Interprocedural Shape Analysis Using Separation Logic-based Transformer Summaries

H. Illous[1,2], M. Lemerre[1], X. Rival[2]

[1]CEA, LIST

[2]CNRS/ENS/INRIA/PSL*

Introduction

- **State analyses**: Computes a set of reachable states
  to verify state properties:
    - Can this program perform a null pointer dereference?
    - Does this program preserve structural invariants of data structures?

Introduction

- **State analyses**: Computes a set of reachable states to verify state properties:
  - Can this program perform a null pointer dereference?
  - Does this program preserve structural invariants of data structures?
- **Transformation analyses**: Compute **abstract transformations**, i.e. relations between program input state and output state:
  - Does this program modify the linked list received as an argument?
  - Is this sorting algorithm in-place?

Introduction

- **State analyses**: Computes a set of reachable states to verify state properties:
  - Can this program perform a null pointer dereference?
  - Does this program preserve structural invariants of data structures?
- **Transformation analyses**: Compute **abstract transformations**, i.e. relations between program input state and output state:
  - Does this program modify the linked list received as an argument?
  - Is this sorting algorithm in-place?

This work

- **Abstract transformations** as **procedure summaries**

Introduction

- **State analyses**: Computes a set of reachable states to verify state properties:
  - Can this program perform a null pointer dereference?
  - Does this program preserve structural invariants of data structures?
- **Transformation analyses**: Compute **abstract transformations**, i.e. relations between program input state and output state:
  - Does this program modify the linked list received as an argument?
  - Is this sorting algorithm in-place?

This work

- **Abstract transformations** as **procedure summaries**
- Applied to **shape analysis** using **separation logic**.

Overview

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

State analysis by inlining

```
double_append(list* k₀,list* k₁,list* k₂){
```

$h_0^\sharp$

```
  append(k₀,k₁);


  append(k₀,k₂);


}
append(list* l₀,list* l₁){


  while(l₀→n≠0x0){l₀=l₀→n;}
  l₀→n = l₁;


}
```

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){
```

$h^\sharp_0$

```
  append(k0,k1);


  append(k0,k2);


}

append(list* l0,list* l1){
```

$h^\sharp_1$

```
  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;


}
```

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){
                    h♯0
  append(k0,k1);

  append(k0,k2);

}
append(list* l0,list* l1){
                            h♯1
  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
                            h♯19
}
```

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){
```

$h^\sharp_0$

```
  append(k0,k1);
```

$h^\sharp_{20}$

```
  append(k0,k2);


}

append(list* l0,list* l1){


  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;


}
```

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){
                    h♯0
  append(k0,k1);
                    h♯20
  append(k0,k2);


}
append(list* l0,list* l1){
                              h♯21
  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;


}
```

$h^\sharp_0$, $h^\sharp_{20}$, $h^\sharp_{21}$

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){
```

$h^\sharp_0$

```
  append(k0,k1);
```

$h^\sharp_{20}$

```
  append(k0,k2);


}
```

```
append(list* l0,list* l1){
```

$h^\sharp_{21}$

```
  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

$h^\sharp_{39}$

```
}
```

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){
```

$h^\sharp_0$

```
  append(k0,k1);
```

$h^\sharp_{20}$

```
  append(k0,k2);
```

$h^\sharp_{40}$

```
}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

State analysis by inlining

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}

append(list* l0,list* l1){

  while(l0→n≠
  l0→n = l1;

}
```

- + Precise analysis of procedures
- − Analysis of `append` is repeated for each calling context
- − Cannot handle recursive procedures

Abstract transformations as procedure summaries

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

Abstract transformations as procedure summaries

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

$t^\sharp$

Abstract transformations as procedure summaries

```
double_append(list* k0,list* k1,list* k2){
```

$h_0^\sharp$

```
  append(k0,k1);


  append(k0,k2);


}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
```

$\downarrow t^\sharp$

```
}
```

```
double_append(list* k0,list* k1,list* k2){
```

$h_0^\sharp$

```
  append(k0,k1);
```

$h_1^\sharp = t^\sharp(h_0^\sharp)$

```
  append(k0,k2);

}
```

```
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

$t^\sharp$

Abstract transformations as procedure summaries

```
double_append(list* k0,list* k1,list* k2){
```

$$h_0^\sharp$$

```
  append(k0,k1);
```

$$h_1^\sharp = t^\sharp(h_0^\sharp)$$

```
  append(k0,k2);
```

$$h_2^\sharp = t^\sharp(h_1^\sharp)$$

```
}
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

$$\Big\downarrow t^\sharp$$

Abstract transformations as procedure summaries

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

$t^\sharp$

- Applying an abstract transformation can speed up a state analysis.

Modular analysis by composition of abstract transformations

```
double_append(list* k0,list* k1,list* k2){


  append(k0,k1);


  append(k0,k2);


}
append(list* l0,list* l1){


  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;


}
```

Modular analysis by composition of abstract transformations

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

$t^\sharp$

Modular analysis by composition of abstract transformations

```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow \mathtt{Id}(h^\sharp_0)$

```
  append(k0,k1);


  append(k0,k2);


}
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

$\downarrow t^\sharp$

Modular analysis by composition of abstract transformations

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

$\mathtt{Id}(h_0^\sharp)$

$t^\sharp \circ \mathtt{Id}(h_0^\sharp)$

$t^\sharp$

Modular analysis by composition of abstract transformations

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

$\mathtt{Id}(h_0^\sharp)$

$t^\sharp \circ \mathtt{Id}(h_0^\sharp)$

$t^\sharp \circ t^\sharp \circ \mathtt{Id}(h_0^\sharp)$

$t^\sharp$

Modular analysis by composition of abstract transformations

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}
append(list* l0,list* l1){

  while(l0→n≠
  l0→n = l1;

}
```

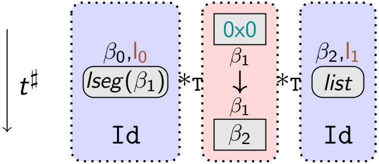
- Composition of relations can produce a new summary from summaries of callee functions.
- Summary was created for a given input state

Modular analysis by composition of abstract transformations

```
double_append(list* k0,list* k1,list* k2){
```

$$\overset{\alpha_0,k_0}{\boxed{lseg(\alpha_1)}} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1}{\boxed{lseg(\alpha_3)}} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,k_2}{\boxed{list}}$$

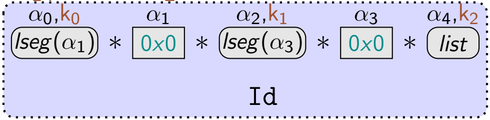
```
  append(k0,k1);

  append(k0,k2);

}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```



Modular analysis by composition of abstract transformations

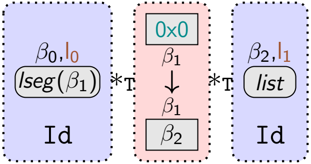
```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow \mathtt{Id}(h_0^\sharp)$




```
  append(k0,k1);


  append(k0,k2);


}

append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

$\downarrow t^\sharp$

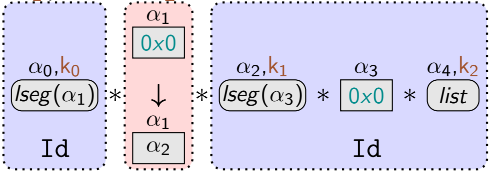


Modular analysis by composition of abstract transformations

```
double_append(list* k0,list* k1,list* k2){

  append(k0,k1);

  append(k0,k2);

}
```

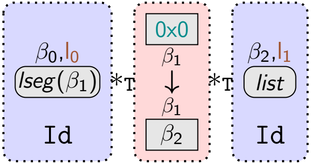
$\Downarrow \mathtt{Id}(h_0^\sharp)$

$t^\sharp \circ \mathtt{Id}(h_0^\sharp)$




```
append(list* l0,list* l1){

  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;

}
```

$t^\sharp$



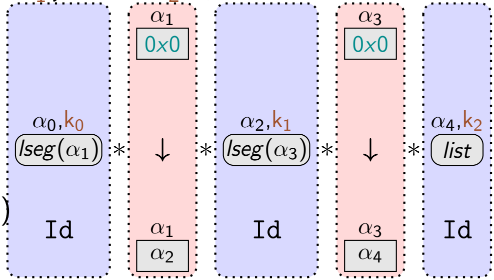
Modular analysis by composition of abstract transformations

```
double_append(list* k0,list* k1,list* k2){
  append(k0,k1);
  append(k0,k2);
}
```

$\mathtt{Id}(h_0^\sharp)$

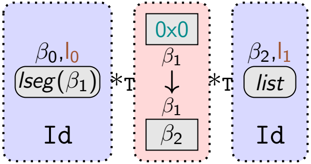
$t^\sharp \circ \mathtt{Id}(h_0^\sharp)$

$t^\sharp \circ t^\sharp \circ \mathtt{Id}(h_0^\sharp)$




