A Relational Shape Abstract Domain

Hugo Illous[1,2], Matthieu Lemerre[1] and Xavier Rival[2]

[1]CEA/LSL

[2]ENS/INRIA Paris (Antique)

Gallium Seminar, March 05, 2018

Static Code Analysis of Data Structures

- Shape: **Dynamically allocated** data structure (linked lists, trees, ...)
- Usual goals:
  - **Structural** properties (heap, red-black, AVL, sorted list, absence of cycles...)
  - **Safety** properties: absence of memory leaks, null pointer dereference, dangling pointers, double free...
- This talk:
  - automatic inference of safety and relational shape properties
  - Using abstract interpretation (with domains based on separation logic)

Background: Analysis of Reachable Memory States

```
struct list { int data; struct list *next; };
#define SIZE sizeof(struct list)


struct list *
cons(struct list *l) {
  list *h = malloc(SIZE);
  h->next = l;
  return h;
}
```

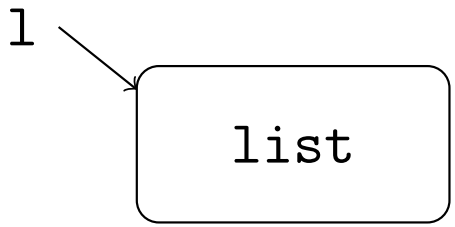
Memory States Properties

- cons inputs a well formed linked list
- cons outputs a list node followed by a well formed linked list

Background: Analysis of Reachable Memory States

```
struct list { int data; struct list *next; };
#define SIZE sizeof(struct list)


struct list *
cons(struct list *l) {
  list *h = malloc(SIZE);
  h->next = l;
  return h;
}
```



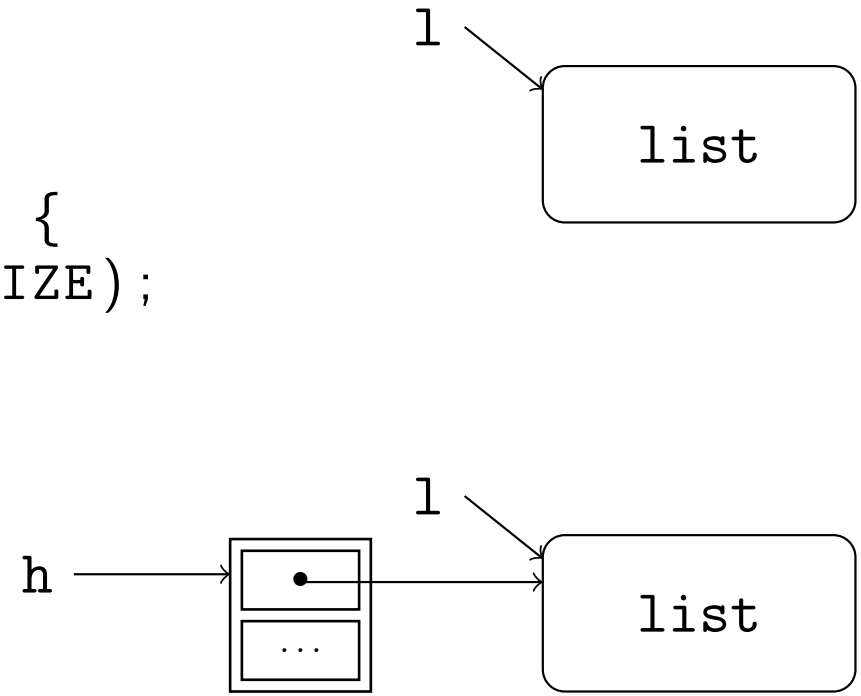
Memory States Properties

- cons inputs a well formed linked list
- cons outputs a list node followed by a well formed linked list

Background: Analysis of Reachable Memory States

```
struct list { int data; struct list *next; };
#define SIZE sizeof(struct list)


struct list *
cons(struct list *l) {
  list *h = malloc(SIZE);
  h->next = l;
  return h;
}
```



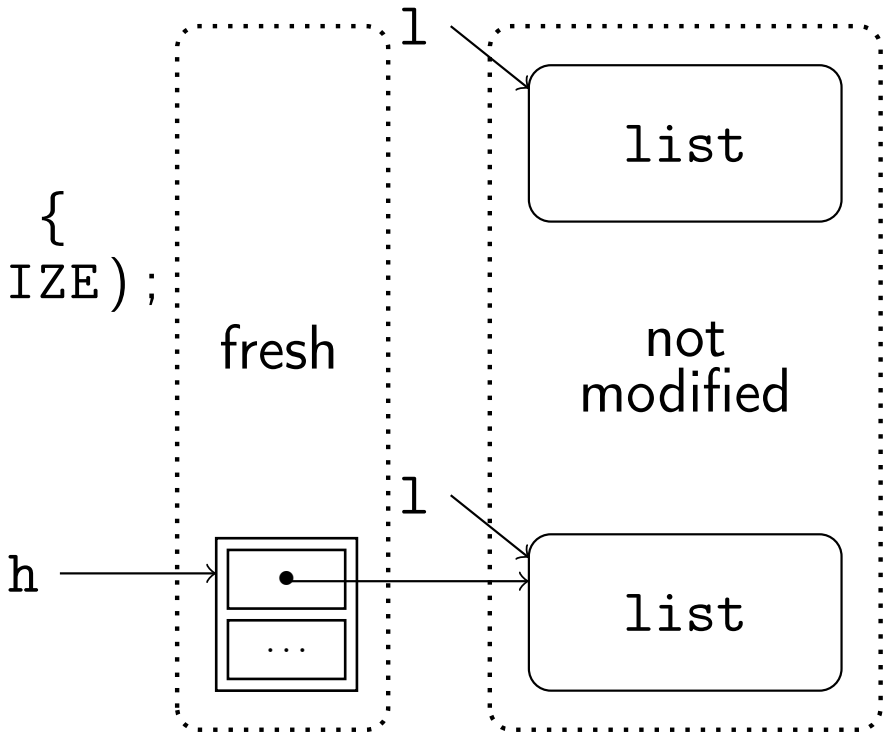
Memory States Properties

- cons inputs a well formed linked list
- cons outputs a list node followed by a well formed linked list

Contribution: Relational Analysis of Memory Properties

```
struct list { int data; struct list *next; };
#define SIZE sizeof(struct list)

struct list *
cons(struct list *l) {
  list *h = malloc(SIZE);
  h->next = l;
  return h;
}
```



Relational Memory Properties

- `l` has been **not modified** by `cons`
- `h` **didn't belong** to the input list

Contributions and Outline

Contribution

We propose a **generic** method to extend a **reachable state memory** analysis into a **relational memory** analysis

Outline:

1. A Relational Separation Logic
2. Static Analysis Algorithm
3. Experiments

Memory States Abstraction: Separation Logic Connectives

Separation Logic Formula $\alpha \mapsto \beta * \beta \mapsto 41$

Abstract Memory

Concrete Memory

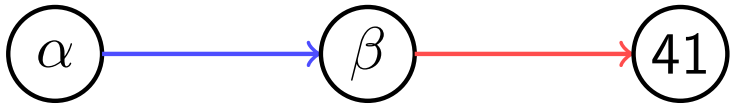
Selected Connectives:

- Points-to Predicate: $\mapsto$
- Separating Conjunction: $*$
- Neutral Element (empty memory): **emp**

Memory States Abstraction: Separation Logic Connectives

Separation Logic Formula $\alpha \mapsto \beta * \beta \mapsto 41$

Abstract Memory



Concrete Memory

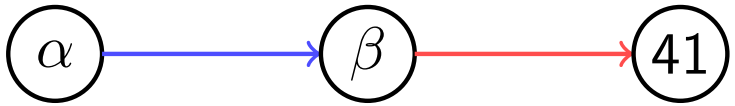
Selected Connectives:

- Points-to Predicate: $\mapsto$
- Separating Conjunction: $*$
- Neutral Element (empty memory): **emp**

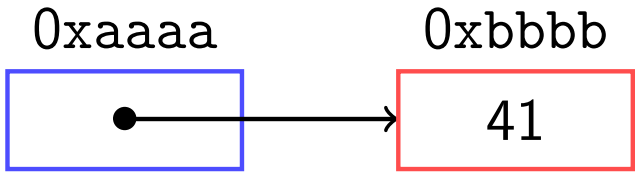
Memory States Abstraction: Separation Logic Connectives

Separation Logic Formula $\alpha \mapsto \beta * \beta \mapsto 41$

Abstract Memory



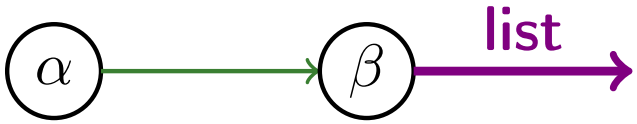
Concrete Memory



Selected Connectives:

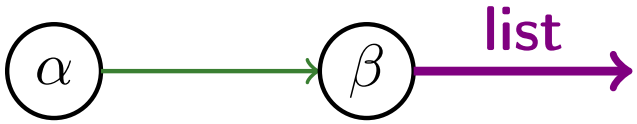
- Points-to Predicate: $\mapsto$
- Separating Conjunction: $*$
- Neutral Element (empty memory): **emp**

Memory States Abstraction: Inductive Predicates



$$\alpha \mapsto \beta * \mathsf{list}(\beta)$$

Memory States Abstraction: Inductive Predicates

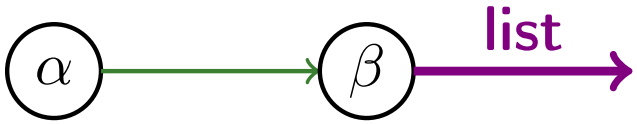


$$\alpha \mapsto \beta * \mathsf{list}(\beta)$$

$$\underline{\underline{\textit{def}}}$$

$$\alpha \mapsto \mathbf{0x0}$$

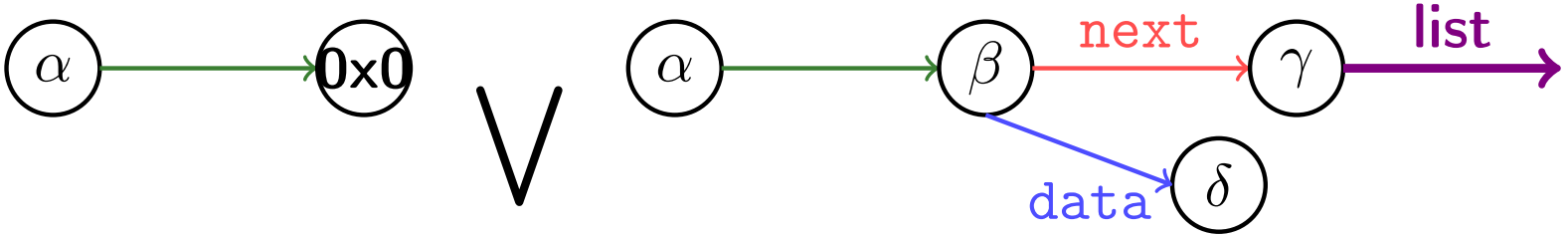
Memory States Abstraction: Inductive Predicates



$$\alpha \mapsto \beta * \mathbf{list}(\beta)$$

def

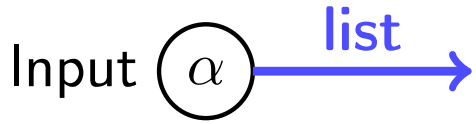
$$=$$



$$\alpha \mapsto \mathbf{0x0} \qquad \alpha \mapsto \beta * \beta \cdot \mathtt{data} \mapsto \delta * \beta \cdot \mathtt{next} \mapsto \gamma * \mathbf{list}(\gamma)$$

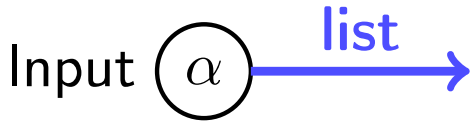
The Need For Relations

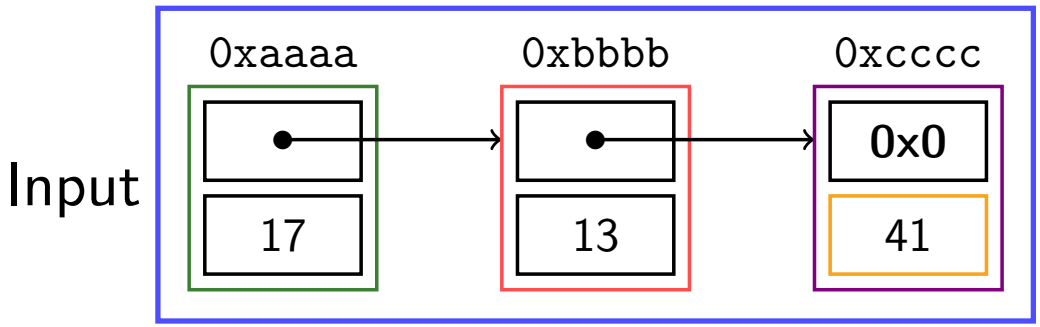
```
struct list *
cons(struct list *l) {
  list *h = malloc(SIZE);
  h->next = l;
  return h;
}
```



The Need For Relations

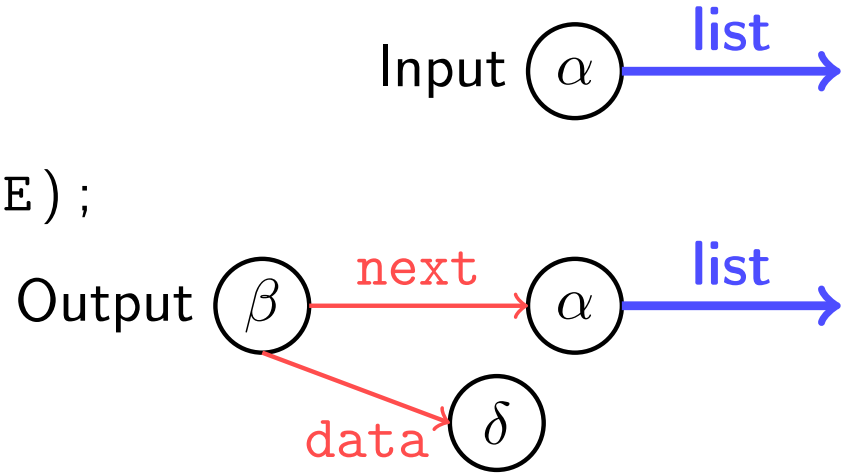
```
struct list *
cons(struct list *l) {
  list *h = malloc(SIZE);
  h->next = l;
  return h;
}
```

