Cyclist – incomplete procedures.
- Iosif, Rogalewicz 2013: bounded tree width data structures:
    - complete procedure based on translation from SL to MSO on graphs,
    - multiply exponential.

Related Approaches and Summary

- Lists:
  - many entailment procedures,
    - recently, e.g., SPEN: graph homomorphisms, SAT, TA membership.
  - Often with hard-coded predicates and/or incomplete.
  - Special procedures in analysers like Space Invader, Predator, or Infer.
- Trees:
  - GRIT: based on translation to SMT, more restricted than our approach.
- User-defined predicates:
  - Sleek, Cyclist – incomplete procedures.
- Iosif, Rogalewicz 2013: bounded tree width data structures:
  - complete procedure based on translation from SL to MSO on graphs,
  - multiply exponential.

- The proposed approach:
  - Lists, trees, user-defined predicates.
  - Complete on a rich class of structures, *EXPTIME*-complete.

Future Work

- Better support of top-level formulae:
  - disconnected systems, Boolean skeleton, ...
- Better implementation, more experiments.
- Integration of the procedure into some verification tool.