```
append(list* l0,list* l1){
  while(l0→n≠0x0){l0=l0→n;}
  l0→n = l1;
}
```

$t^\sharp$



Contributions

Interprocedural transformation analysis using separation logic

1. Interprocedural analysis by composition of abstract transformations
2. Evaluation
3. Application to shape abstract transformations

Outline

1. Interprocedural analysis by composition of abstract transformations
2. Evaluation
3. Application to shape abstract transformations

A simple abstract state and transformation

Example (State abstraction $\mathbb{S}^\sharp$)

$\mathbb{S}^\sharp \triangleq$ "linear inequalities over program variables" [Cousot&Halbwachs 1978]

- $\gamma_{\mathbb{S}} : \mathbb{S}^\sharp \rightarrow \mathcal{P}(\mathbb{S})$

$$[\![\mathrm{z} := \mathrm{x} + 1]\!](\ \mathrm{x} < \mathrm{y}\ ) = \begin{cases} \mathrm{x} < \mathrm{y} \\ \mathrm{z} = \mathrm{x} + 1 \end{cases}$$

A simple abstract state and transformation

Example (State abstraction $\mathbb{S}^\sharp$)

$\mathbb{S}^\sharp \triangleq$ "linear inequalities over program variables" [Cousot&Halbwachs 1978]

- $\gamma_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathcal{P}(\mathbb{S})$

$$[\![z := x + 1]\!](\ x < y\ ) = \begin{cases} x < y \\ z = x + 1 \end{cases}$$

Example (Abstract transformation abstraction $\mathbb{T}^\sharp$)

$\mathbb{T}^\sharp \triangleq$ "linear inequalities over primed and unprimed program variables"

- $\gamma_{\mathbb{T}} : \mathbb{T}^\sharp \to \mathcal{P}(\mathbb{S} \times \mathbb{S})$

$$[\![z := x + 1]\!]\left(\begin{cases} x < y \\ x' = x \\ y' = y \\ z' = z \end{cases}\right) = \begin{cases} x < y \\ x' = x \\ y' = y \\ z' = x + 1 \end{cases}$$

Anatomy of an abstract transformation $t^\sharp \in \mathbb{T}^\sharp$

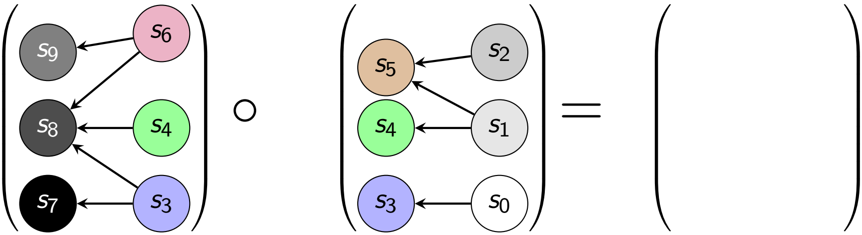
Let $t^\sharp = \begin{cases} \mathrm{x} < \mathrm{y} \\ \mathrm{x}' = \mathrm{x} \\ \mathrm{y}' = \mathrm{y} \\ \mathrm{z}' = \mathrm{x} + 1 \end{cases}$ $t^\sharp$ simultaneously contains:

1. A description $\in \mathbb{S}^\sharp$ of the *input states:* $\mathcal{I}(t^\sharp) = \mathrm{x} < \mathrm{y}$;
2. A description $\in \mathbb{S}^\sharp$ of the *output states:* $\mathcal{O}(t^\sharp) = \begin{cases} \mathrm{x} < \mathrm{y} \\ \mathrm{z} = \mathrm{x} + 1 \end{cases}$;
3. A description $\in \mathbb{T}^\sharp$ of the relation between the input and output:

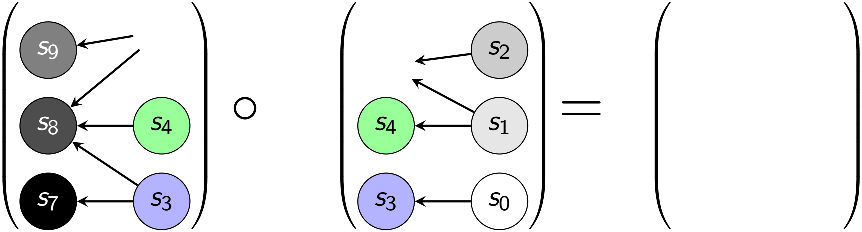
$$\begin{cases} \mathrm{x}' = \mathrm{x} \\ \mathrm{y}' = \mathrm{y} \\ \mathrm{z}' = \mathrm{x} + 1 \end{cases}$$

A relational abstraction $t^\sharp \in \mathbb{T}^\sharp$ is more precise than $(\mathcal{I}(t^\sharp), \mathcal{O}(t^\sharp))$, the pair of its pre and postcondition.

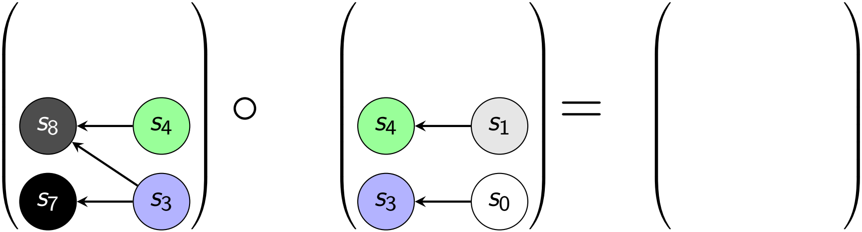
Composition of relations and abstract transformations



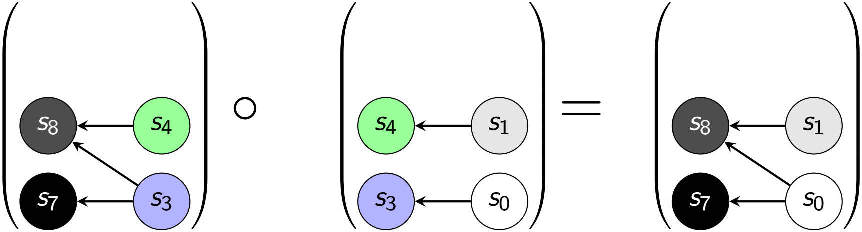
Composition of relations and abstract transformations



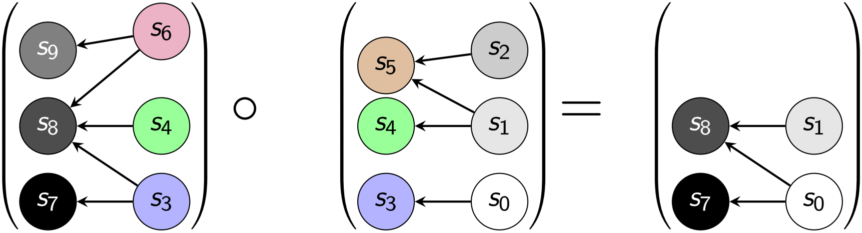
Composition of relations and abstract transformations



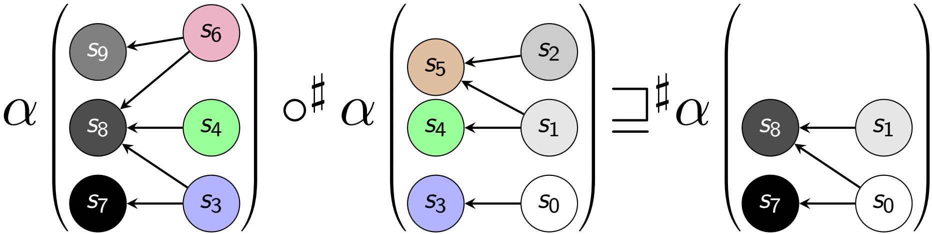
Composition of relations and abstract transformations



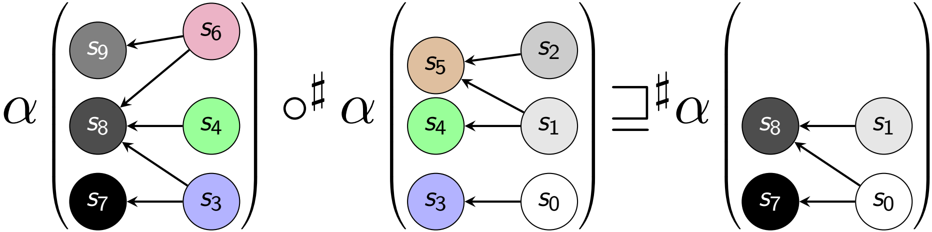
Composition of relations and abstract transformations



Composition of relations and abstract transformations



Composition of relations and abstract transformations



Soundness theorem for $\circ^\sharp$

$\circ^\sharp$ over-approximates the relational composition $\circ$: $\forall s_a, s_b, s_c \in \mathbb{S}$

$$(s_a, s_b) \in \gamma_{\mathbb{T}}(t_1^\sharp) \land (s_b, s_c) \in \gamma_{\mathbb{T}}(t_2^\sharp) \quad \Rightarrow \quad (s_a, s_c) \in \gamma_{\mathbb{T}}(t_2^\sharp \circ^\sharp t_1^\sharp)$$

Abstract composition to use procedure summaries

Function	Summary
`def f() { x := x + 1 }`	$t^\sharp_{\mathtt{f}} = \begin{cases} \mathtt{x}' = \mathtt{x} + 1 \\ \mathtt{y}' = \mathtt{y} \\ \mathtt{z}' = \mathtt{z} \end{cases}$
`def g() { y := x }`	$t^\sharp_{\mathtt{g}} = \begin{cases} \mathtt{x}' = \mathtt{x} \\ \mathtt{y}' = \mathtt{x} \\ \mathtt{z}' = \mathtt{z} \end{cases}$
`def h() { f(); g() }`	$t^\sharp_{\mathtt{h}} = t^\sharp_{\mathtt{g}} \circ^\sharp t^\sharp_{\mathtt{f}} = \begin{cases} \mathtt{x}' = \mathtt{x} + 1 \\ \mathtt{y}' = \mathtt{x} + 1 \\ \mathtt{z}' = \mathtt{z} \end{cases}$

Abstract composition is the operator to use an abstract transformation as a procedure summary.

Global vs context-specific transformation summaries (1)

Global transformation summary $t^\sharp_{\mathtt{f}}$: represents **all** the behaviours of f.

$$\forall s \in \mathbb{S} : (s, [\![\mathtt{f}]\!](s)) \in \gamma_{\mathbb{T}}(t^\sharp_{\mathtt{f}})$$

+ Allows purely bottom-up analysis of the program [Sharir&Pnueli 1981]

Global vs context-specific transformation summaries (1)

Global transformation summary $t^\sharp_{\mathtt{f}}$: represents **all** the behaviours of f.

$$\forall s \in \mathbb{S} : (s, [\![\mathtt{f}]\!](s)) \in \gamma_{\mathbb{T}}(t^\sharp_{\mathtt{f}})$$

+ Allows purely bottom-up analysis of the program [Sharir&Pnueli 1981]
− Leads to imprecisions of the abstract transformation:

Function	Computed global summary
`def f() { z := x * y }`	$t^\sharp_{\mathtt{f}} = \begin{cases} \mathrm{x}' = \mathrm{x} \\ \mathrm{y}' = \mathrm{y} \end{cases}$
`def g() { x := 3; f() }`	$t^\sharp_{\mathtt{g}} = \begin{cases} \mathrm{x}' = 3 \\ \mathrm{y}' = \mathrm{y} \end{cases}$

Global vs context-specific transformation summaries (2)

Context transformation summary $(s^\sharp_{\mathtt{f}}, t^\sharp_{\mathtt{f}})$: represents all the behaviours of f **for some precondition** $s^\sharp_{\mathtt{f}}$:

$$\forall s \in \gamma_{\mathbb{S}}(s^\sharp_{\mathtt{f}}) : (s, [\![\mathtt{f}]\!](s)) \in \gamma_{\mathbb{T}}(t^\sharp_{\mathtt{f}})$$

Global vs context-specific transformation summaries (2)

Context transformation summary $(s^\sharp_{\mathtt{f}}, t^\sharp_{\mathtt{f}})$: represents all the behaviours of f **for some precondition** $s^\sharp_{\mathtt{f}}$:

$$\forall s \in \gamma_{\mathbb{S}}(s^\sharp_{\mathtt{f}}) : (s, [\![\mathtt{f}]\!](s)) \in \gamma_{\mathbb{T}}(t^\sharp_{\mathtt{f}})$$

Function	Computed context summary
`def f() { z := x * y }`	$s^\sharp_{\mathtt{f}} = \{\mathtt{x} = 3\}, t^\sharp_{\mathtt{f}} = \begin{cases} \mathtt{x}' = \mathtt{x} \\ \mathtt{y}' = \mathtt{y} \\ \mathtt{z}' = 3 * \mathtt{y} \end{cases}$
`def g() { x := 3; f() }`	$s^\sharp_{\mathtt{g}} = \top, t^\sharp_{\mathtt{g}} = \begin{cases} \mathtt{x}' = 3 \\ \mathtt{y}' = \mathtt{y} \\ \mathtt{z}' = 3 * \mathtt{y} \end{cases}$

Context transformation summary $(s^\sharp_{\mathtt{f}}, t^\sharp_{\mathtt{f}})$: represents all the behaviours of f **for some precondition** $s^\sharp_{\mathtt{f}}$:

$$\forall s \in \gamma_{\mathbb{S}}(s^\sharp_{\mathtt{f}}) : (s, [\![\mathtt{f}]\!](s)) \in \gamma_{\mathbb{T}}(t^\sharp_{\mathtt{f}})$$

Function	Computed context summary
`def f() { z := x * y }`	$s^\sharp_{\mathtt{f}} = \{\mathtt{x} = 3\}, t^\sharp_{\mathtt{f}} = \begin{cases} \mathtt{x}' = \mathtt{x} \\ \mathtt{y}' = \mathtt{y} \\ \mathtt{z}' = 3 * \mathtt{y} \end{cases}$
`def g() { x := 3; f() }`	$s^\sharp_{\mathtt{g}} = \top, t^\sharp_{\mathtt{g}} = \begin{cases} \mathtt{x}' = 3 \\ \mathtt{y}' = \mathtt{y} \\ \mathtt{z}' = 3 * \mathtt{y} \end{cases}$

→ Requires a top-down algorithm:

- Reuse summary if possible
- Recompute summary with a larger calling context if needed.

We need both a context and a transformation

When $\texttt{f}$ is simple, then $s^\sharp_{\texttt{f}} = \mathcal{I}(t^\sharp_{\texttt{f}})$, and $s^\sharp_{\texttt{f}}$ seems redundant.

We need both a context and a transformation

When f is simple, then $s^\sharp_{\mathtt{f}} = \mathcal{I}(t^\sharp_{\mathtt{f}})$, and $s^\sharp_{\mathtt{f}}$ seems redundant. But:

```
def f()
{ if(x > y)
    while(1);
  else if(x < y)
    z = 1 /0;
}
```

$$s^\sharp_{\mathtt{f}} = \top, t^\sharp_{\mathtt{f}} = \begin{cases} x = y \\ \mathtt{x}' = \mathtt{x} \\ \mathtt{y}' = \mathtt{y} \\ \mathtt{z}' = \mathtt{z} \end{cases}, \mathcal{I}(t^\sharp_{\mathtt{f}}) = \{x = y\}$$

- $s^\sharp_{\mathtt{f}}$: context where the summary can be applied;
- $t^\sharp_{\mathtt{f}}$: summary to apply;
- $\mathcal{I}(t^\sharp_{\mathtt{f}})$: inferred necessary pre-condition on states that return from f.

Algorithm idea

Top down, hybrid inter/intra procedural algorithm:

- Simple statements: use relational abstract transformers

$$t'^{\sharp} = [\![x := x + 1]\!]^{\sharp}(t^{\sharp})$$

- Function call to `f`:
  1. Determine if the current context transformation summary can be used

$$\mathcal{O}(t^{\sharp}) \sqsubseteq^{\sharp}_{\mathbb{S}} s^{\sharp}_{\mathtt{f}}$$

  2. Recompute the summary with a larger context if needed.

$$\text{new } s^{\sharp}_{\mathtt{f}} = \text{previous } s^{\sharp}_{\mathtt{f}} \sqcup_{\mathbb{S}} \mathcal{O}(t^{\sharp})$$

$$\mathcal{O}(t^{\sharp})))\text{new } t^{\sharp}_{\mathtt{f}} = [\![\text{body of } \mathtt{f}]\!]^{\sharp}(Id(\text{new } s^{\sharp}_{\mathtt{f}}))$$

  3. Abstract composition to use the summary of `f`:

$$t'^{\sharp} = [\![f()]\!]^{\sharp}(t^{\sharp}) = t^{\sharp}_{\mathtt{f}} \circ^{\sharp} t^{\sharp}$$

- If recursion: grow context of procedure summaries until fixpoint.

Towards relational separation logic (1) : Frame rule

`def f() { x := x + 1 }`	$s^\sharp_\mathrm{f} = \{y = 3\}, t^\sharp_\mathrm{f} = \begin{cases} y = 3 \\ \mathrm{x}' = \mathrm{x} + 1 \\ \mathrm{y}' = \mathrm{y} \\ \mathrm{z}' = \mathrm{z} \end{cases}$

`def g() {`
`  y := 3; f();`
`  y := 4; f();`
`}`
	$s^\sharp_\mathrm{g} = \top, t^\sharp_\mathrm{g} = \quad [\![f()]\!] \quad \left(\begin{cases} \mathrm{x}' = \mathrm{x} + 1 \\ \mathrm{y}' = 4 \\ \mathrm{z}' = \mathrm{z} \end{cases} \right)$

- Irrelevant memory regions in the context $\Rightarrow$ spurious summary recomputations.

Towards relational separation logic (1) : Frame rule

`def f() { x := x + 1 }`	$s^\sharp_{\mathrm{f}} = \{y = 3\}, t^\sharp_{\mathrm{f}} = \begin{cases} y = 3 \\ \mathrm{x}' = \mathrm{x} + 1 \\ \mathrm{y}' = \mathrm{y} \\ \mathrm{z}' = \mathrm{z} \end{cases}$

`def g() {`
`  y := 3; f();`
`  y := 4; f();`
`}`
	$s^\sharp_{\mathrm{g}} = \top, t^\sharp_{\mathrm{g}} = \quad [\![f()]\!] \quad \left(\begin{cases} \mathrm{x}' = \mathrm{x} + 1 \\ \mathrm{y}' = 4 \\ \mathrm{z}' = \mathrm{z} \end{cases} \right)$

- Irrelevant memory regions in the context $\Rightarrow$ spurious summary recomputations.

➜ Frame Rule of (relational) separation logic

Towards relational separation logic (1) : Frame rule

`def f() { x := x + 1 }`	$s^\sharp_{\mathrm{f}} = \top \quad , t^\sharp_{\mathrm{f}} = \left\{ \begin{array}{l} \mathrm{x}' = \mathrm{x} + 1 \\ \\ \end{array} \right.$

`def g() {`
`  y := 3; f();`
`  y := 4; f();`
`}`
	$s^\sharp_{\mathrm{g}} = \top, t^\sharp_{\mathrm{g}} = \quad [\![f()]\!] \quad \left(\left\{ \begin{array}{l} \mathrm{x}' = \mathrm{x} + 1 \\ \mathrm{y}' = 4 \\ \mathrm{z}' = \mathrm{z} \end{array} \right. \right)$

- Irrelevant memory regions in the context $\Rightarrow$ spurious summary recomputations.

➜ Frame Rule of (relational) separation logic

Towards relational separation logic (1) : Frame rule

`def f() { x := x + 1 }`	$s^\sharp_{\mathtt{f}} = \top \quad , t^\sharp_{\mathtt{f}} = \left\{ \begin{array}{l} \mathtt{x}' = \mathtt{x} + 1 \\ \\ \end{array} \right.$

`def g() {`
`  y := 3; f();`
`  y := 4; f();`
`}`
	$s^\sharp_{\mathtt{g}} = \top, t^\sharp_{\mathtt{g}} = \left(t^\sharp_{\mathtt{f}} *_{\mathrm{T}} \left\{ \begin{array}{l} \mathtt{y}' = y \\ \mathtt{z}' = \mathtt{z} \end{array} \right. \right) \circ^\sharp \left(\left\{ \begin{array}{l} \mathtt{x}' = \mathtt{x} + 1 \\ \mathtt{y}' = 4 \\ \mathtt{z}' = \mathtt{z} \end{array} \right. \right)$

- Irrelevant memory regions in the context $\Rightarrow$ spurious summary recomputations.
- → Frame Rule of (relational) separation logic

Towards relational separation logic (2): unbounded memory

How to handle more complex and unbounded memory states?

```
def f() { z := x + 1 }
```

$$s^\sharp_{\mathtt{f}} = \{x < y\}, t^\sharp_{\mathtt{f}} = \begin{cases} x < y \\ \mathtt{x}' = \mathtt{x} \\ \mathtt{y}' = \mathtt{y} \\ \mathtt{z}' = \mathtt{x} + 1 \end{cases}$$

1. Separate memory descriptions and use a shared numerical abstraction

$$t^\sharp_{\mathtt{f}} = \left(\begin{bmatrix} \mathtt{x} \mapsto \alpha_{\mathtt{x}} \\ \mathtt{y} \mapsto \alpha_{\mathtt{y}} \\ \mathtt{z} \mapsto \alpha_{\mathtt{z}} \end{bmatrix} \dashrightarrow \begin{bmatrix} \mathtt{x} \mapsto \alpha_{\mathtt{x}} \\ \mathtt{y} \mapsto \alpha_{\mathtt{y}} \\ \mathtt{z} \mapsto \alpha'_{\mathtt{z}} \end{bmatrix} \right) \wedge \begin{cases} \alpha_{\mathtt{x}} < \alpha_{\mathtt{y}} \\ \alpha'_{\mathtt{z}} = \alpha_{\mathtt{x}} + 1 \end{cases}$$

Towards relational separation logic (2): unbounded memory

How to handle more complex and unbounded memory states?