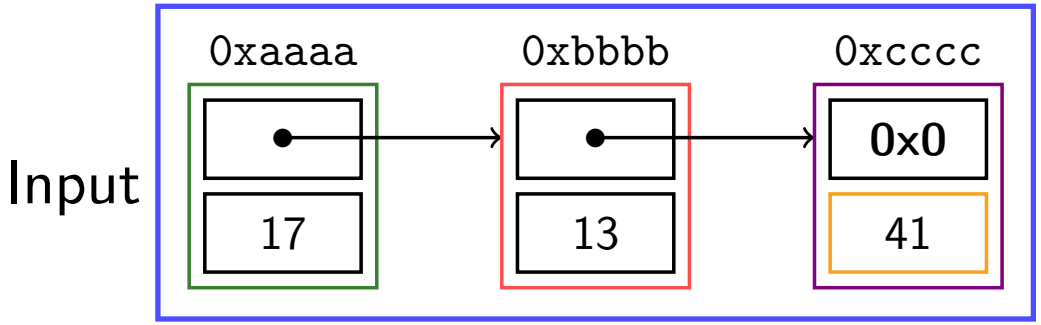




The Need For Relations

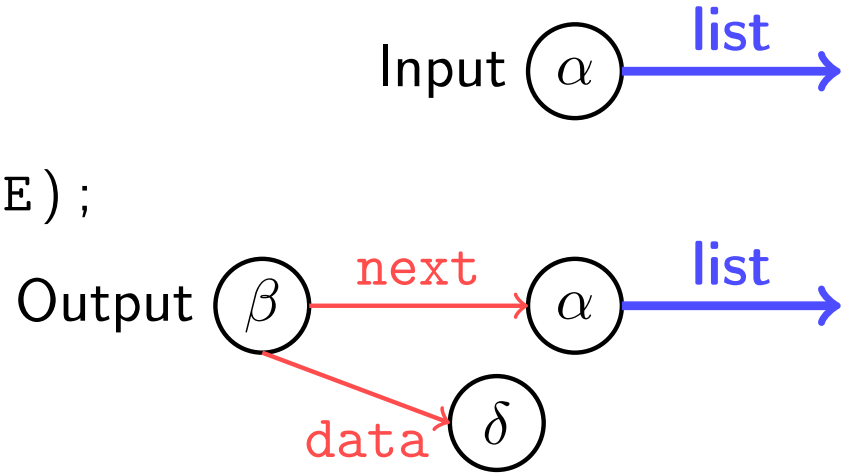
```
struct list *
cons(struct list *l) {
  list *h = malloc(SIZE);
  h->next = l;
  return h;
}
```

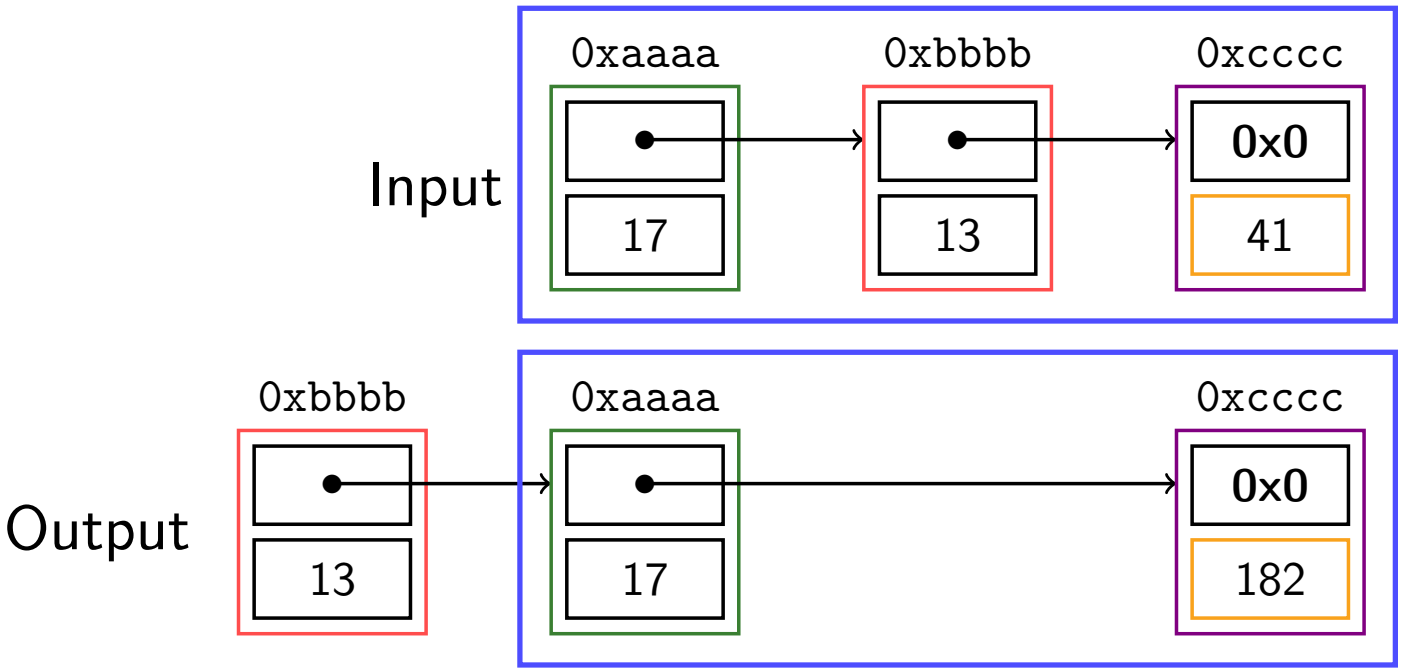




The Need For Relations

```
struct list *
cons(struct list *l) {
  list *h = malloc(SIZE);
  h->next = l;
  return h;
}
```





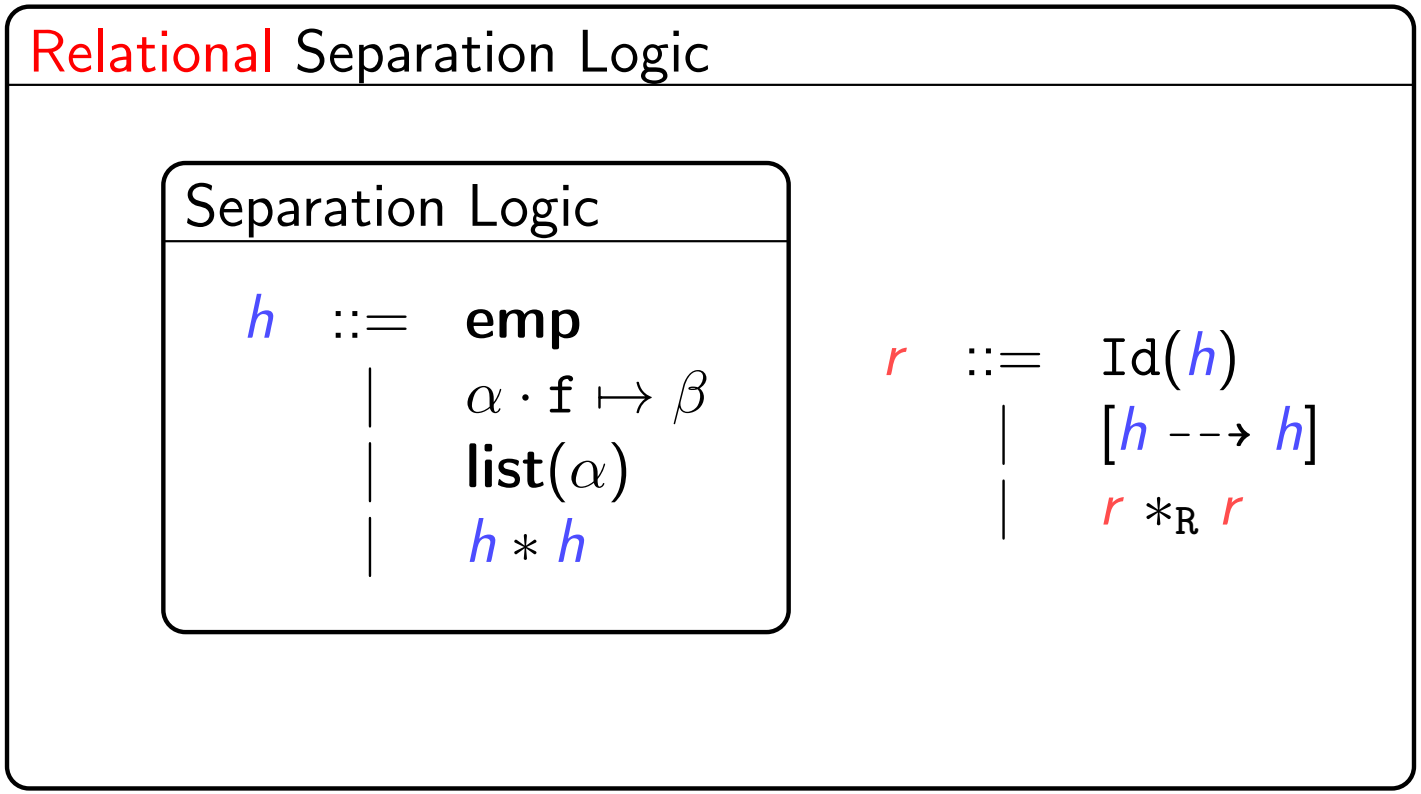
From Separation Logic to Relational Separation Logic

Separation Logic

$$
\begin{array}{rcl}
h & ::= & \mathbf{emp} \\
 & | & \alpha \cdot \mathtt{f} \mapsto \beta \\
 & | & \mathbf{list}(\alpha) \\
 & | & h * h
\end{array}
$$

- Separation Logic formulas represent **states**

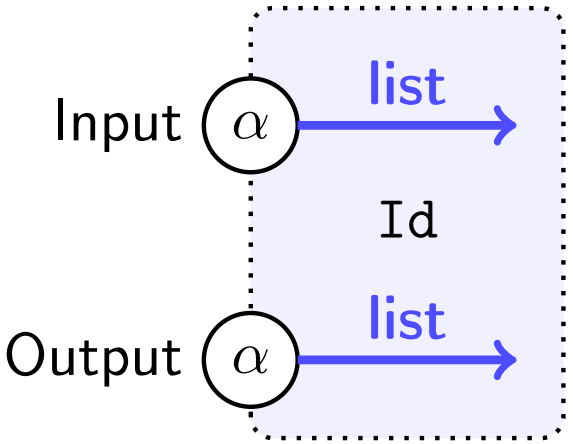
From Separation Logic to Relational Separation Logic

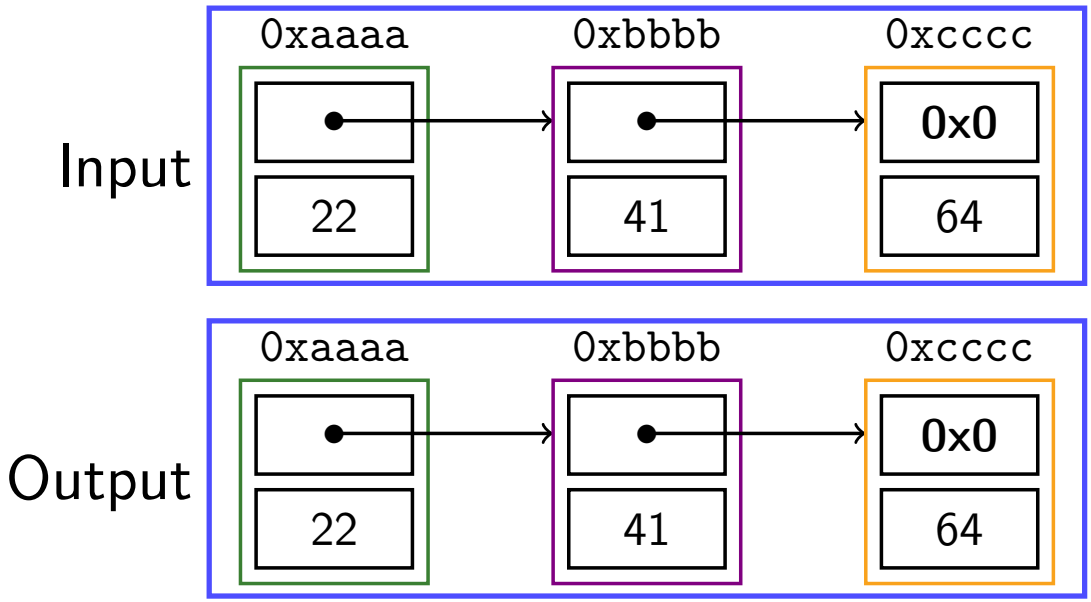


- Separation Logic formulas represent **states**
- Relational Separation Logic formulas represent **pairs of states**

Identity Relation: $\mathtt{Id}(h)$

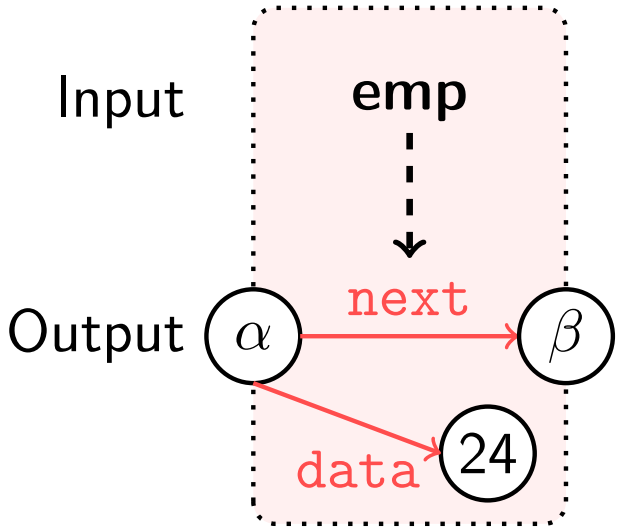
$\mathtt{Id}(\mathsf{list}(\alpha))$

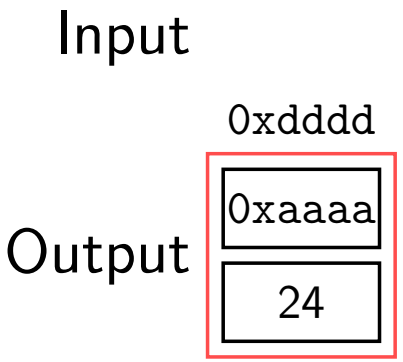




Transform into Relation: $[h_i \dashrightarrow h_o]$

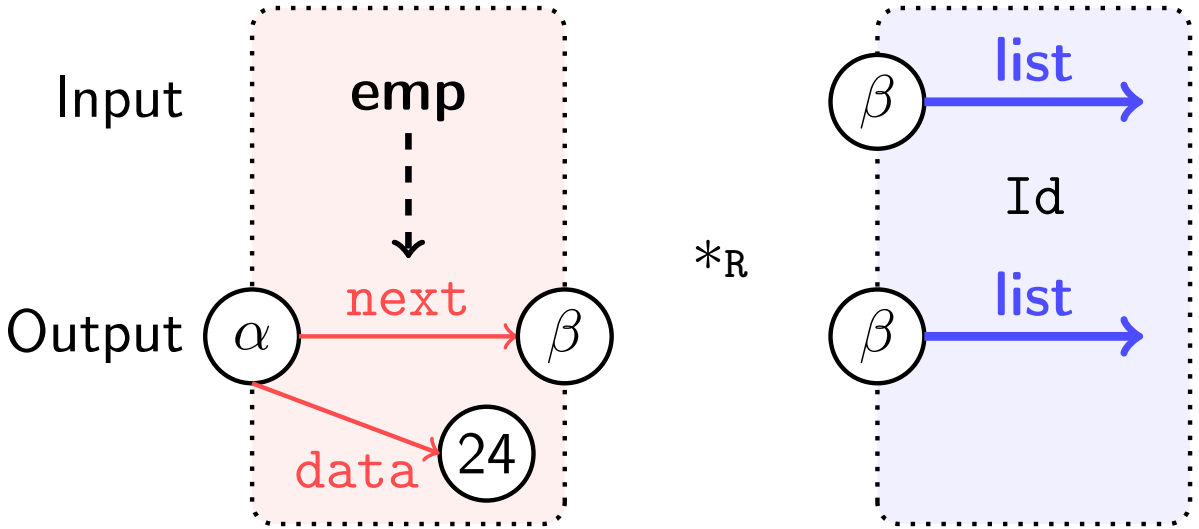
$$[\textbf{emp} \quad \dashrightarrow \quad \alpha \cdot \texttt{data} \mapsto 24 * \alpha \cdot \texttt{next} \mapsto \beta]$$