```
def f() { z := x + 1 }
```

$$s^\sharp_{\mathtt{f}} = \{x < y\},\ t^\sharp_{\mathtt{f}} = \begin{cases} x < y \\ \mathtt{x}' = \mathtt{x} \\ \mathtt{y}' = \mathtt{y} \\ \mathtt{z}' = \mathtt{x} + 1 \end{cases}$$

1. Separate memory descriptions and use a shared numerical abstraction

$$t^\sharp_{\mathtt{f}} = \left(\begin{bmatrix} \mathtt{x} \mapsto \alpha_{\mathtt{x}} \\ \mathtt{y} \mapsto \alpha_{\mathtt{y}} \\ \mathtt{z} \mapsto \alpha_{\mathtt{z}} \end{bmatrix} \dashrightarrow \begin{bmatrix} \mathtt{x} \mapsto \alpha_{\mathtt{x}} \\ \mathtt{y} \mapsto \alpha_{\mathtt{y}} \\ \mathtt{z} \mapsto \alpha'_{\mathtt{z}} \end{bmatrix} \right) \wedge \begin{cases} \alpha_{\mathtt{x}} < \alpha_{\mathtt{y}} \\ \alpha'_{\mathtt{z}} = \alpha_{\mathtt{x}} + 1 \end{cases}$$

1. Introduce `Id` predicate to represent equal regions without enumerating values

$$t^\sharp_{\mathtt{f}} = \mathtt{Id} \left(\begin{bmatrix} \mathtt{x} \mapsto \alpha_{\mathtt{x}} \\ \mathtt{y} \mapsto \alpha_{\mathtt{y}} \end{bmatrix} \right) *_{\mathrm{T}} \left(\begin{bmatrix} \mathtt{z} \mapsto \alpha_{\mathtt{z}} \end{bmatrix} \dashrightarrow \begin{bmatrix} \mathtt{z} \mapsto \alpha'_{\mathtt{z}} \end{bmatrix} \right) \wedge \begin{cases} \alpha_{\mathtt{x}} < \alpha_{\mathtt{y}} \\ \alpha'_{\mathtt{z}} = \alpha_{\mathtt{x}} + 1 \end{cases}$$

How to handle more complex and unbounded memory states?

```
def f() { z := x + 1 }
```

$$s^\sharp_{\mathtt{f}} = \{x < y\},\ t^\sharp_{\mathtt{f}} = \begin{cases} x < y \\ \mathtt{x}' = \mathtt{x} \\ \mathtt{y}' = \mathtt{y} \\ \mathtt{z}' = \mathtt{x} + 1 \end{cases}$$

1. Separate memory descriptions and use a shared numerical abstraction

$$t^\sharp_{\mathtt{f}} = \left(\begin{bmatrix} \mathtt{x} \mapsto \alpha_{\mathtt{x}} \\ \mathtt{y} \mapsto \alpha_{\mathtt{y}} \\ \mathtt{z} \mapsto \alpha_{\mathtt{z}} \end{bmatrix} \dashrightarrow \begin{bmatrix} \mathtt{x} \mapsto \alpha_{\mathtt{x}} \\ \mathtt{y} \mapsto \alpha_{\mathtt{y}} \\ \mathtt{z} \mapsto \alpha'_{\mathtt{z}} \end{bmatrix} \right) \wedge \begin{cases} \alpha_{\mathtt{x}} < \alpha_{\mathtt{y}} \\ \alpha'_{\mathtt{z}} = \alpha_{\mathtt{x}} + 1 \end{cases}$$

2. Introduce `Id` predicate to represent equal regions without enumerating values

$$t^\sharp_{\mathtt{f}} = \mathtt{Id} \left(\begin{bmatrix} \mathtt{x} \mapsto \alpha_{\mathtt{x}} \\ \mathtt{y} \mapsto \alpha_{\mathtt{y}} \end{bmatrix} \right) *_{\mathtt{T}} \left(\begin{bmatrix} \mathtt{z} \mapsto \alpha_{\mathtt{z}} \end{bmatrix} \dashrightarrow \begin{bmatrix} \mathtt{z} \mapsto \alpha'_{\mathtt{z}} \end{bmatrix} \right) \wedge \begin{cases} \alpha_{\mathtt{x}} < \alpha_{\mathtt{y}} \\ \alpha'_{\mathtt{z}} = \alpha_{\mathtt{x}} + 1 \end{cases}$$

3. Generalize to arbitrary representations of heap (shape analysis)

$$t^\sharp_{\mathtt{f}} = \mathtt{Id}\left(h^\sharp_{xy}\right) *_{\mathtt{T}} \left(h^\sharp_z \dashrightarrow h'^\sharp_z\right) \wedge \begin{cases} \alpha_{\mathtt{x}} < \alpha_{\mathtt{y}} \\ \alpha'_{\mathtt{z}} = \alpha_{\mathtt{x}} + \end{cases}$$

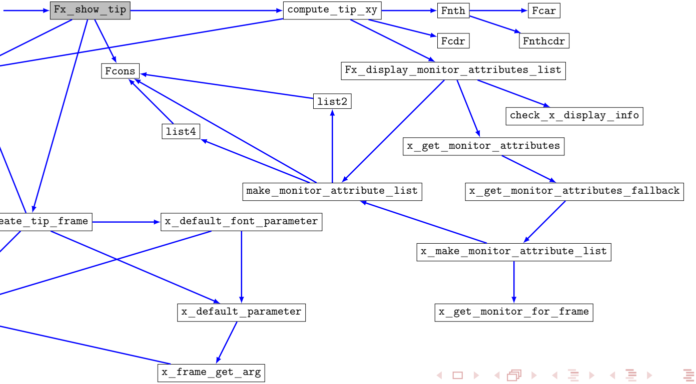
Outline

Experimental evaluation: Static call graph

- Analysers implementation as a Frama-C plugin
  - Three modes:
    - call-string state analysis
    - call-string relational analysis
    - summary-based relational analysis

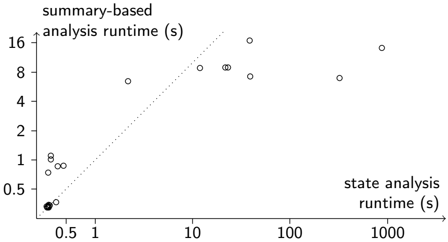
Experimental evaluation: Static call graph

- Analysers implementation as a Frama-C plugin
  - Three modes:
    - call-string state analysis
    - call-string relational analysis
    - summary-based relational analysis
- Main use case: part (2,000 lines of C) of Emacs
  - Heavy manipulation of pairs, used as untyped lists



Experimental evaluation: Results

	Time (in s)	
	state	relational
inline	877	4257
summary-based	-	15



- Summary-based analysis is **much faster** on all functions but leaves:
  - Gain of **58x** compared to the state analysis
  - Gain of **284x** compared to the relational analysis with inlining
- Most reanalyzed function: `Fcons` (reanalyzed 3 times, used 47 times)
- **No observed loss of precision** wrt. state and relational analysis
- Inferred relational properties stronger than state properties

Summary and conclusions

- Contextual procedure summaries can be represented as an abstract transformation with a context
- Summary-based transformation analyses can be done by composing abstract transformations
- Can be applied to memory analysis using separation logic

Summary and conclusions

- Contextual procedure summaries can be represented as an abstract transformation with a context
- Summary-based transformation analyses can be done by composing abstract transformations
- Can be applied to memory analysis using separation logic

Transformations are harder to abstract than states
but using them can be very rewarding

Transformations are:

- A basis for **compact** and **precise** function summaries
- Capture a **natural abstraction** for programmers
- Can verify **useful functional properties**

Outline

From Separation Logic to **Relational** Separation Logic

Separation Logic

$$
\begin{array}{rcl}
h^\sharp & ::= & \mathsf{emp} \\
& | & \alpha \mapsto \beta \\
& | & \mathsf{list}(\alpha) \\
& | & \mathsf{lseg}(\alpha) \\
& | & h^\sharp * h^\sharp
\end{array}
$$

- Separation Logic: **properties on states**

A New Abstract Domain of Relations: Abstract Transformations

Relational Separation Logic

Separation Logic

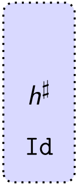
$$
\begin{array}{rcl}
h^\sharp & ::= & \mathsf{emp} \\
& | & \alpha \mapsto \beta \\
& | & \mathsf{list}(\alpha) \\
& | & \mathsf{lseg}(\alpha) \\
& | & h^\sharp * h^\sharp
\end{array}
$$

$$
\begin{array}{rcl}
t^\sharp & ::= & \mathtt{Id}(h^\sharp) \\
& | & [h^\sharp \dashrightarrow h^\sharp] \\
& | & t^\sharp *_{\mathtt{T}} t^\sharp
\end{array}
$$

- Separation Logic: **properties on states**
- **Relational** Separation Logic: **properties on pairs of states** (in, out)

Relational Separation Logic Connectives

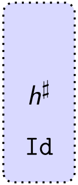
$\mathtt{Id}(h^\sharp)$: **No modification**



Relational Separation Logic Connectives

$\mathtt{Id}(h^\sharp)$: **No modification**

$\gamma_{\mathbb{T}}(Id(h^\sharp)) = \{(\sigma, \sigma) : \sigma \in \gamma_{\mathbb{H}}(h^\sharp)\}$



Relational Separation Logic Connectives

$\mathtt{Id}(h^\sharp)$: **No modification**

$\gamma_{\mathbb{T}}(Id(h^\sharp)) = \{(\sigma, \sigma) : \sigma \in \gamma_{\mathbb{H}}(h^\sharp)\}$

$[h_i^\sharp \dashrightarrow h_o^\sharp]$: **Memory transformation**

$h^\sharp$

$\mathtt{Id}$

$h_i^\sharp$

$\downarrow$

$h_o^\sharp$

Relational Separation Logic Connectives

$\mathtt{Id}(h^\sharp)$: **No modification**

$$\gamma_{\mathbb{T}}(Id(h^\sharp)) = \{(\sigma, \sigma) : \sigma \in \gamma_{\mathbb{H}}(h^\sharp)\}$$

$[h_i^\sharp \dashrightarrow h_o^\sharp]$: **Memory transformation**

$$\gamma_{\mathbb{T}}([h_i^\sharp \dashrightarrow h_o^\sharp]) = \left\{ \begin{array}{ccl} (\sigma_i, \sigma_o) & : & \sigma_i \in \gamma_{\mathbb{H}}(h_i^\sharp) \\ & \wedge & \sigma_o \in \gamma_{\mathbb{H}}(h_o^\sharp) \end{array} \right\}$$

$h^\sharp$

$\mathtt{Id}$

$$h_i^\sharp \downarrow h_o^\sharp$$

Relational Separation Logic Connectives

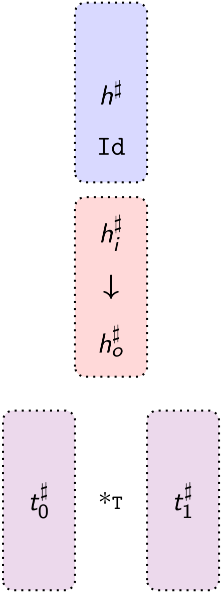
$\texttt{Id}(h^\sharp)$: **No modification**

$$\gamma_{\mathbb{T}}(Id(h^\sharp)) = \{(\sigma, \sigma) : \sigma \in \gamma_{\mathbb{H}}(h^\sharp)\}$$

$[h_i^\sharp \dashrightarrow h_o^\sharp]$: **Memory transformation**

$$\gamma_{\mathbb{T}}([h_i^\sharp \dashrightarrow h_o^\sharp]) = \left\{ \begin{array}{lll} (\sigma_i, \sigma_o) & : & \sigma_i \in \gamma_{\mathbb{H}}(h_i^\sharp) \\ & \wedge & \sigma_o \in \gamma_{\mathbb{H}}(h_o^\sharp) \end{array} \right\}$$

$t_0^\sharp *_{\mathtt{T}} t_1^\sharp$: **Independent transformations**



Relational Separation Logic Connectives

$\texttt{Id}(h^\sharp)$: **No modification**

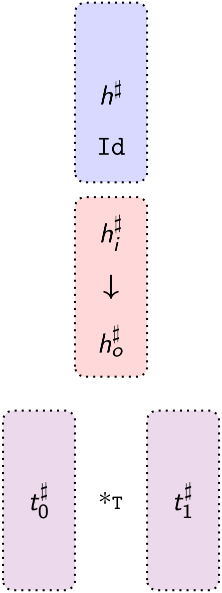
$$\gamma_{\mathbb{T}}(Id(h^\sharp)) = \{(\sigma, \sigma) : \sigma \in \gamma_{\mathbb{H}}(h^\sharp)\}$$

$[h_i^\sharp \dashrightarrow h_o^\sharp]$: **Memory transformation**

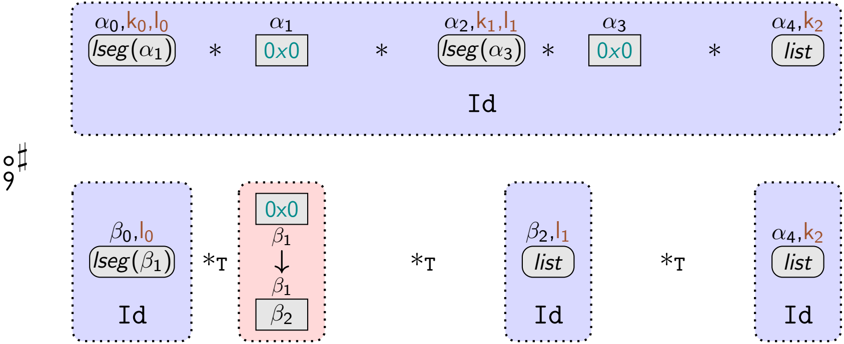
$$\gamma_{\mathbb{T}}([h_i^\sharp \dashrightarrow h_o^\sharp]) = \left\{ \begin{array}{lcl} (\sigma_i, \sigma_o) & : & \sigma_i \in \gamma_{\mathbb{H}}(h_i^\sharp) \\ & \wedge & \sigma_o \in \gamma_{\mathbb{H}}(h_o^\sharp) \end{array} \right\}$$

$t_0^\sharp *_{\mathtt{T}} t_1^\sharp$: **Independent transformations**

$$\gamma_{\mathbb{T}}(t_0^\sharp *_{\mathtt{T}} t_1^\sharp) = \left\{ \begin{array}{ll} \multicolumn{2}{l}{(\sigma_{i,0} \uplus \sigma_{i,1}, \sigma_{o,0} \uplus \sigma_{o,1}) :} \\ & (\sigma_{i,0}, \sigma_{o,0}) \in \gamma_{\mathbb{T}}(t_0^\sharp) \\ \wedge & (\sigma_{i,1}, \sigma_{o,1}) \in \gamma_{\mathbb{T}}(t_1^\sharp) \\ \wedge & \text{separation conditions} \end{array} \right\}$$

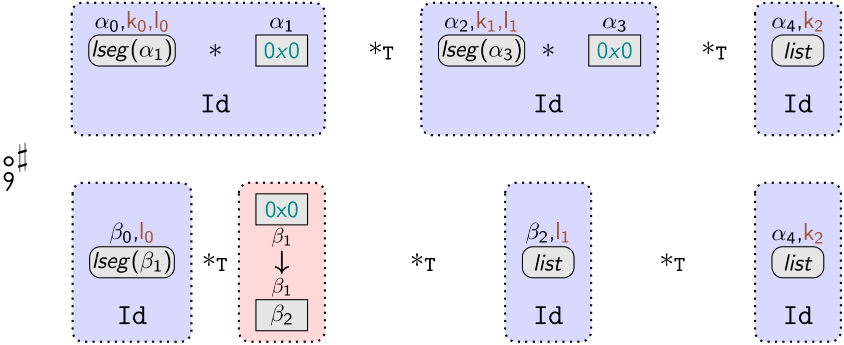


Abstract transformation composition: step-by-step example



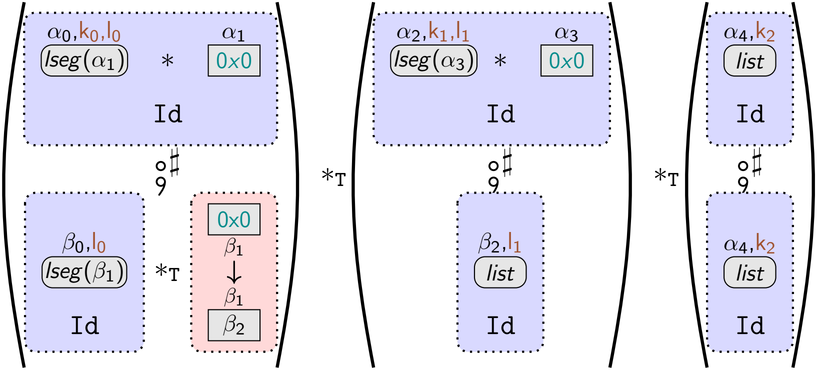
- Step-by-step composition on the first `append` call

Abstract transformation composition: step-by-step example



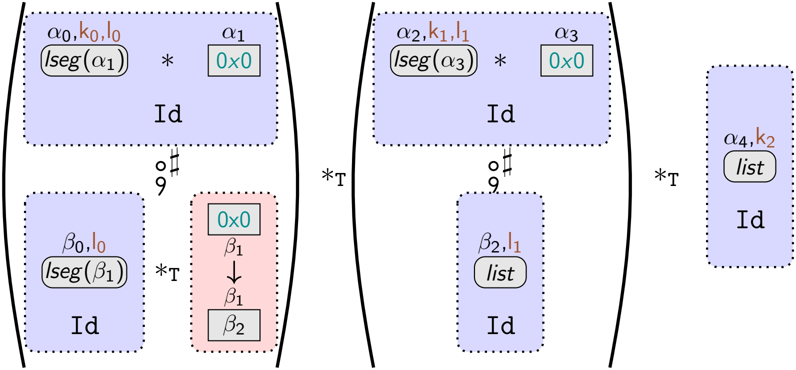
- $Id(h_1) *_T Id(h_2) \Leftrightarrow Id(h_1 * h_2)$

Abstract transformation composition: step-by-step example



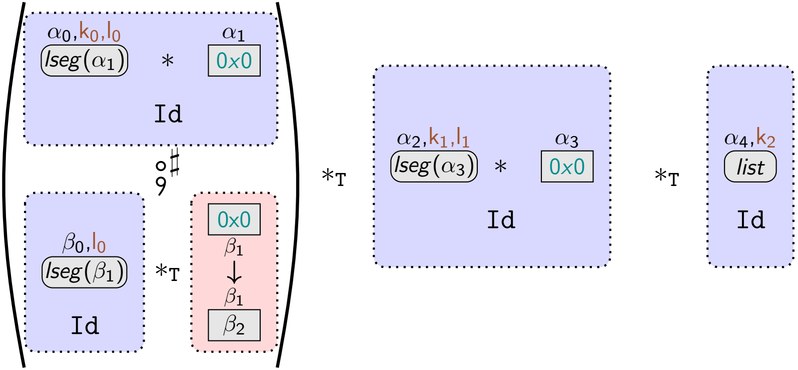
- Local composition

Abstract transformation composition: step-by-step example



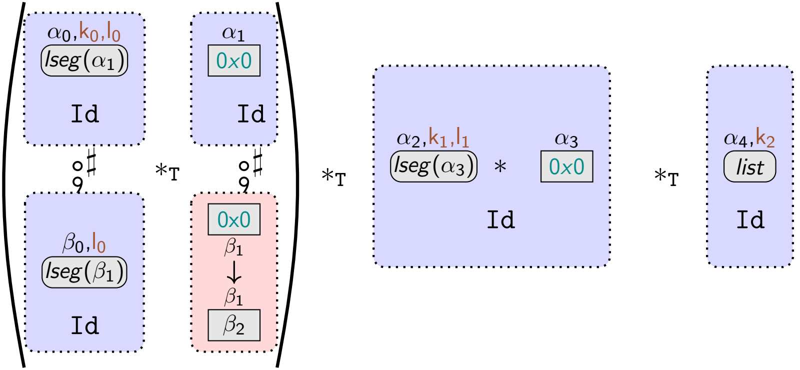
- $Id(h_1) \fatsemi^\sharp Id(h_2) = Id(h_1 \sqcap h_2)$

Abstract transformation composition: step-by-step example



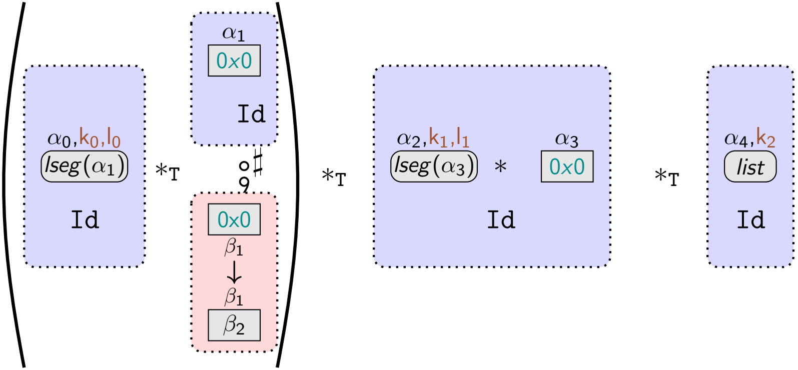
- $Id(h_1) \mathbin{⨟^\sharp} Id(h_2) = Id(h_1 \sqcap h_2)$

Abstract transformation composition: step-by-step example



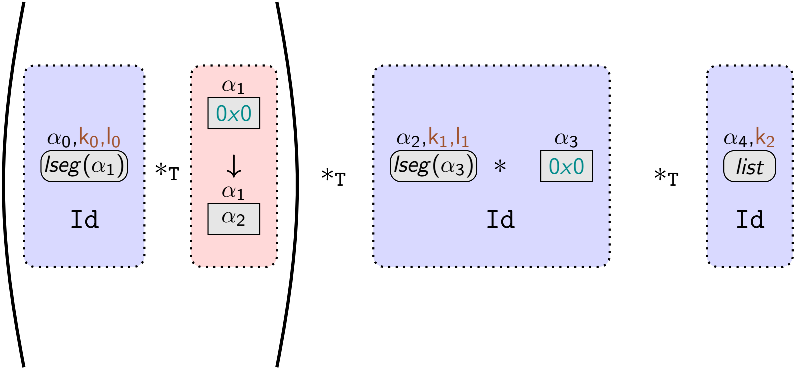
- $Id(h_1) *_{\mathtt{T}} Id(h_2) \Leftrightarrow Id(h_1 * h_2)$

Abstract transformation composition: step-by-step example



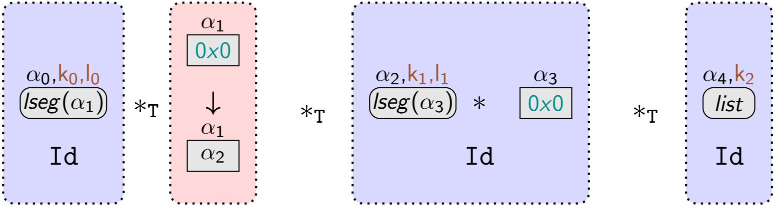
- $Id(h_1) \fatsemi^{\sharp} Id(h_2) = Id(h_1 \sqcap h_2)$

Abstract transformation composition: step-by-step example



- $Id(h_1) \mathbin{⨟}^\sharp [h_2 \longrightarrow h_3] = [h_1 \sqcap h_2 \longrightarrow h_3]$

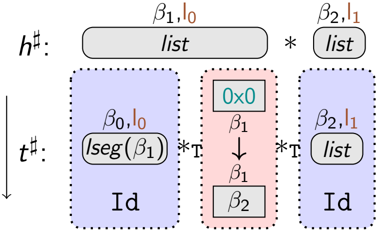
Abstract transformation composition: step-by-step example