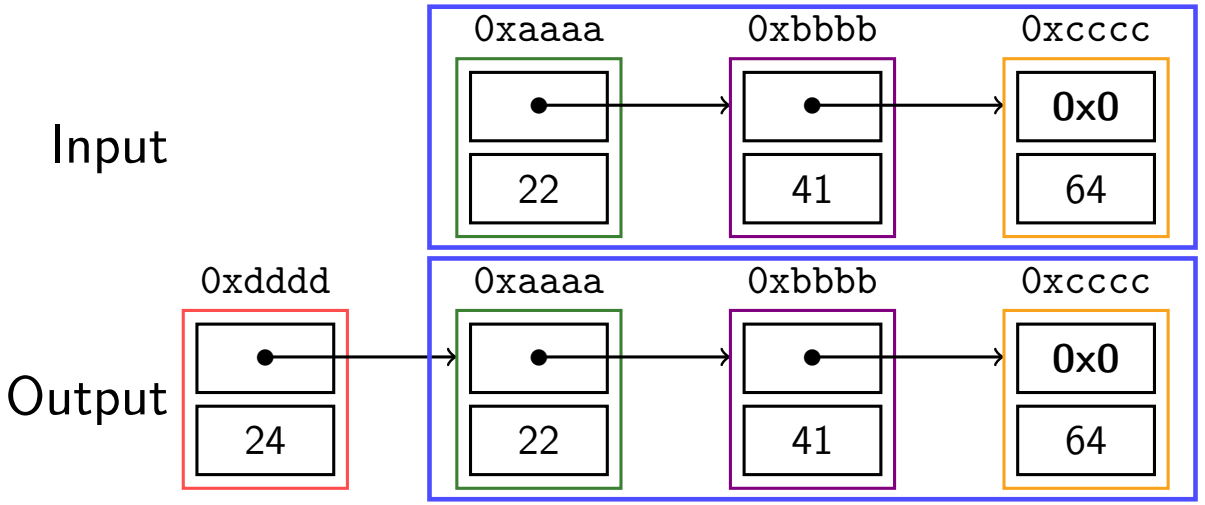




Relation Separating Conjunction: $r_0 *_R r_1$

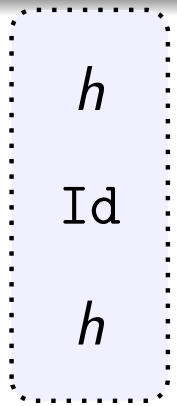
$$[\mathbf{emp} \dashrightarrow \alpha \cdot \texttt{data} \mapsto 24 * \alpha \cdot \texttt{next} \mapsto \beta] *_R \texttt{Id}(\texttt{list}(\beta))$$



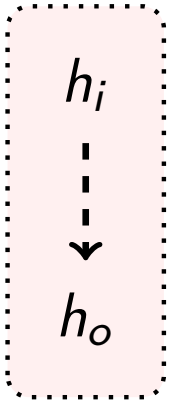


Relational Separation Logic Connectives

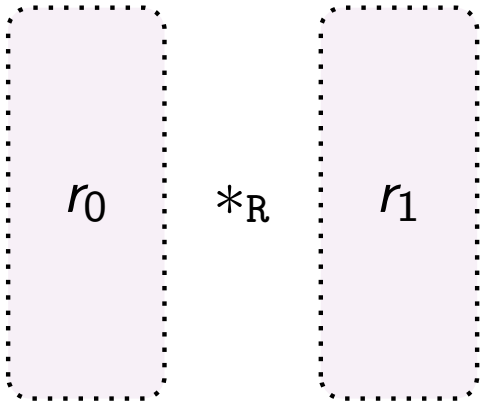
$\texttt{Id}(h)$: Unmodified memory



$[h_i \dashrightarrow h_o]$: Memory transformation

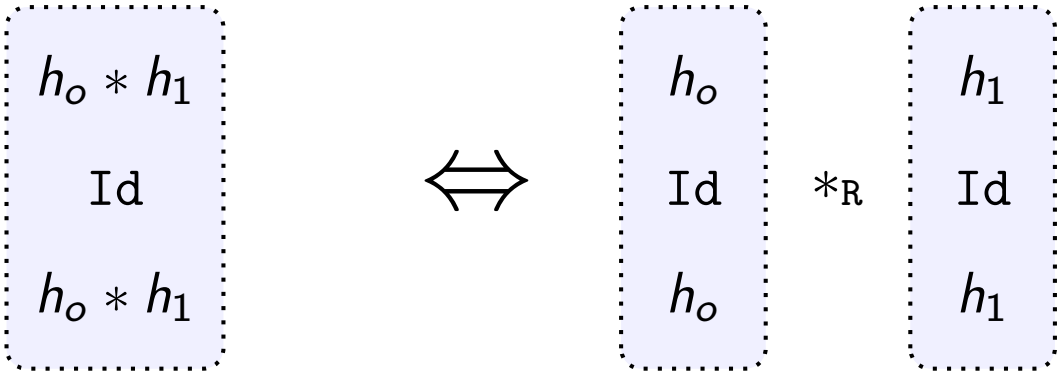


$r_0 *_{\mathtt{R}} r_1$: Independent relations

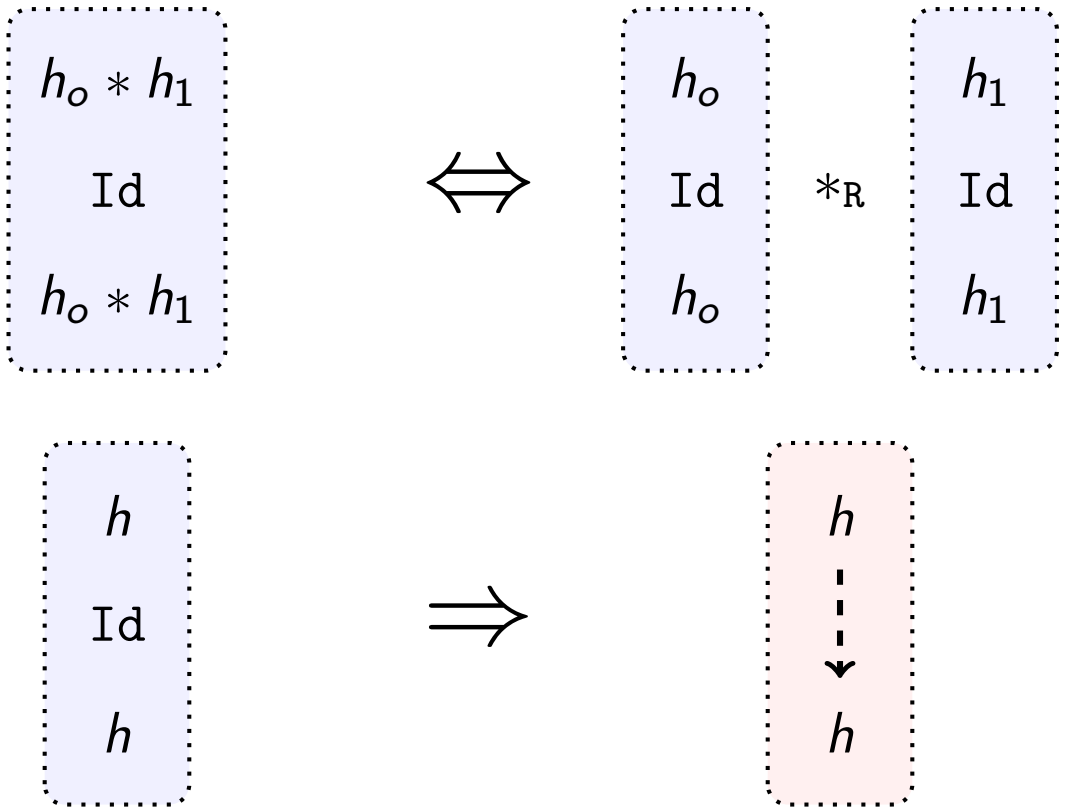


Properties: Weakening Relations

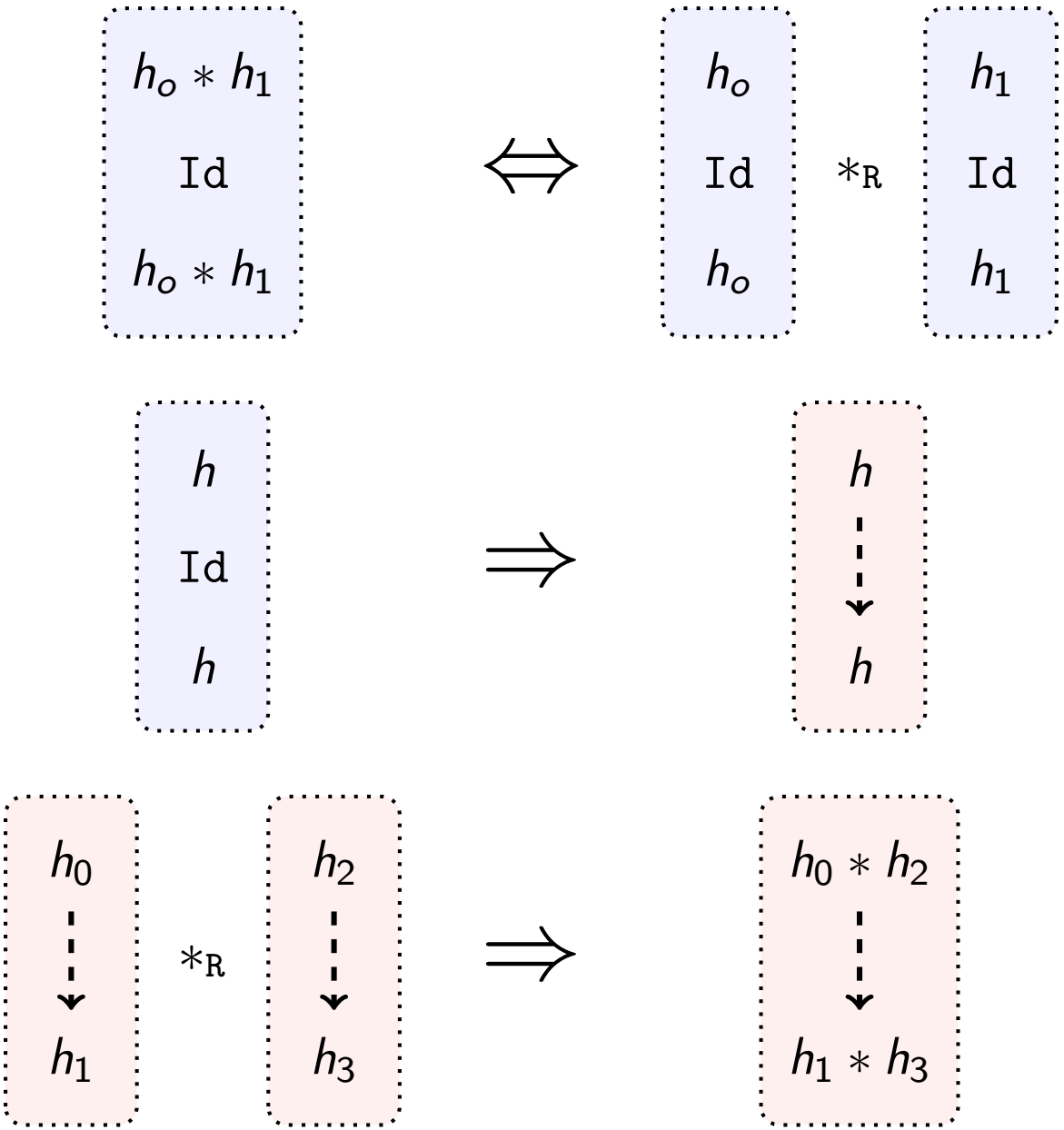
Properties: Weakening Relations



Properties: Weakening Relations

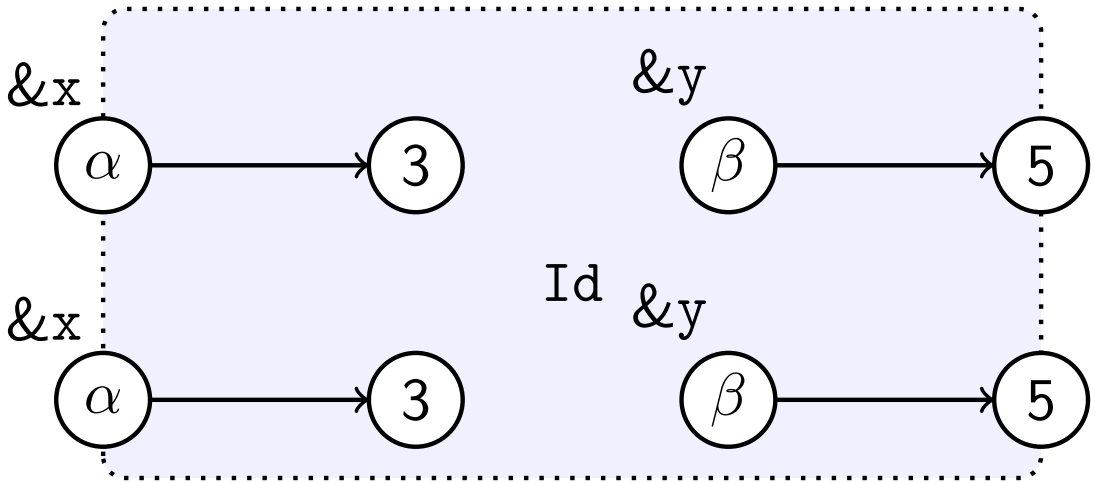


Properties: Weakening Relations



Transfer function: Assignment

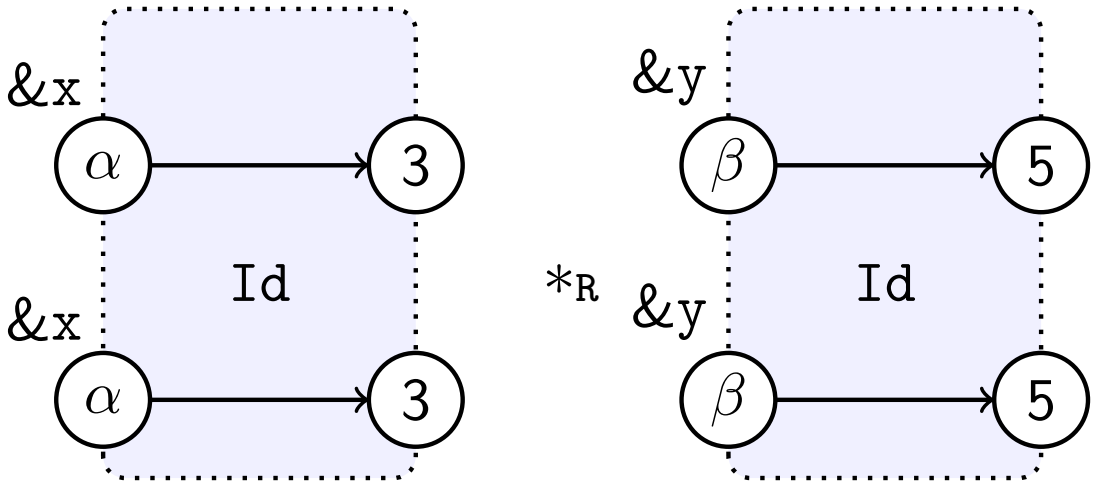
Example: $\mathtt{y} := 17;$



$\mathtt{Id}(\alpha \mapsto 3 * \beta \mapsto 5)$

Transfer function: Assignment

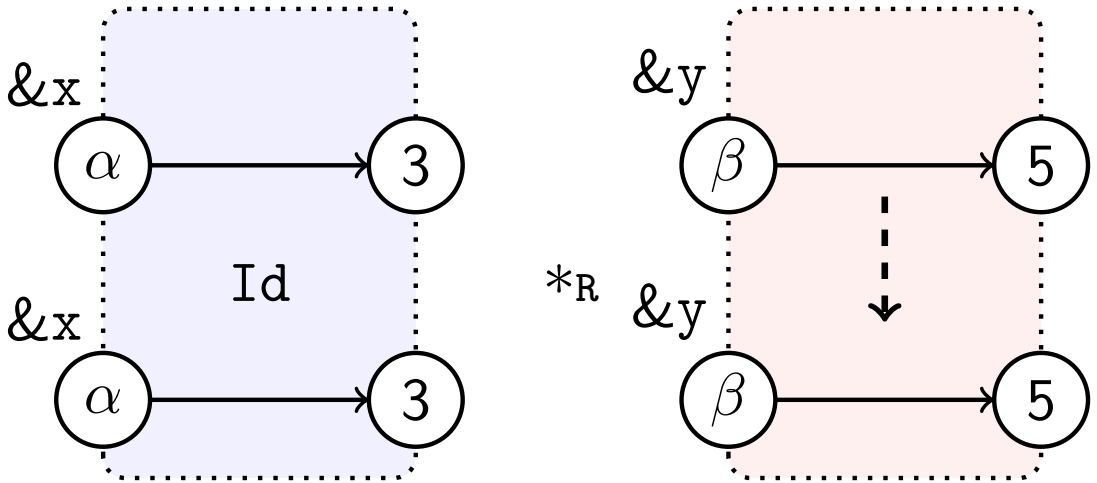
Example: $\mathtt{y} := 17;$



$\mathtt{Id}(\alpha \mapsto 3) \quad *_{\mathtt{R}} \quad \mathtt{Id}(\beta \mapsto 5)$