```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow t_0^\sharp \qquad t_0^\sharp = \underset{\alpha_0, k_0}{lseg(\alpha_1)} * \underset{\alpha_1}{\boxed{0x0}} * \underset{\alpha_2, k_1}{lseg(\alpha_3)} * \underset{\alpha_3}{\boxed{0x0}} * \underset{\alpha_4, k_2}{list}$ Id

```
  append(k0,k1);
```



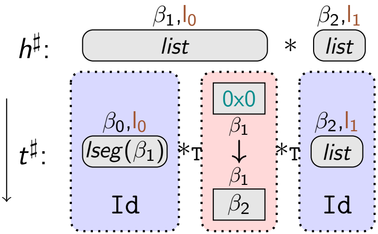
}

Using the summary when available

```
double_append(list* k0,list* k1,list* k2){
                                  ↓ t0#
  append(k0,k1);
```

$$t_0^\sharp = \underbrace{\overset{\alpha_0,k_0,l_0}{\boxed{lseg(\alpha_1)}} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{\boxed{lseg(\alpha_3)}} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,k_2}{\boxed{list}}}_{\mathtt{Id}}$$

1. Parameter passing



}

Using the summary when available

```
double_append(list* k0,list* k1,list* k2){
```

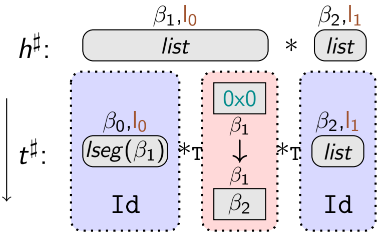
$\downarrow t_0^\sharp$

$$t_0^\sharp = \underbrace{\overset{\alpha_0,\mathrm{k}_0,\mathrm{l}_0}{\boxed{lseg(\alpha_1)}} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,\mathrm{k}_1,\mathrm{l}_1}{\boxed{lseg(\alpha_3)}} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,\mathrm{k}_2}{\boxed{list}}}_{\mathtt{Id}}$$

```
  append(k0,k1);
```

$$\mathcal{O}(t_0^\sharp) = \overset{\alpha_0,\mathrm{k}_0,\mathrm{l}_0}{\boxed{lseg(\alpha_1)}} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,\mathrm{k}_1,\mathrm{l}_1}{\boxed{lseg(\alpha_3)}} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,\mathrm{k}_2}{\boxed{list}}$$

1. Parameter passing
2. Extracting output state



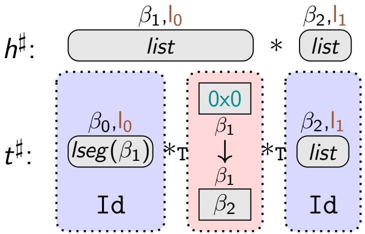
Using the summary when available

```
double_append(list* k0,list* k1,list* k2){
                    ↓ t0♯
  append(k0,k1);
```

$$t_0^\sharp = \underbrace{\overset{\alpha_0,k_0,l_0}{\mathit{lseg}(\alpha_1)} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{\mathit{lseg}(\alpha_3)} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,k_2}{\mathit{list}}}_{\mathtt{Id}}$$

$$\mathcal{O}(t_0^\sharp) = \underbrace{\overset{\alpha_0,k_0,l_0}{\mathit{lseg}(\alpha_1)} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{\mathit{lseg}(\alpha_3)} * \overset{\alpha_3}{\boxed{0x0}}}_{\text{Reachable part } \mathcal{R}[l_0,\, l_1](\mathcal{O}(t_0^\sharp))} * \underbrace{\overset{\alpha_4,k_2}{\mathit{list}}}_{\text{Not reachable}}$$

1. Parameter passing
2. Extracting output state
3. Procedure footprint



}

Using the summary when available

```
double_append(list* k0,list* k1,list* k2){
  append(k0,k1);
}
```

$\downarrow t_0^\sharp$

$$t_0^\sharp = \underbrace{\overset{\alpha_0,k_0,l_0}{lseg(\alpha_1)} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{lseg(\alpha_3)} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,k_2}{list}}_{\mathtt{Id}}$$

$$\mathcal{O}(t_0^\sharp) = \underbrace{\overset{\alpha_0,k_0,l_0}{lseg(\alpha_1)} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{lseg(\alpha_3)} * \overset{\alpha_3}{\boxed{0x0}}}_{\text{Reachable part } \mathcal{R}[l_0,l_1](\mathcal{O}(t_0^\sharp))} * \underbrace{\overset{\alpha_4,k_2}{list}}_{\text{Not reachable}}$$

$$\mathcal{R}[l_0,l_1](\mathcal{O}(t_0^\sharp)) \sqsubseteq h^\sharp \ ?$$

$$h^\sharp: \overset{\beta_1,l_0}{list} * \overset{\beta_2,l_1}{list}$$

$$t^\sharp: \underbrace{\overset{\beta_0,l_0}{lseg(\beta_1)}}_{\mathtt{Id}} *_{\mathtt{T}} \begin{array}{c}\boxed{0x0} \\ \beta_1 \\ \downarrow \\ \beta_1 \\ \boxed{\beta_2}\end{array} *_{\mathtt{T}} \underbrace{\overset{\beta_2,l_1}{list}}_{\mathtt{Id}}$$

1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

```
double_append(list* k0,list* k1,list* k2){
```

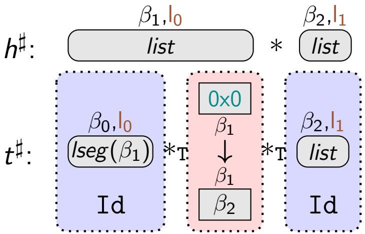
$\downarrow t_0^\sharp$

```
  append(k0,k1);
```

$$t_0^\sharp = \underbrace{\overset{\alpha_0,k_0,l_0}{\mathit{lseg}(\alpha_1)} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{\mathit{lseg}(\alpha_3)} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,k_2}{\mathit{list}}}_{\mathtt{Id}}$$

$$\mathcal{O}(t_0^\sharp) = \underbrace{\overset{\alpha_0,k_0,l_0}{\mathit{lseg}(\alpha_1)} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{\mathit{lseg}(\alpha_3)} * \overset{\alpha_3}{\boxed{0x0}}}_{\text{Reachable part } \mathcal{R}[l_0,l_1](\mathcal{O}(t_0^\sharp))} * \underbrace{\overset{\alpha_4,k_2}{\mathit{list}}}_{\text{Not reachable}}$$