Transfer function: Assignment

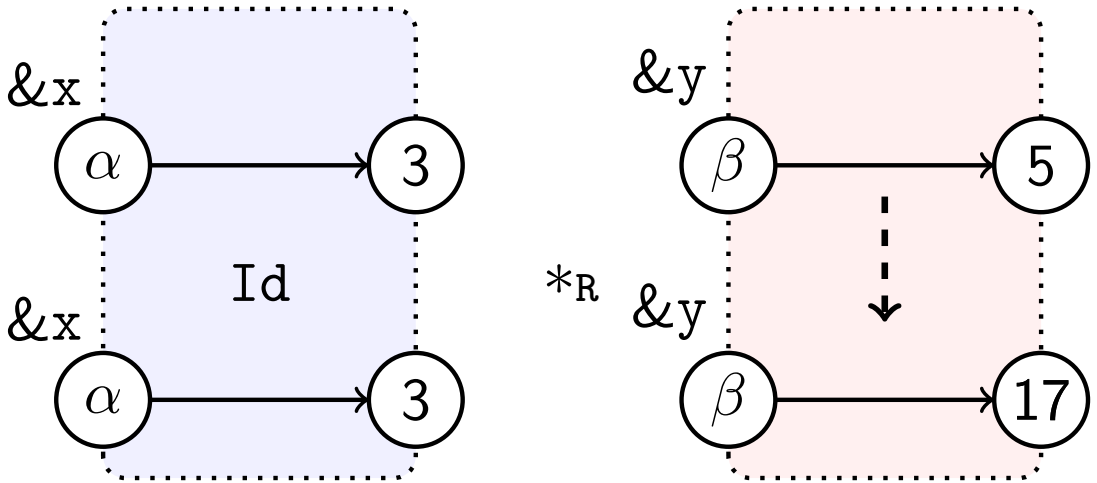
Example: $\mathtt{y} := 17;$



$\mathtt{Id}(\alpha \mapsto 3)$ $\qquad$ $*_{\mathtt{R}}$ $[\beta \mapsto 5 \dashrightarrow \beta \mapsto 5]$

Transfer function: Assignment

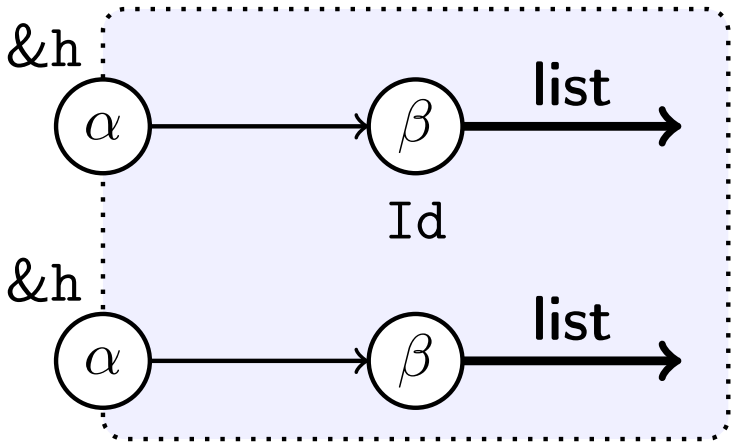
Example: $\mathtt{y} := 17;$



$\mathtt{Id}(\alpha \mapsto 3)$ $\qquad$ $*_{\mathtt{R}}\ [\beta \mapsto 5 \dashrightarrow \beta \mapsto 17]$

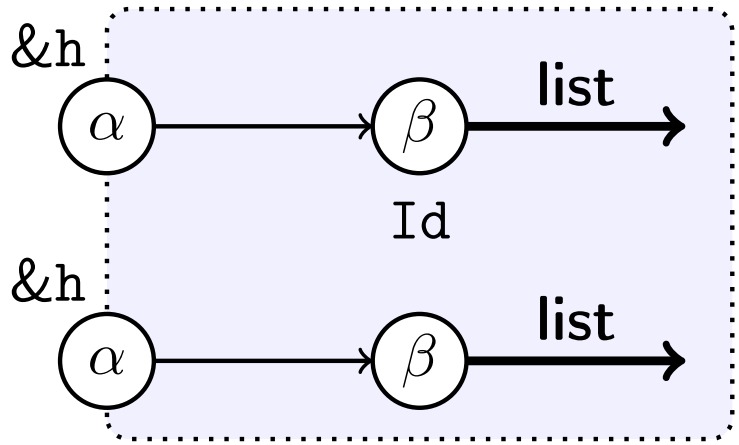
Unfolding over the Identity Relation

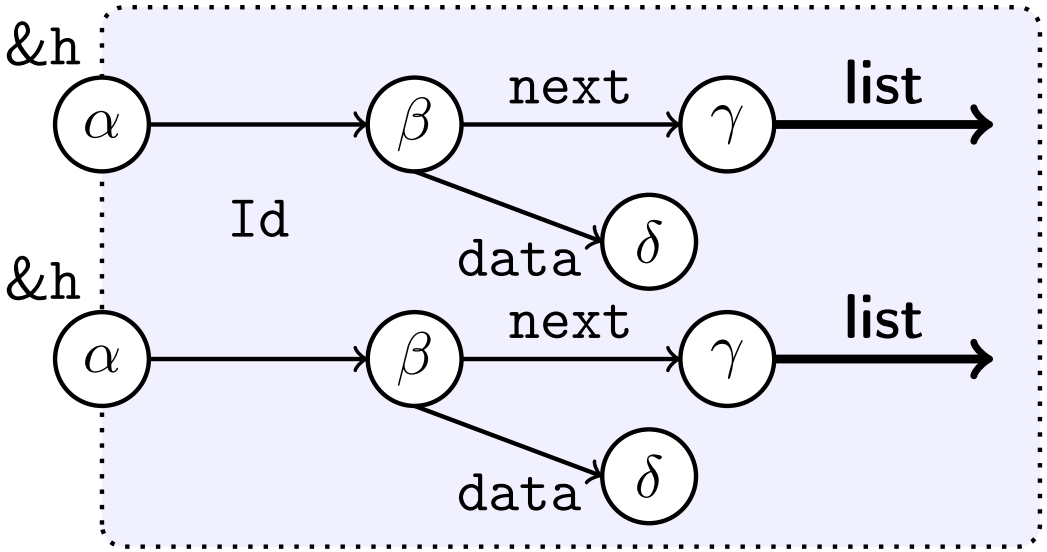
Example: `h->data` := 21;