$\mathcal{R}[l_0, l_1](\mathcal{O}(t_0^\sharp)) \sqsubseteq h^\sharp$? Yes



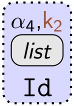
1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

}

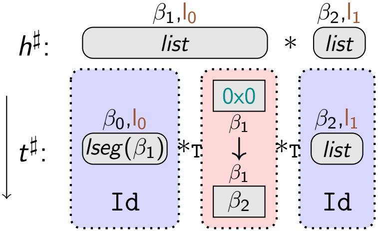
Using the summary when available

```
double_append(list* k0,list* k1,list* k2){
                                  ↓ t0#
  append(k0,k1);
}
```

$t_0^\sharp = \underbrace{\boxed{lseg(\alpha_1)}_{\alpha_0, k_0, l_0} * \boxed{0x0}_{\alpha_1} * \boxed{lseg(\alpha_3)}_{\alpha_2, k_1, l_1} * \boxed{0x0}_{\alpha_3} * \boxed{list}_{\alpha_4, k_2}}_{\mathtt{Id}}$



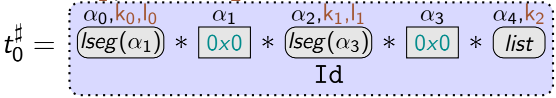
1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing
5. Summary application



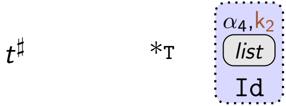
Using the summary when available

```
double_append(list* k0,list* k1,list* k2){
```

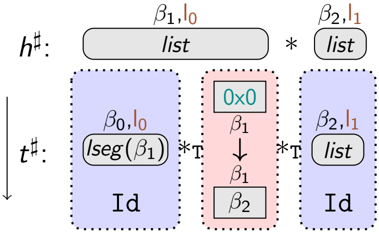
$\downarrow t_0^\sharp$




```
  append(k0,k1);
```



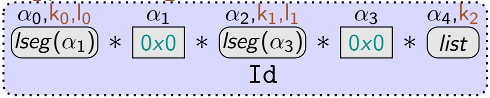
1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing
5. Summary application



}

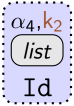
```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow t_0^\sharp$ $\quad t_0^\sharp =$

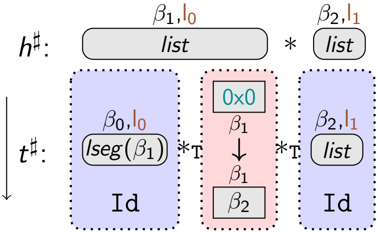



```
  append(k0,k1);
```

$\circ^\sharp_9 \quad t^\sharp \quad *_{\mathtt{T}}$



1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing
5. Summary application

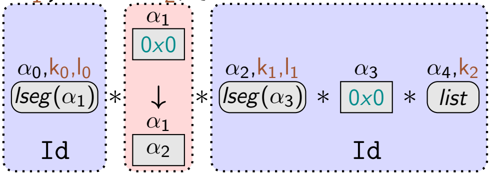


Using the summary when available

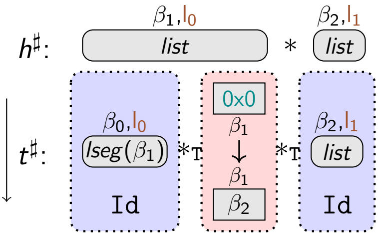
```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow t_0^\sharp$

```
  append(k0,k1);
```

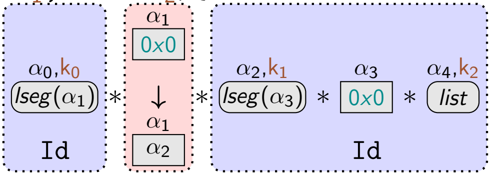


1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing
5. Summary application

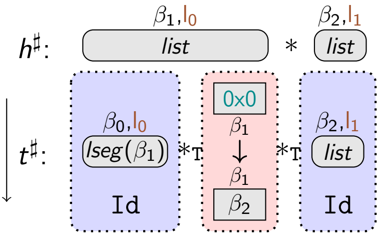


Using the summary when available

```
double_append(list* k0,list* k1,list* k2){

        ↓ t0♯

    append(k0,k1);
}
```



1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing
5. Summary application
6. Parameter suppression

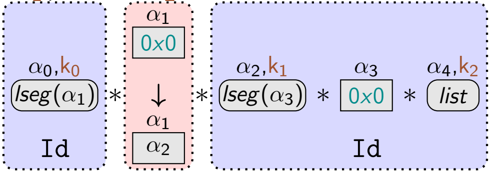


Using the summary when available

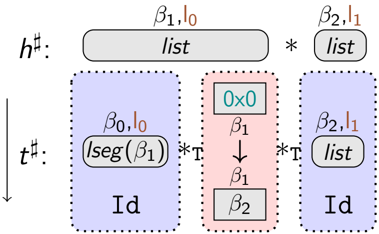
```
double_append(list* k0,list* k1,list* k2){
    append(k0,k1);
```

$t_0^\sharp$

$t_1^\sharp$



1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing
5. Summary application
6. Parameter suppression



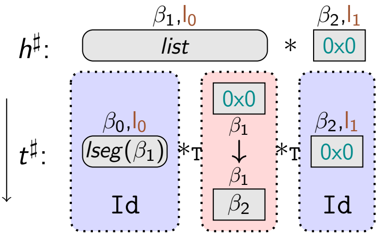
}

Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow t_0^\sharp \qquad t_0^\sharp = \underbrace{\overset{\alpha_0,k_0}{\mathit{lseg}(\alpha_1)} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1}{\mathit{lseg}(\alpha_3)} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,k_2}{\mathit{list}}}_{\mathtt{Id}}$

```
  append(k0,k1);
```



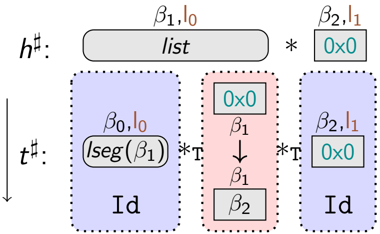
Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow t_0^\sharp \qquad t_0^\sharp = \overbrace{\underset{\alpha_0, k_0, l_0}{lseg(\alpha_1)}}^{} * \underset{\alpha_1}{\boxed{0x0}} * \underset{\alpha_2, k_1, l_1}{lseg(\alpha_3)} * \underset{\alpha_3}{\boxed{0x0}} * \underset{\alpha_4, k_2}{list}$ Id

```
  append(k0,k1);
```

1. Parameter passing



Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
```

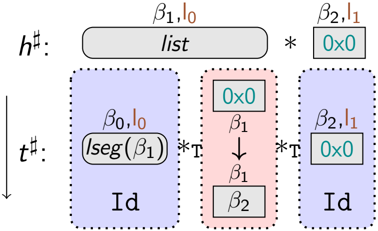
$\downarrow t_0^\sharp$

$$t_0^\sharp = \underbrace{\overset{\alpha_0,k_0,l_0}{\mathit{lseg}(\alpha_1)} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{\mathit{lseg}(\alpha_3)} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,k_2}{\mathit{list}}}_{\mathtt{Id}}$$

```
append(k0,k1);
```

$$\mathcal{O}(t_0^\sharp) = \overset{\alpha_0,k_0,l_0}{\mathit{lseg}(\alpha_1)} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{\mathit{lseg}(\alpha_3)} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,k_2}{\mathit{list}}$$

1. Parameter passing
2. Extracting output state



Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
```

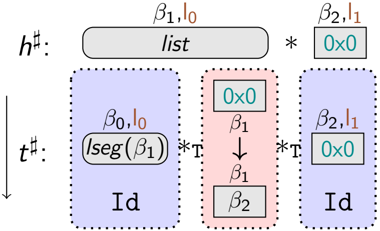
$\downarrow t_0^\sharp$

$$t_0^\sharp = \underbrace{\boxed{lseg(\alpha_1)}^{\alpha_0,k_0,l_0} * \boxed{0x0}^{\alpha_1} * \boxed{lseg(\alpha_3)}^{\alpha_2,k_1,l_1} * \boxed{0x0}^{\alpha_3} * \boxed{list}^{\alpha_4,k_2}}_{\mathtt{Id}}$$

```
  append(k0,k1);
```

$$\mathcal{O}(t_0^\sharp) = \underbrace{\boxed{lseg(\alpha_1)}^{\alpha_0,k_0,l_0} * \boxed{0x0}^{\alpha_1} * \boxed{lseg(\alpha_3)}^{\alpha_2,k_1,l_1} * \boxed{0x0}^{\alpha_3}}_{\text{Reachable part } \mathcal{R}[l_0, l_1](\mathcal{O}(t_0^\sharp))} * \underbrace{\boxed{list}^{\alpha_4,k_2}}_{\text{Not reachable}}$$

1. Parameter passing
2. Extracting output state
3. Procedure footprint



Computing the summary when needed

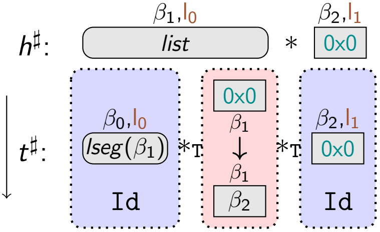
```
double_append(list* k0,list* k1,list* k2){
    append(k0,k1);
```

$\downarrow t_0^\sharp$

$$t_0^\sharp = \underbrace{\overset{\alpha_0,k_0,l_0}{\boxed{lseg(\alpha_1)}} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{\boxed{lseg(\alpha_3)}} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,k_2}{\boxed{list}}}_{\mathtt{Id}}$$

$$\mathcal{O}(t_0^\sharp) = \underbrace{\overset{\alpha_0,k_0,l_0}{\boxed{lseg(\alpha_1)}} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{\boxed{lseg(\alpha_3)}} * \overset{\alpha_3}{\boxed{0x0}}}_{\text{Reachable part } \mathcal{R}[l_0, l_1](\mathcal{O}(t_0^\sharp))} * \underbrace{\overset{\alpha_4,k_2}{\boxed{list}}}_{\text{Not reachable}}$$

$$\mathcal{R}[l_0, l_1](\mathcal{O}(t_0^\sharp)) \sqsubseteq h^\sharp \ ?$$



1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

```
double_append(list* k0,list* k1,list* k2){
    append(k0,k1);
```

$\downarrow t_0^\sharp$

$$t_0^\sharp = \overset{\alpha_0,k_0,l_0}{lseg(\alpha_1)} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{lseg(\alpha_3)} * \overset{\alpha_3}{\boxed{0x0}} * \overset{\alpha_4,k_2}{list} \quad \mathtt{Id}$$

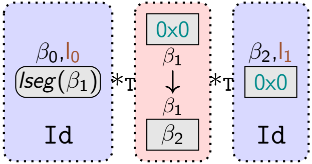
$$\mathcal{O}(t_0^\sharp) = \underbrace{\overset{\alpha_0,k_0,l_0}{lseg(\alpha_1)} * \overset{\alpha_1}{\boxed{0x0}} * \overset{\alpha_2,k_1,l_1}{lseg(\alpha_3)} * \overset{\alpha_3}{\boxed{0x0}}}_{\text{Reachable part } \mathcal{R}[l_0,l_1](\mathcal{O}(t_0^\sharp))} * \underbrace{\overset{\alpha_4,k_2}{list}}_{\text{Not reachable}}$$

1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

$\mathcal{R}[l_0, l_1](\mathcal{O}(t_0^\sharp)) \sqsubseteq h^\sharp$? No

$$h^\sharp: \overset{\beta_1,l_0}{list} * \overset{\beta_2,l_1}{\boxed{0x0}}$$

$t^\sharp$:



Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow t_0^\sharp$

$$t_0^\sharp = \underset{\alpha_0, k_0, l_0}{lseg(\alpha_1)} * \underset{\alpha_1}{\boxed{0x0}} * \underset{\alpha_2, k_1, l_1}{lseg(\alpha_3)} * \underset{\alpha_3}{\boxed{0x0}} * \underset{\alpha_4, k_2}{list}$$

Id

```
  append(k0,k1);
```

$$\mathcal{O}(t_0^\sharp) = \underbrace{\underset{\alpha_0, k_0, l_0}{lseg(\alpha_1)} * \underset{\alpha_1}{\boxed{0x0}} * \underset{\alpha_2, k_1, l_1}{lseg(\alpha_3)} * \underset{\alpha_3}{\boxed{0x0}}}_{\text{Reachable part } \mathcal{R}[l_0, l_1](\mathcal{O}(t_0^\sharp))} * \underbrace{\underset{\alpha_4, k_2}{list}}_{\text{Not reachable}}$$

1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

Summary recomputation

$$h^\sharp : \underset{\beta_1, l_0}{list} * \underset{\beta_2, l_1}{\boxed{0x0}}$$

$\downarrow t^\sharp :$

Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow t_0^\sharp$ $\quad t_0^\sharp = \underbrace{\underset{\alpha_0,k_0,l_0}{\boxed{lseg(\alpha_1)}} * \underset{\alpha_1}{\boxed{0x0}} * \underset{\alpha_2,k_1,l_1}{\boxed{lseg(\alpha_3)}} * \underset{\alpha_3}{\boxed{0x0}} * \underset{\alpha_4,k_2}{\boxed{list}}}_{\mathtt{Id}}$

```
  append(k0,k1);
```

$$\mathcal{O}(t_0^\sharp) = \underbrace{\underset{\alpha_0,k_0,l_0}{\boxed{lseg(\alpha_1)}} * \underset{\alpha_1}{\boxed{0x0}} * \underset{\alpha_2,k_1,l_1}{\boxed{lseg(\alpha_3)}} * \underset{\alpha_3}{\boxed{0x0}}}_{\text{Reachable part } \mathcal{R}[l_0,\, l_1](\mathcal{O}(t_0^\sharp))} * \underbrace{\underset{\alpha_4,k_2}{\boxed{list}}}_{\text{Not reachable}}$$

1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

Summary recomputation

$$h^\sharp := h^\sharp \;\nabla\; \mathcal{R}[l_0,\, l_1](\mathcal{O}(t_0^\sharp))$$

$h^\sharp$: $\underset{\beta_1,l_0}{\boxed{list}} * \underset{\beta_2,l_1}{\boxed{list}}$

$t^\sharp$:

Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
    append(k0,k1);
```

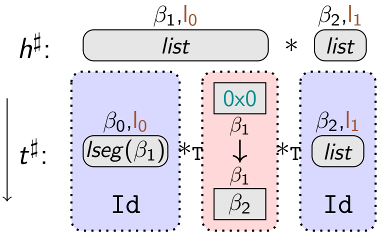
$\downarrow t_0^\sharp$

$t_0^\sharp = \underset{\alpha_0, k_0, l_0}{\boxed{lseg(\alpha_1)}} * \underset{\alpha_1}{\boxed{0x0}} * \underset{\alpha_2, k_1, l_1}{\boxed{lseg(\alpha_3)}} * \underset{\alpha_3}{\boxed{0x0}} * \underset{\alpha_4, k_2}{\boxed{list}}$ Id

$\mathcal{O}(t_0^\sharp) = \underbrace{\boxed{lseg(\alpha_1)}_{\alpha_0, k_0, l_0} * \boxed{0x0}_{\alpha_1} * \boxed{lseg(\alpha_3)}_{\alpha_2, k_1, l_1} * \boxed{0x0}_{\alpha_3}}_{\text{Reachable part } \mathcal{R}[l_0, l_1](\mathcal{O}(t_0^\sharp))} * \underbrace{\boxed{list}_{\alpha_4, k_2}}_{\text{Not reachable}}$

1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

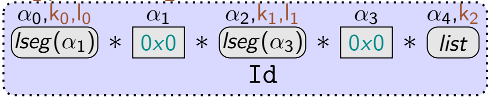
Summary recomputation



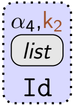
Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow t_0^\sharp$ $\quad t_0^\sharp =$

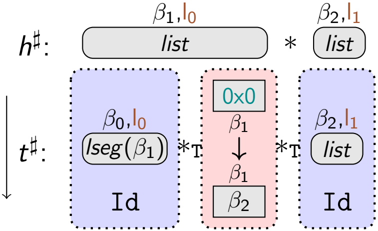



```
  append(k0,k1);
```



1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

   Summary recomputation
5. Summary application

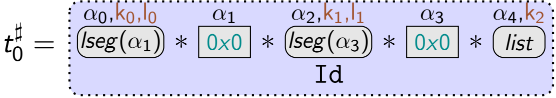



```
}
```

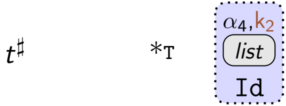
Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
```

$\downarrow t_0^\sharp$

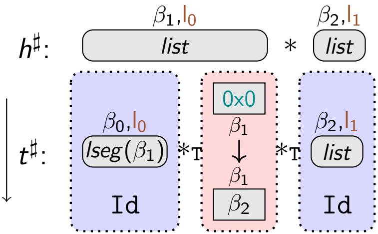



```
  append(k0,k1);
```



1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

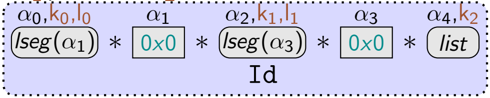
   Summary recomputation
5. Summary application



Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
```

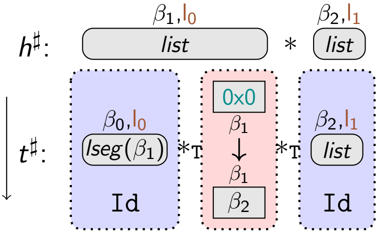
$\downarrow t_0^\sharp$ $\quad t_0^\sharp =$




```
  append(k0,k1);
```

1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

   Summary recomputation
5. Summary application



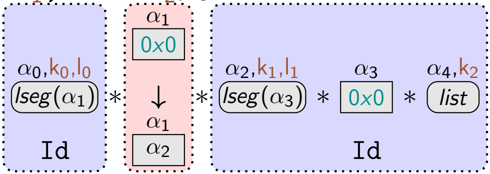
}

Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
```

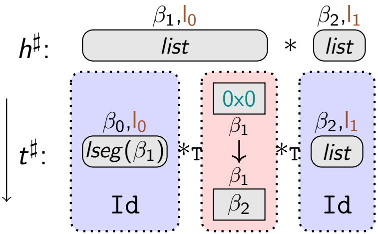
$\downarrow t_0^\sharp$

```
  append(k0,k1);
```



1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

   Summary recomputation
5. Summary application

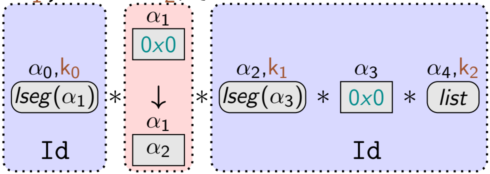


Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
```

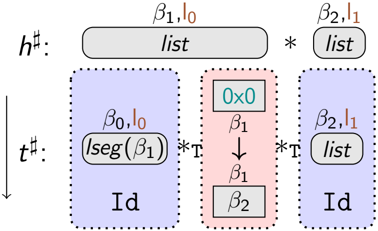
$\downarrow t_0^\sharp$

```
  append(k0,k1);
```



1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

   Summary recomputation
5. Summary application
6. Parameter suppression

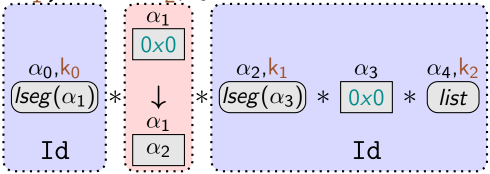


Computing the summary when needed

```
double_append(list* k0,list* k1,list* k2){
  append(k0,k1);
```

$t_0^\sharp$

$t_1^\sharp$



1. Parameter passing
2. Extracting output state
3. Procedure footprint
4. Summary coverage testing

   Summary recomputation
5. Summary application
6. Parameter suppression