Unfolding over the Identity Relation

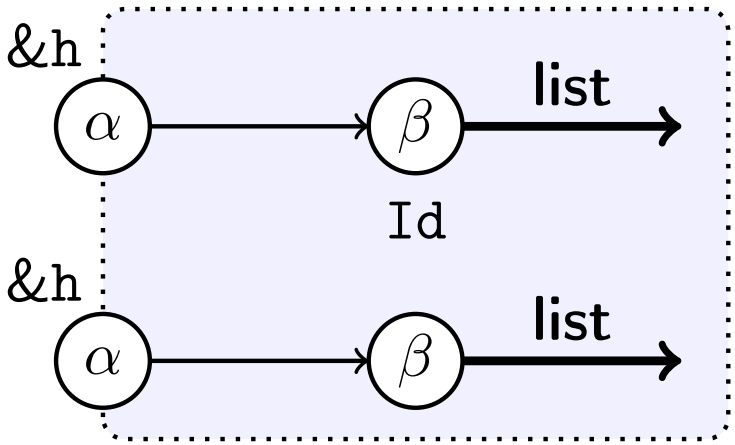
Example: `h->data := 21;`

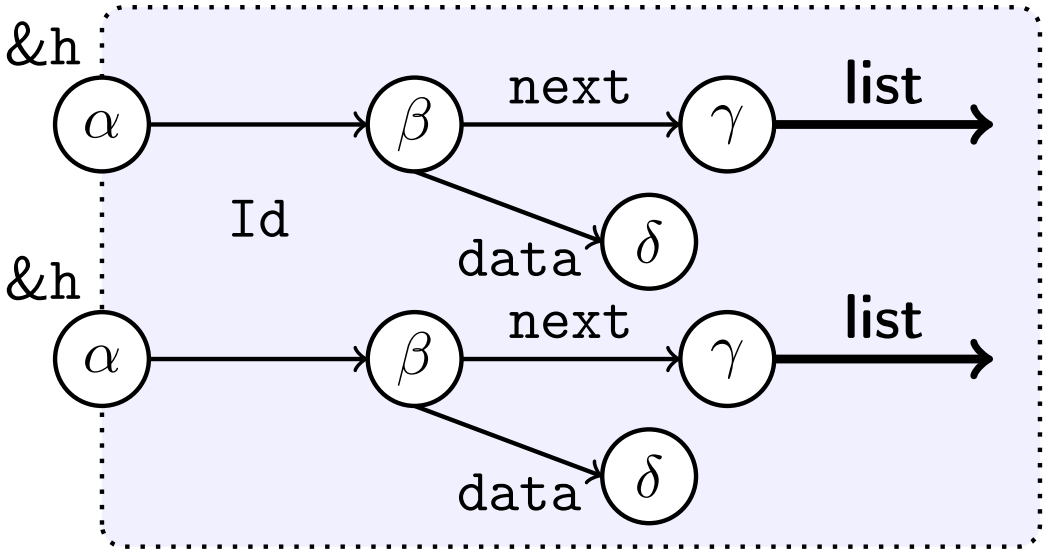


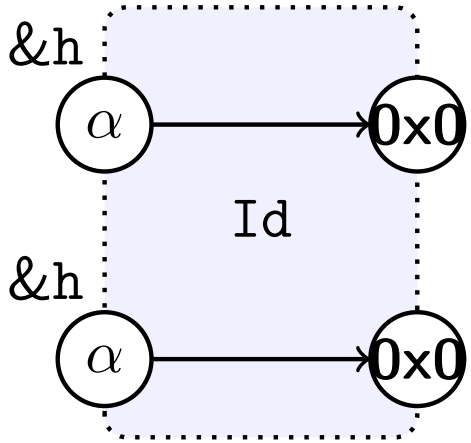


Unfolding over the Identity Relation

Example: `h->data := 21;`

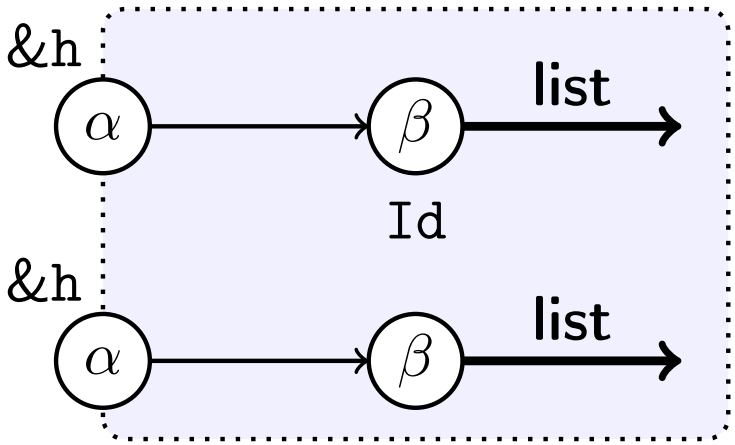


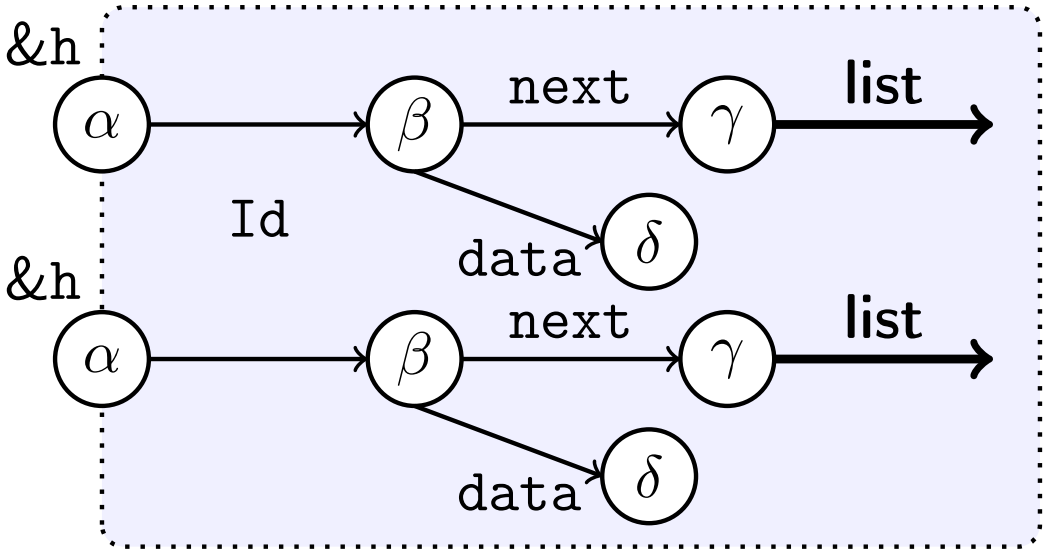


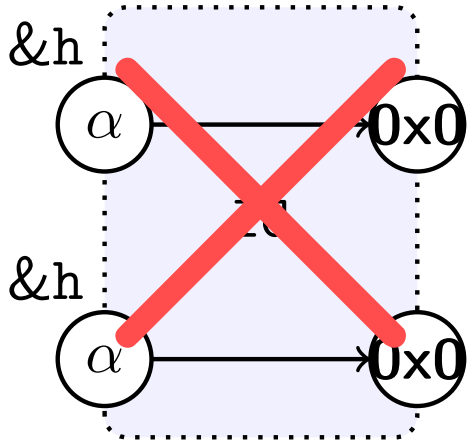


Unfolding over the Identity Relation

Example: `h->data := 21;`

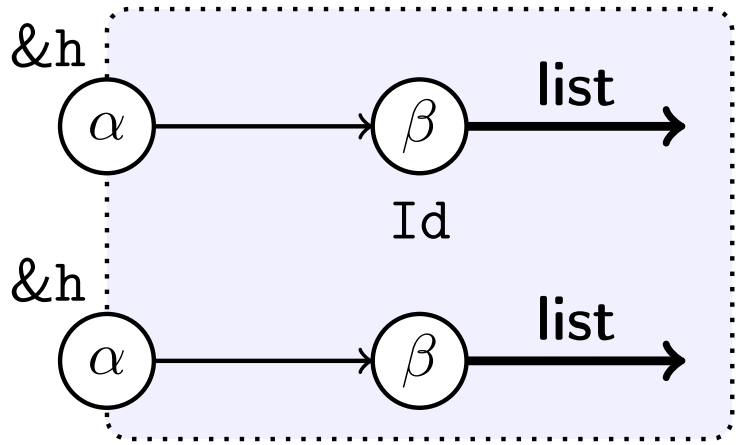


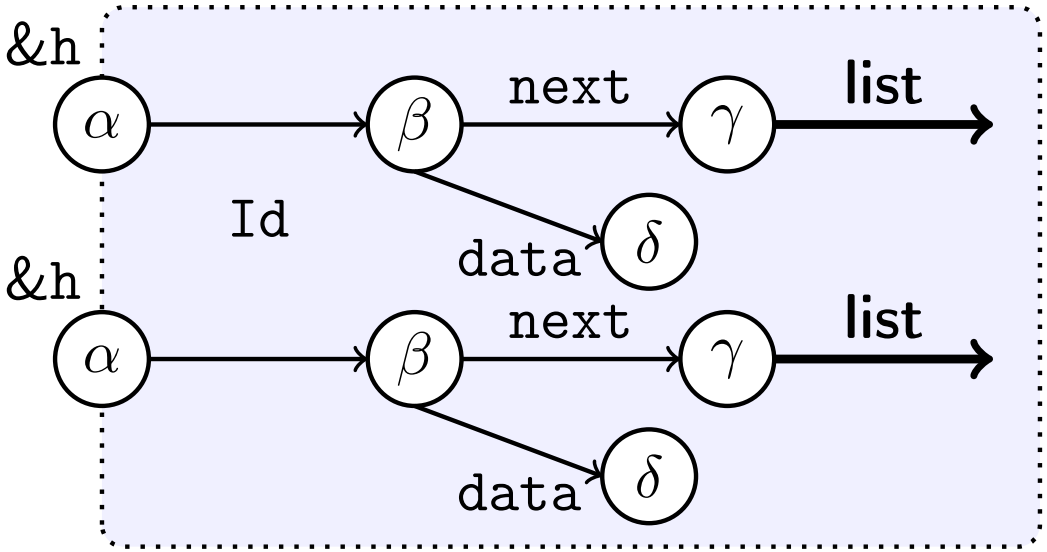




Unfolding over the Identity Relation

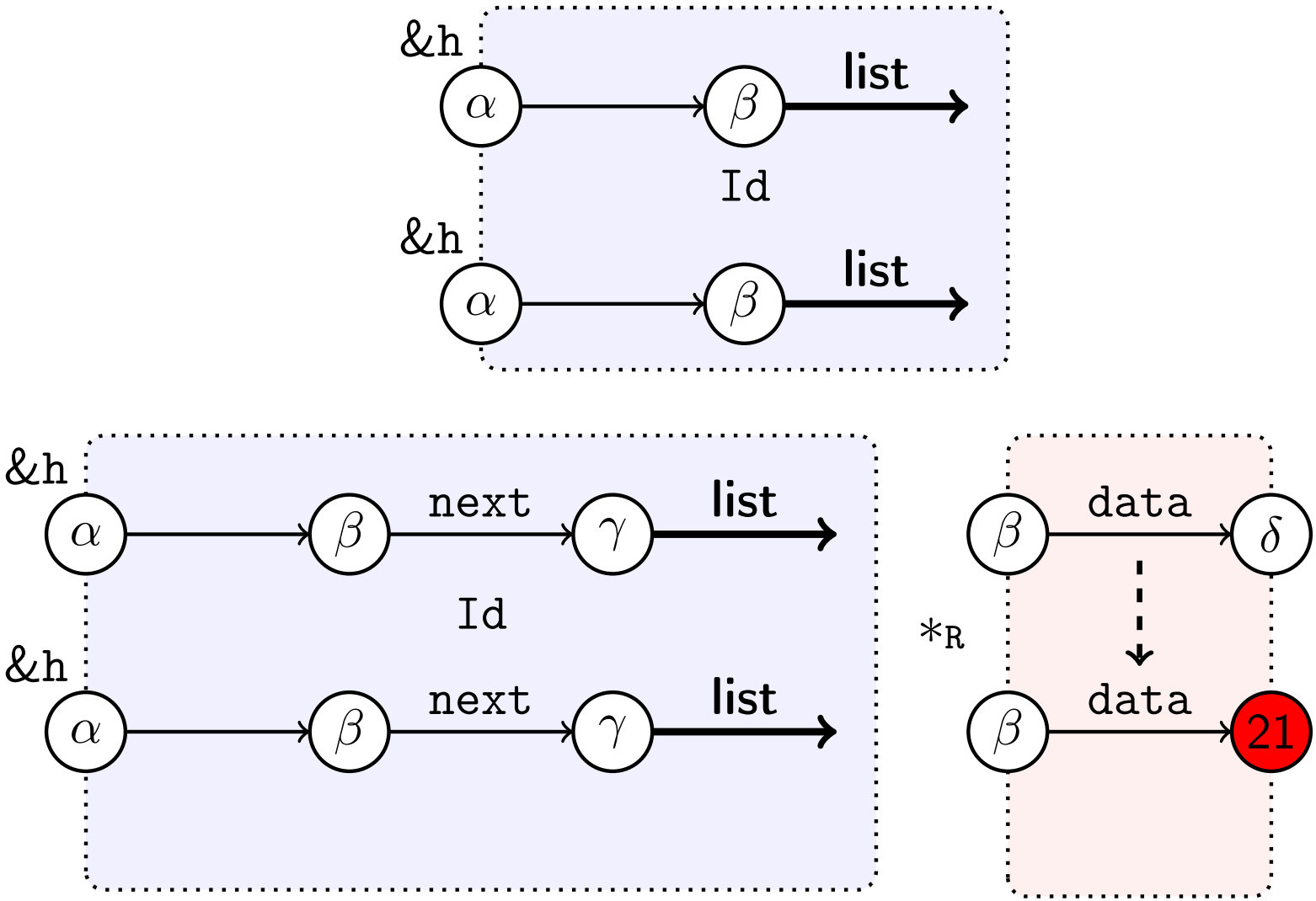
Example: `h->data := 21;`





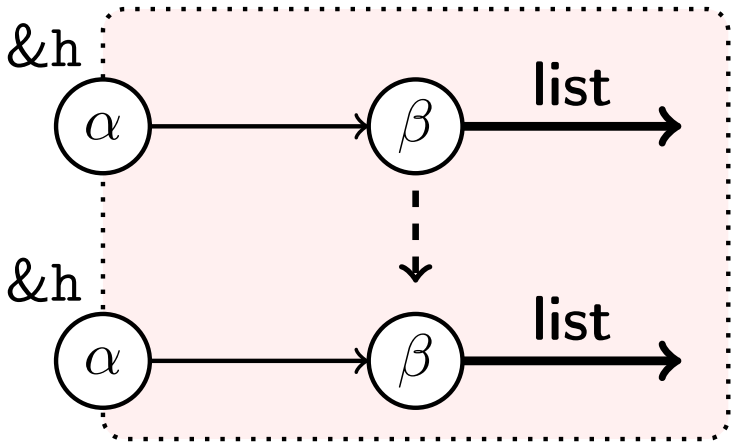
Unfolding over the Identity Relation

Example: `h->data` := 21;



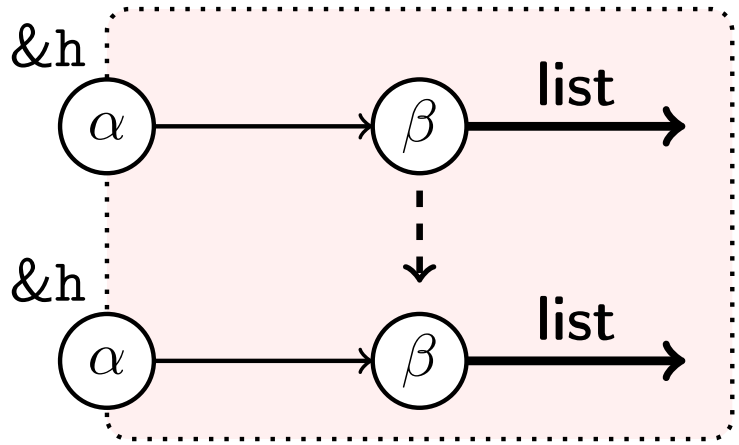
Unfolding over the Transform into Relation

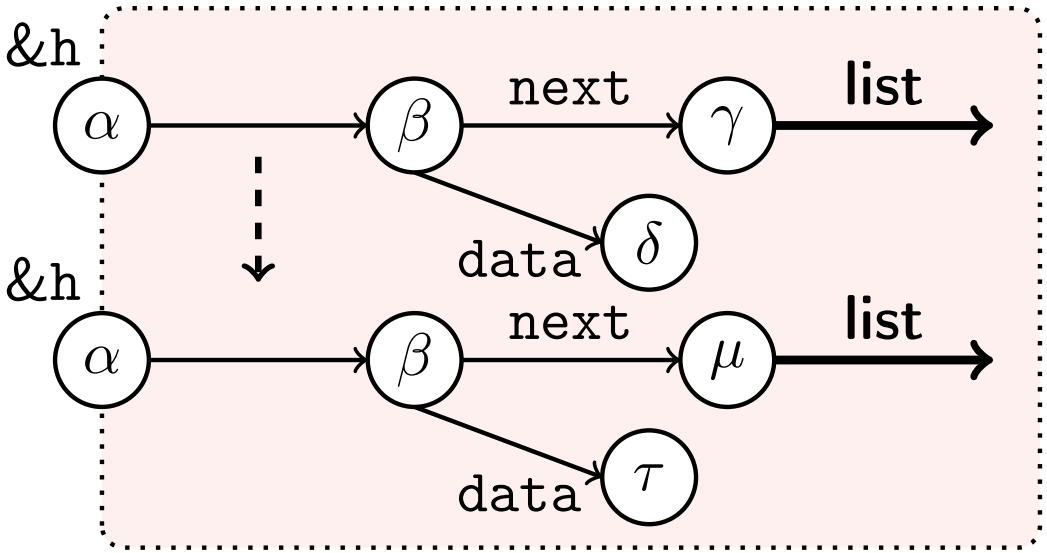
Example: `h->data` := 21;



Unfolding over the Transform into Relation

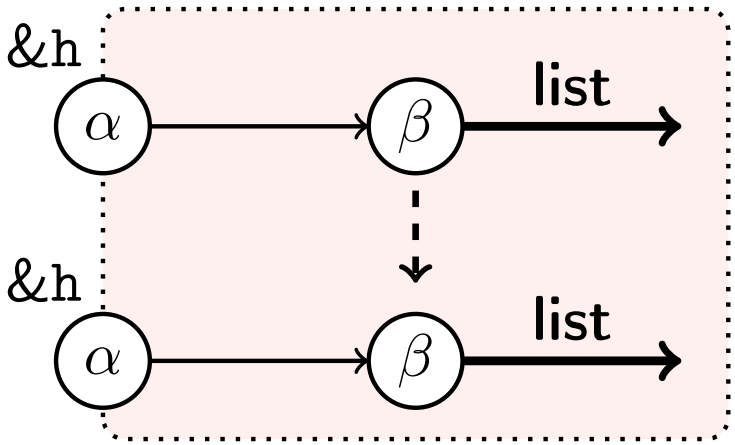
Example: `h->data` := 21;

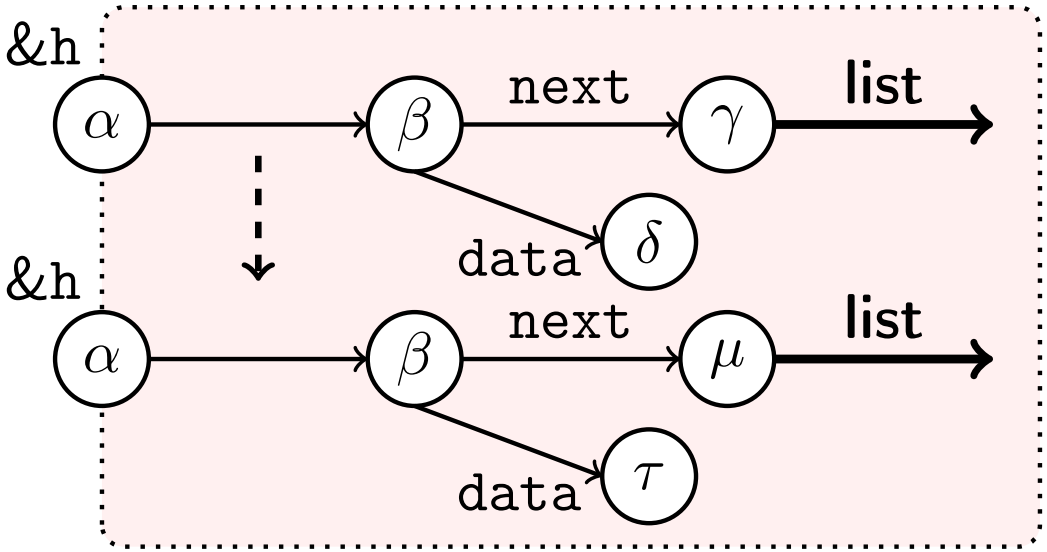


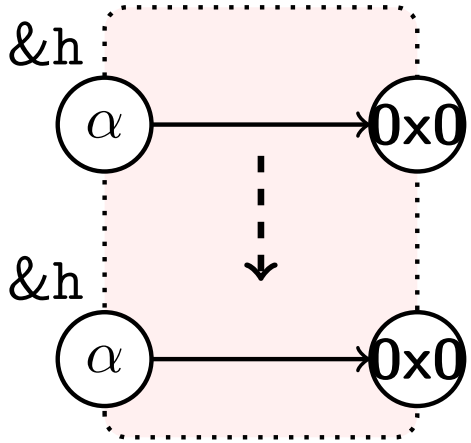


Unfolding over the Transform into Relation

Example: `h->data := 21;`

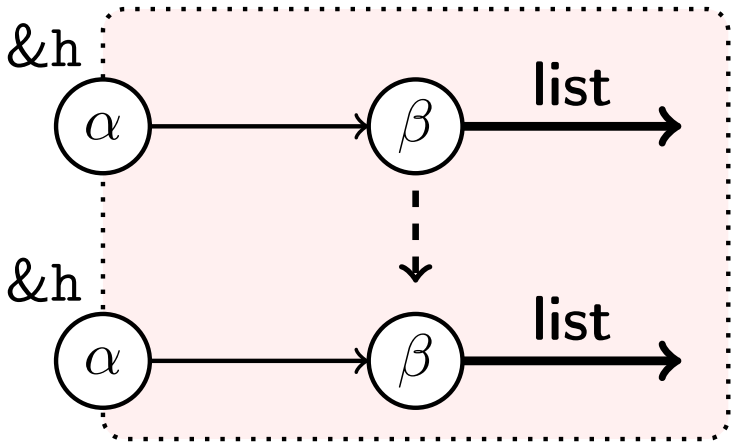


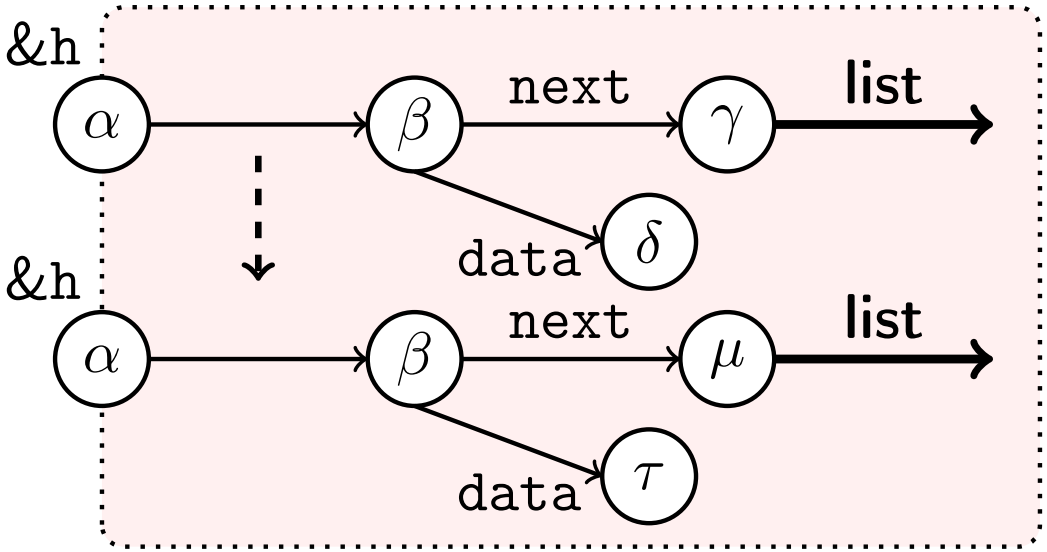


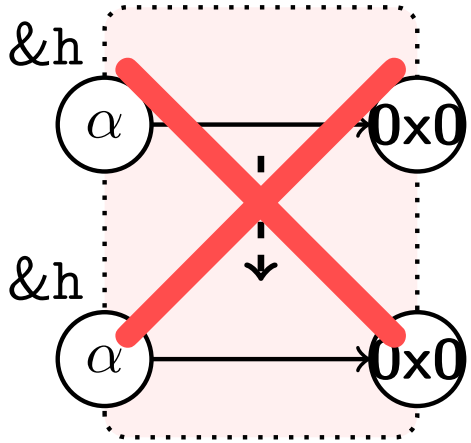


Unfolding over the Transform into Relation

Example: `h->data := 21;`

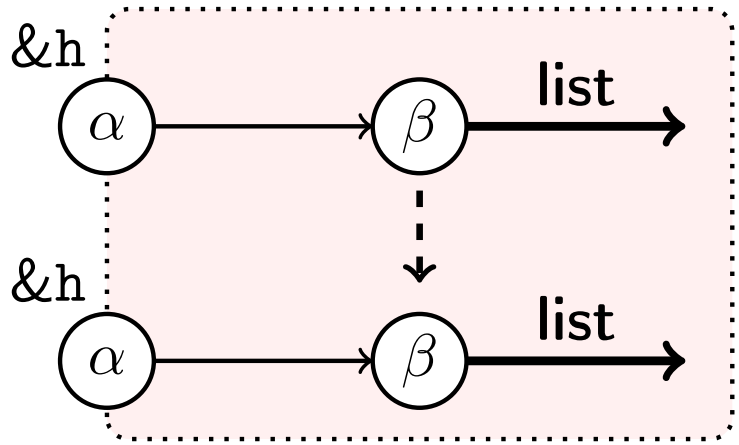


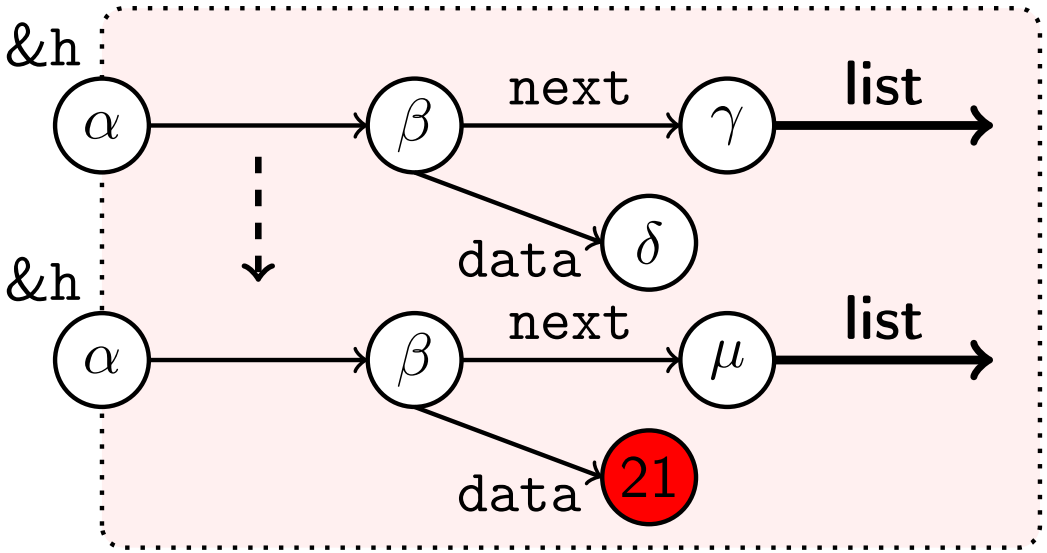




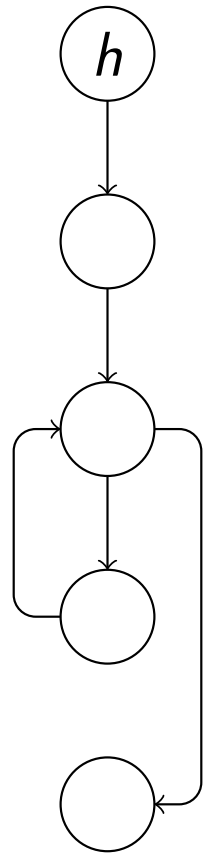
Unfolding over the Transform into Relation

Example: `h->data` := 21;



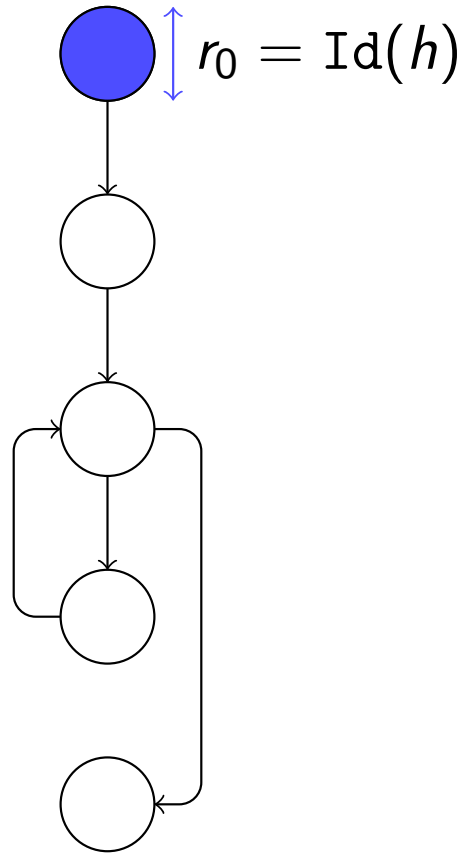


Analysis Algorithm: Forward Abstract Interpretation



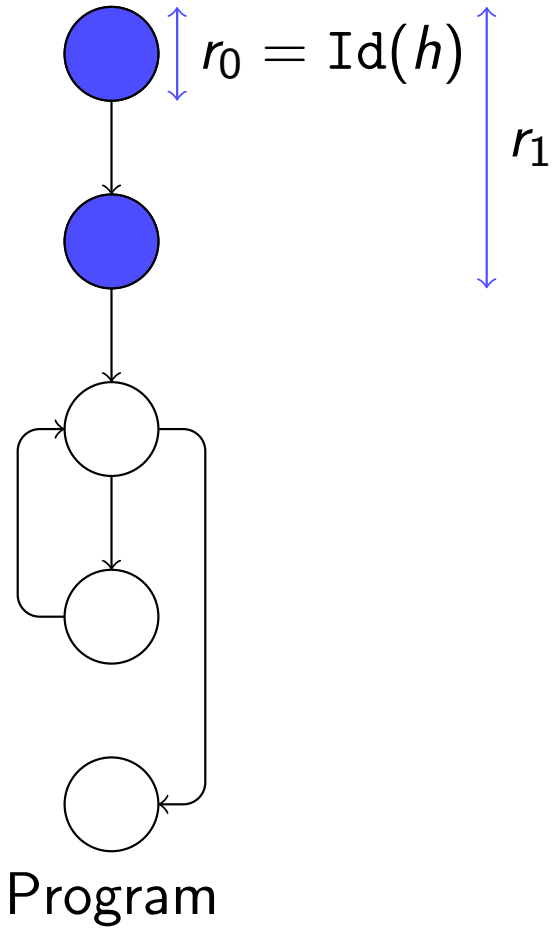
Program

Analysis Algorithm: Forward Abstract Interpretation

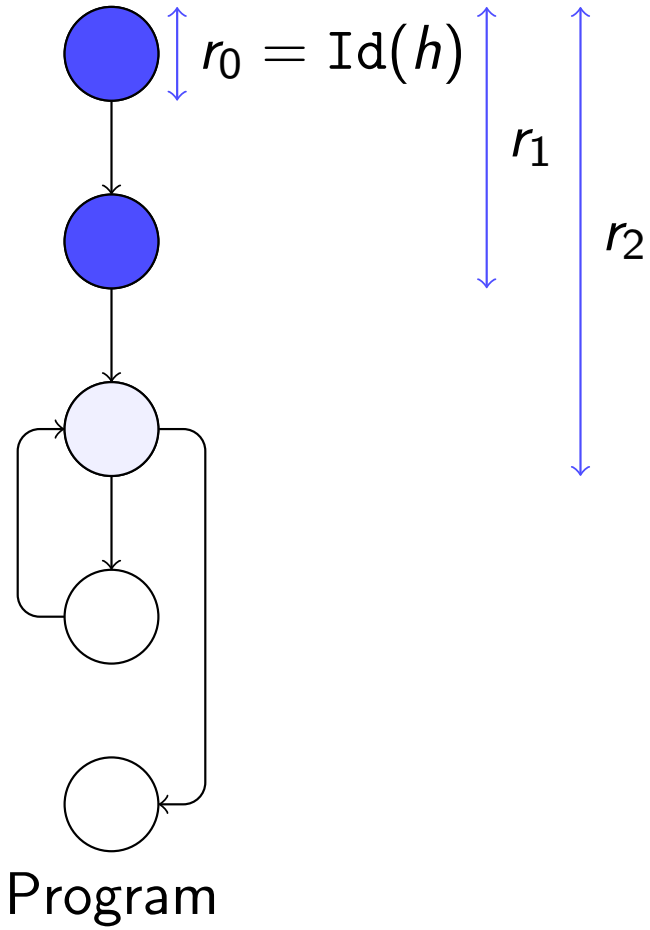


Program

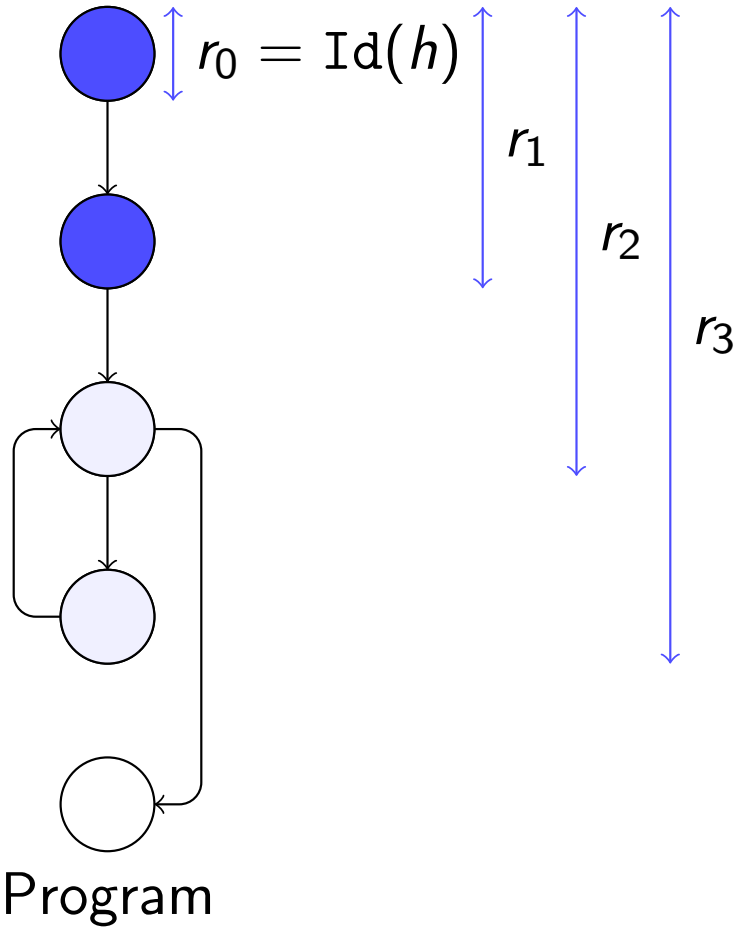
Analysis Algorithm: Forward Abstract Interpretation



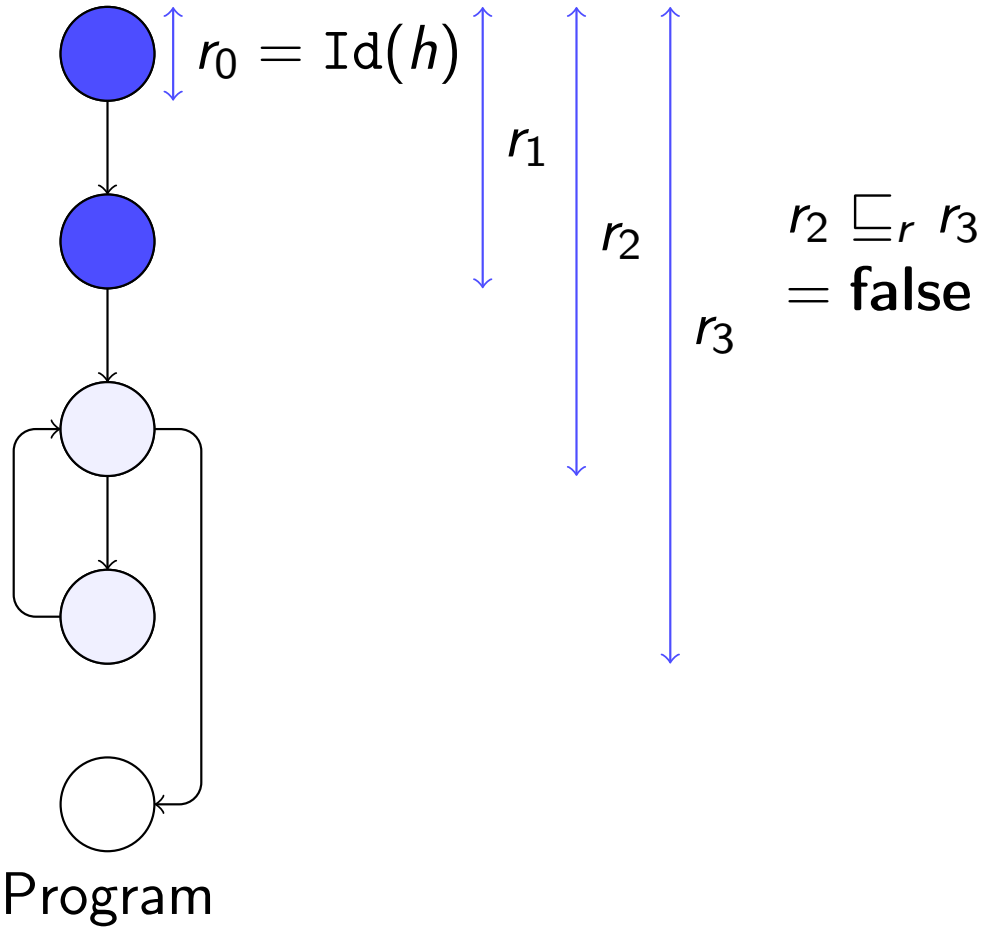
Analysis Algorithm: Forward Abstract Interpretation



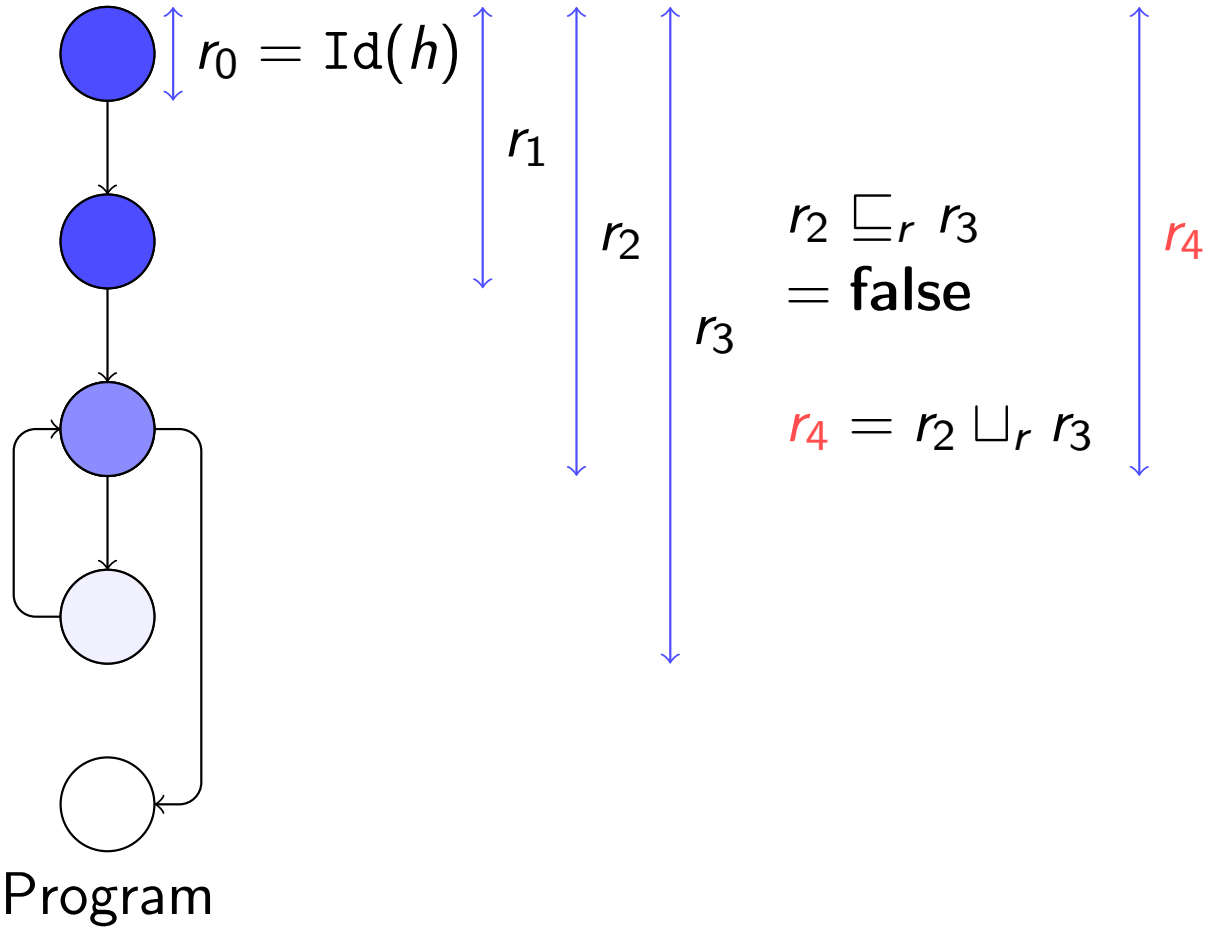
Analysis Algorithm: Forward Abstract Interpretation



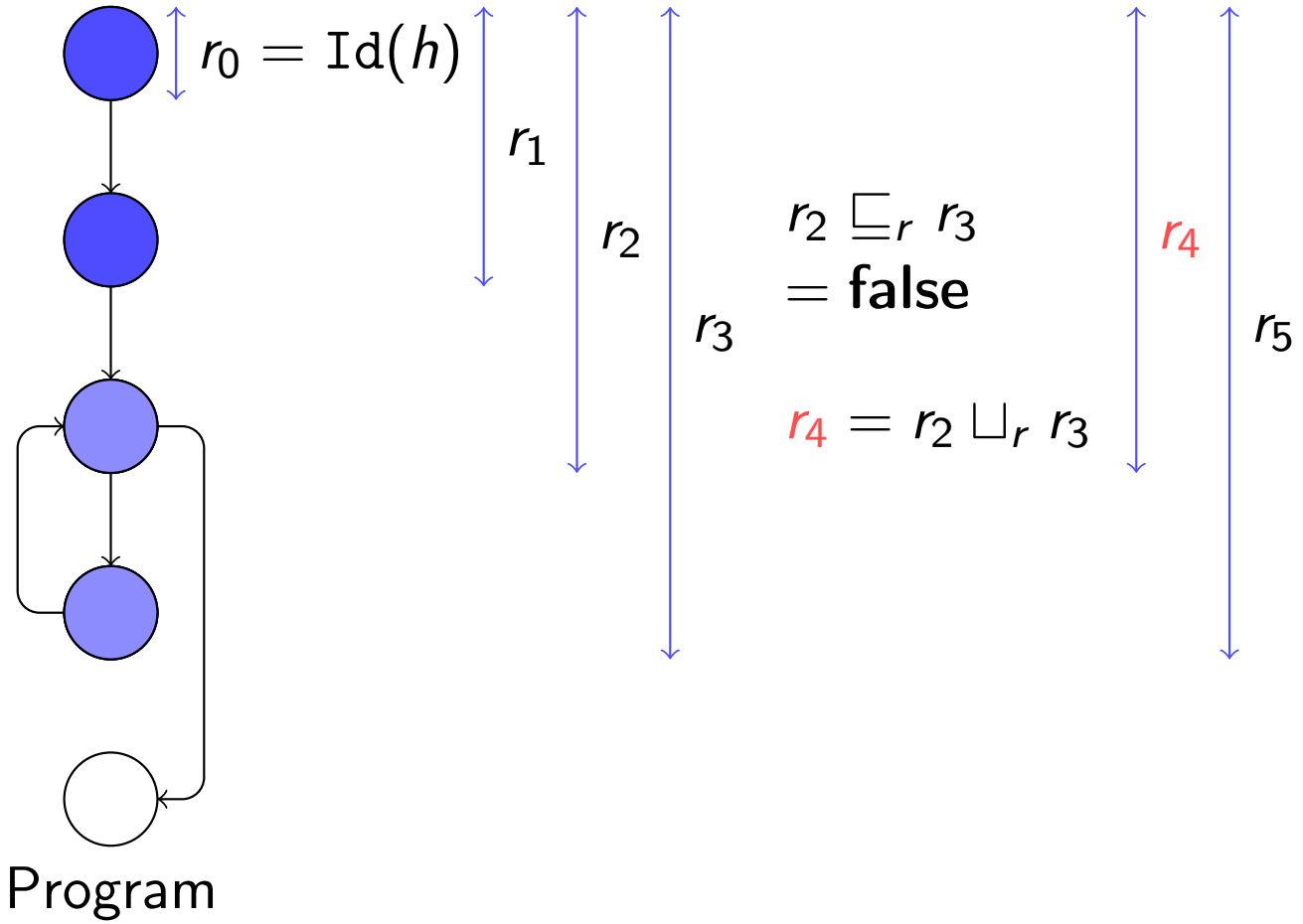
Analysis Algorithm: Forward Abstract Interpretation



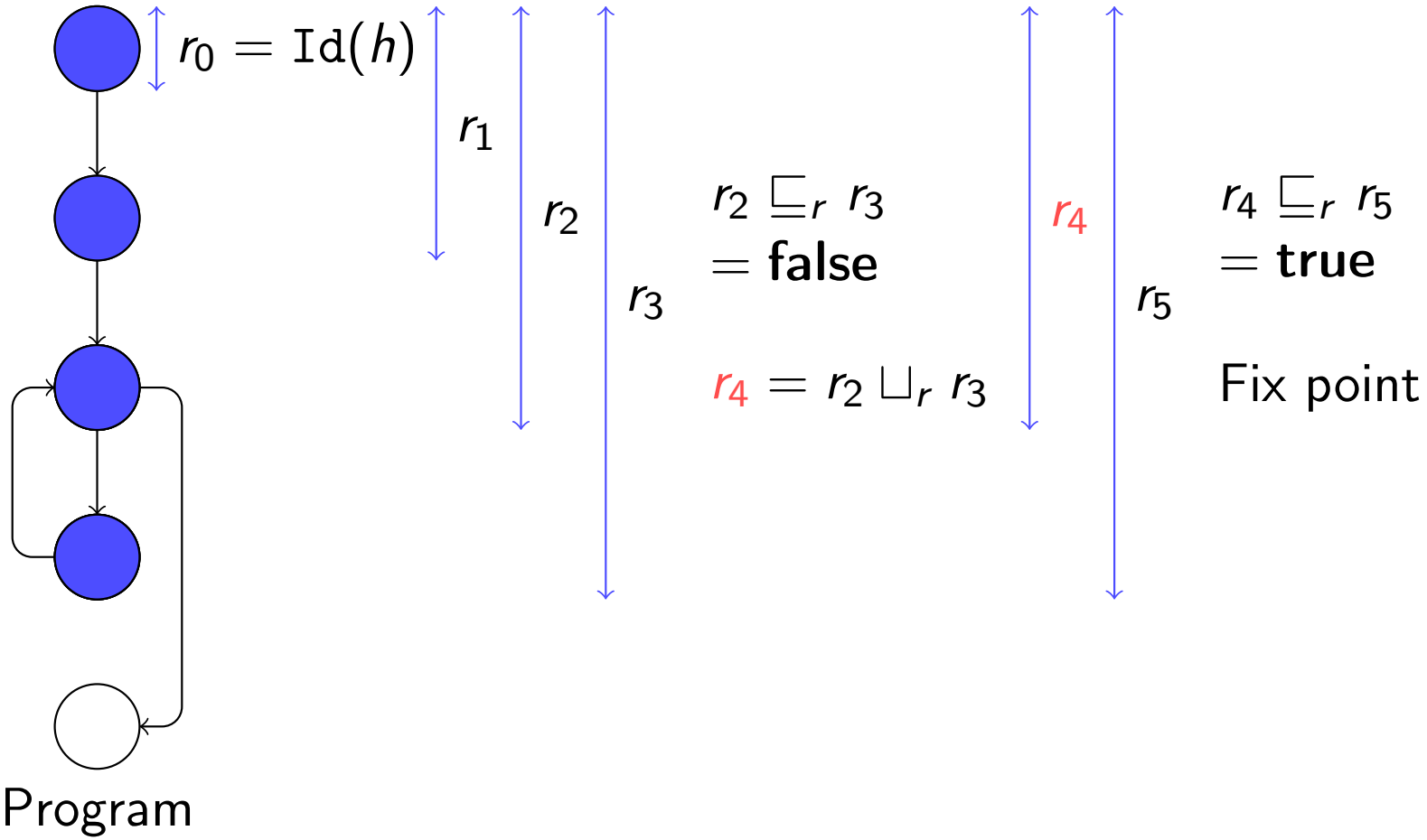
Analysis Algorithm: Forward Abstract Interpretation



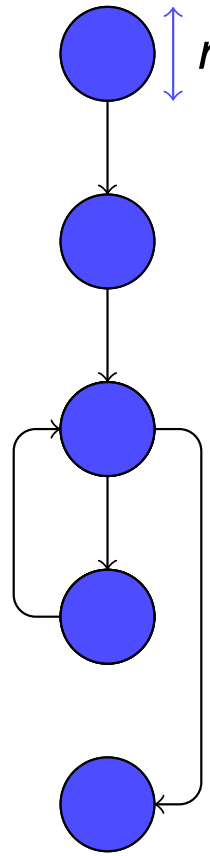
Analysis Algorithm: Forward Abstract Interpretation



Analysis Algorithm: Forward Abstract Interpretation



Analysis Algorithm: Forward Abstract Interpretation



Program

$r_0 = \mathtt{Id}(h)$

r_1

r_2

r_3

$r_2 \sqsubseteq_r r_3$
$= \textbf{false}$

$r_4 = r_2 \sqcup_r r_3$

r_4

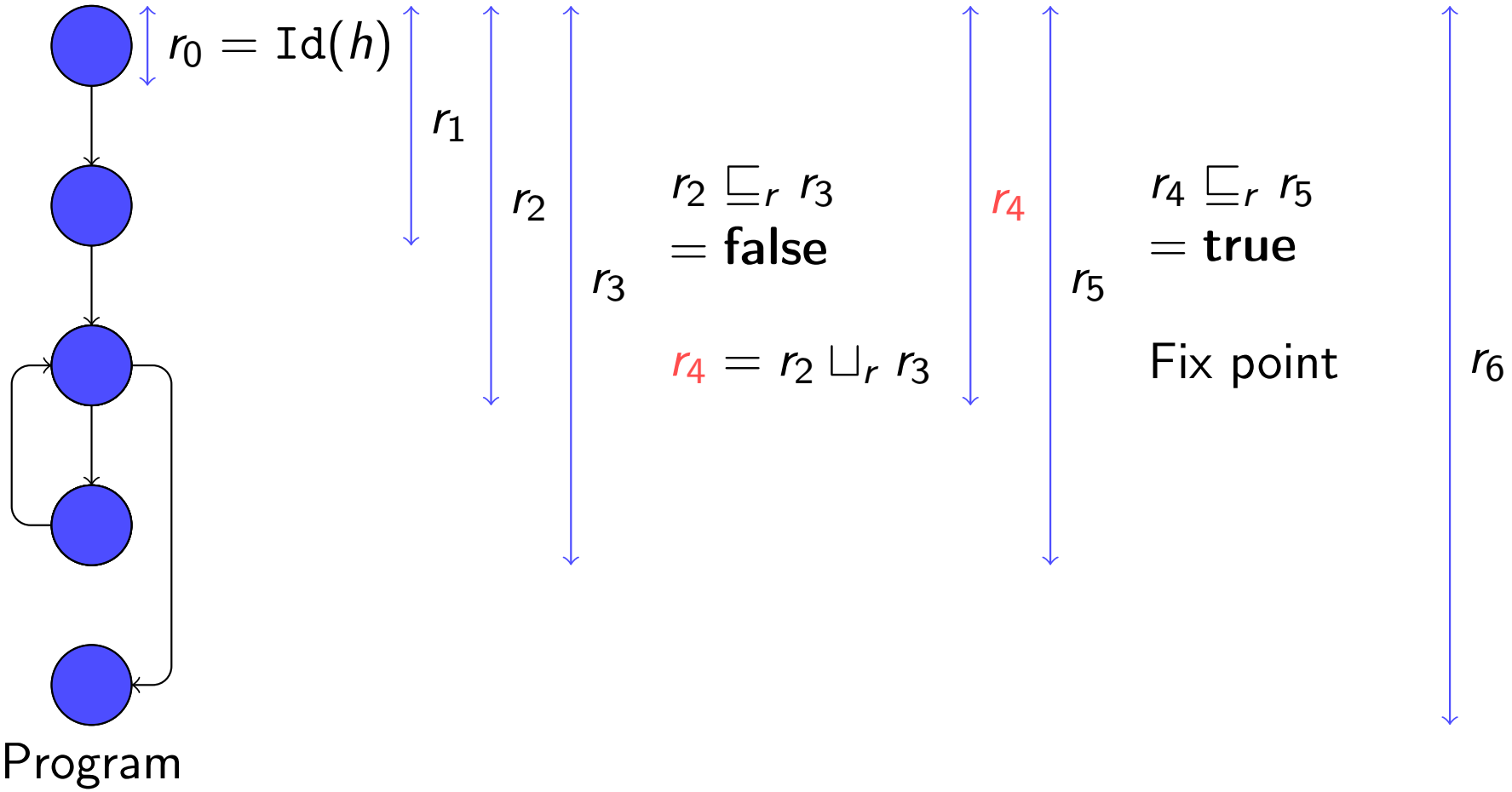
r_5

$r_4 \sqsubseteq_r r_5$
$= \textbf{true}$

Fix point

r_6

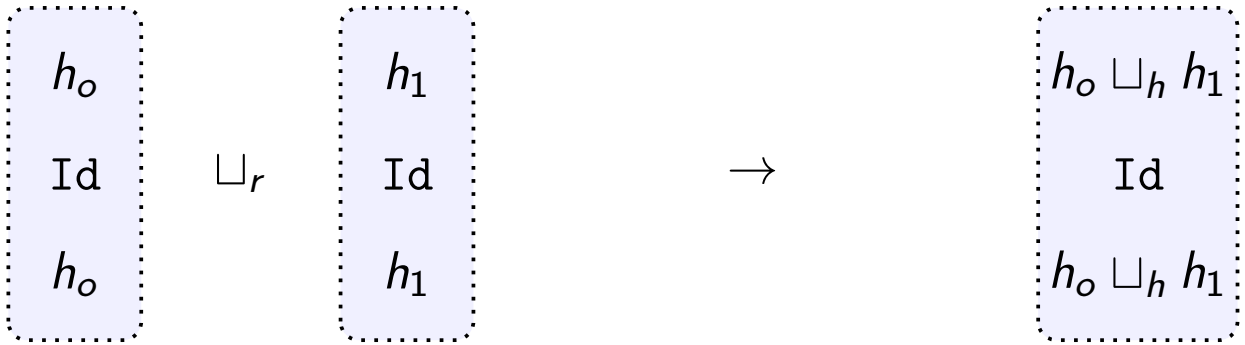
Analysis Algorithm: Forward Abstract Interpretation



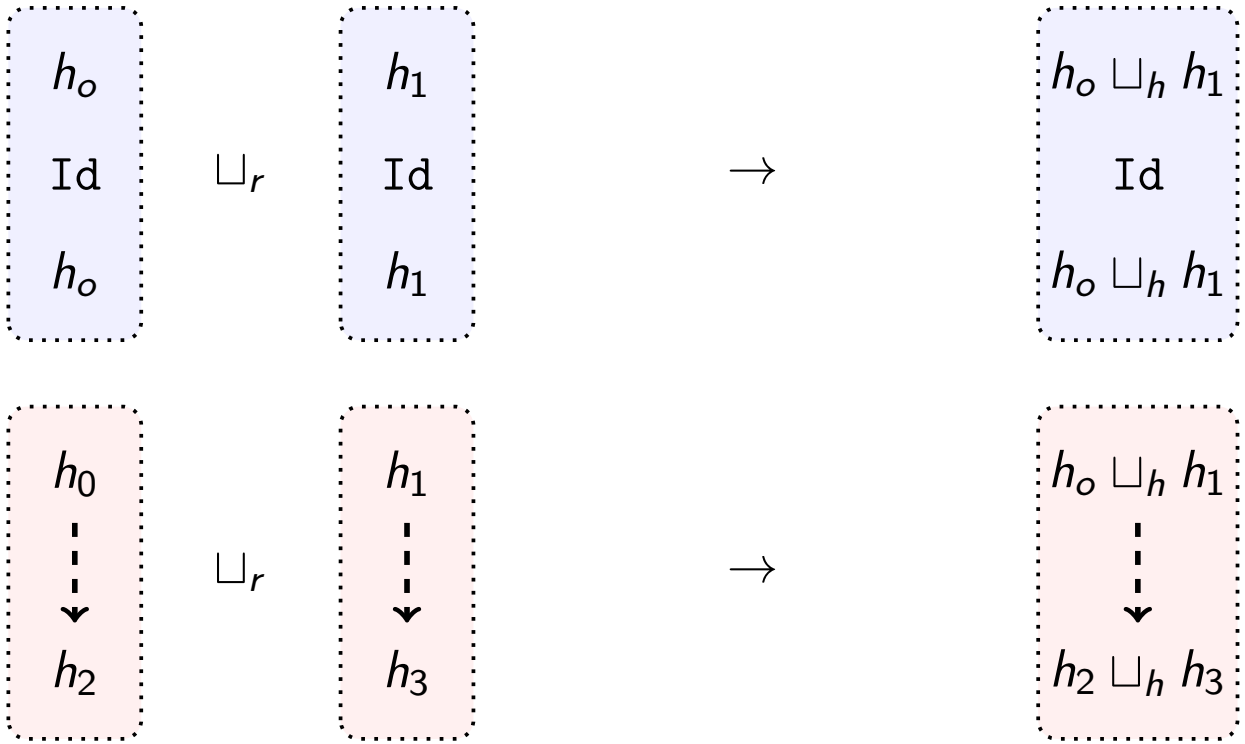
Theorem: The analysis is sound

Transfer functions, $\sqsubseteq$ and $\sqcup$ are all sound $\Rightarrow$ Analysis is sound $\square$

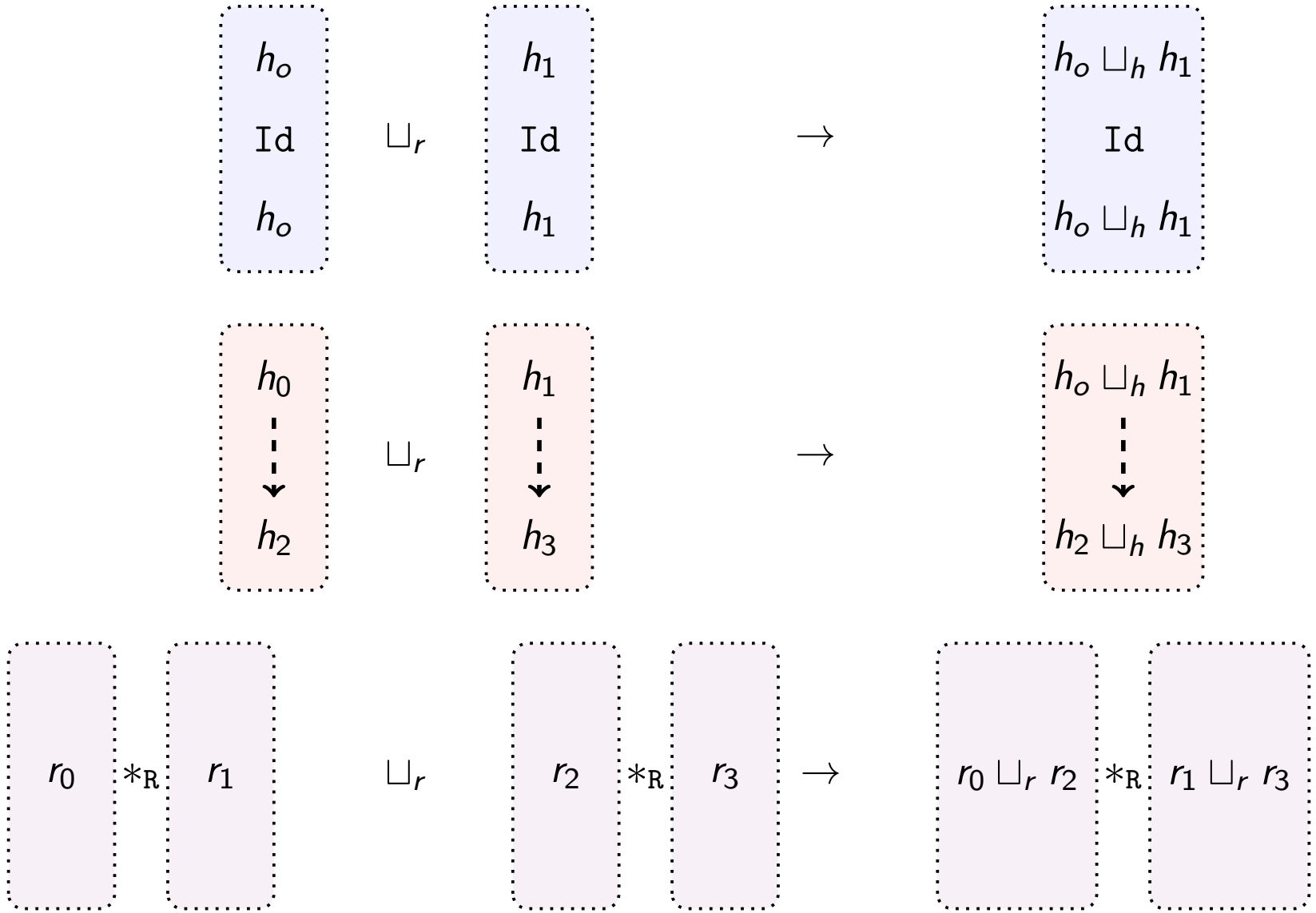
Join/Widening Rules (1/2)



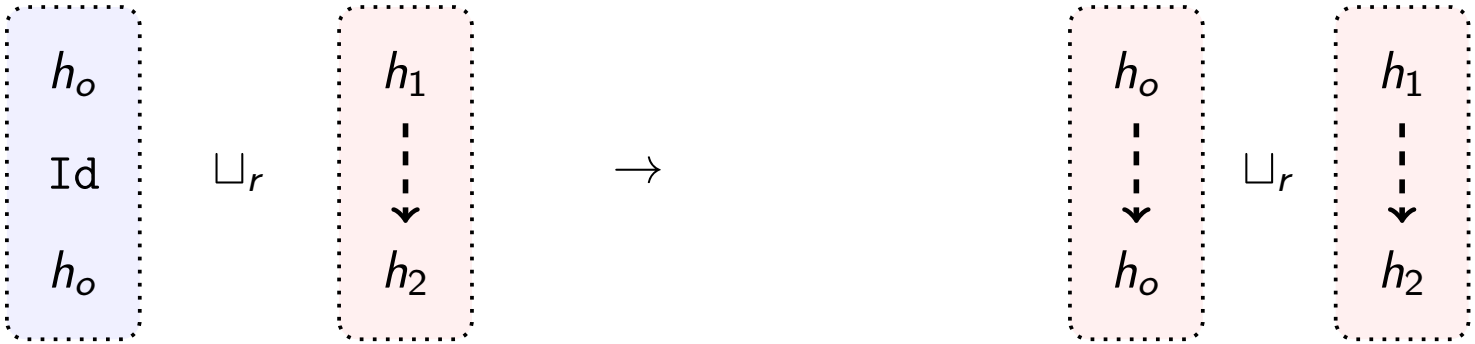
Join/Widening Rules (1/2)



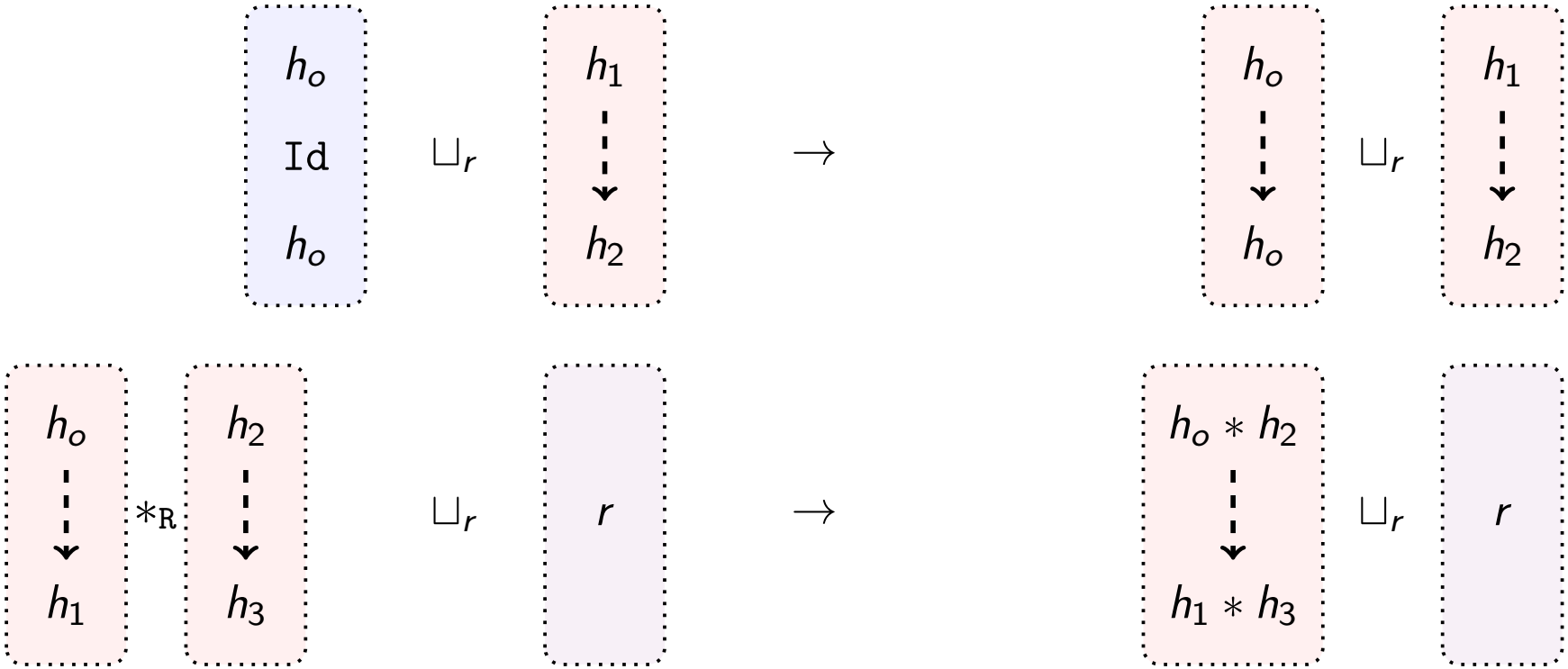
Join/Widening Rules (1/2)



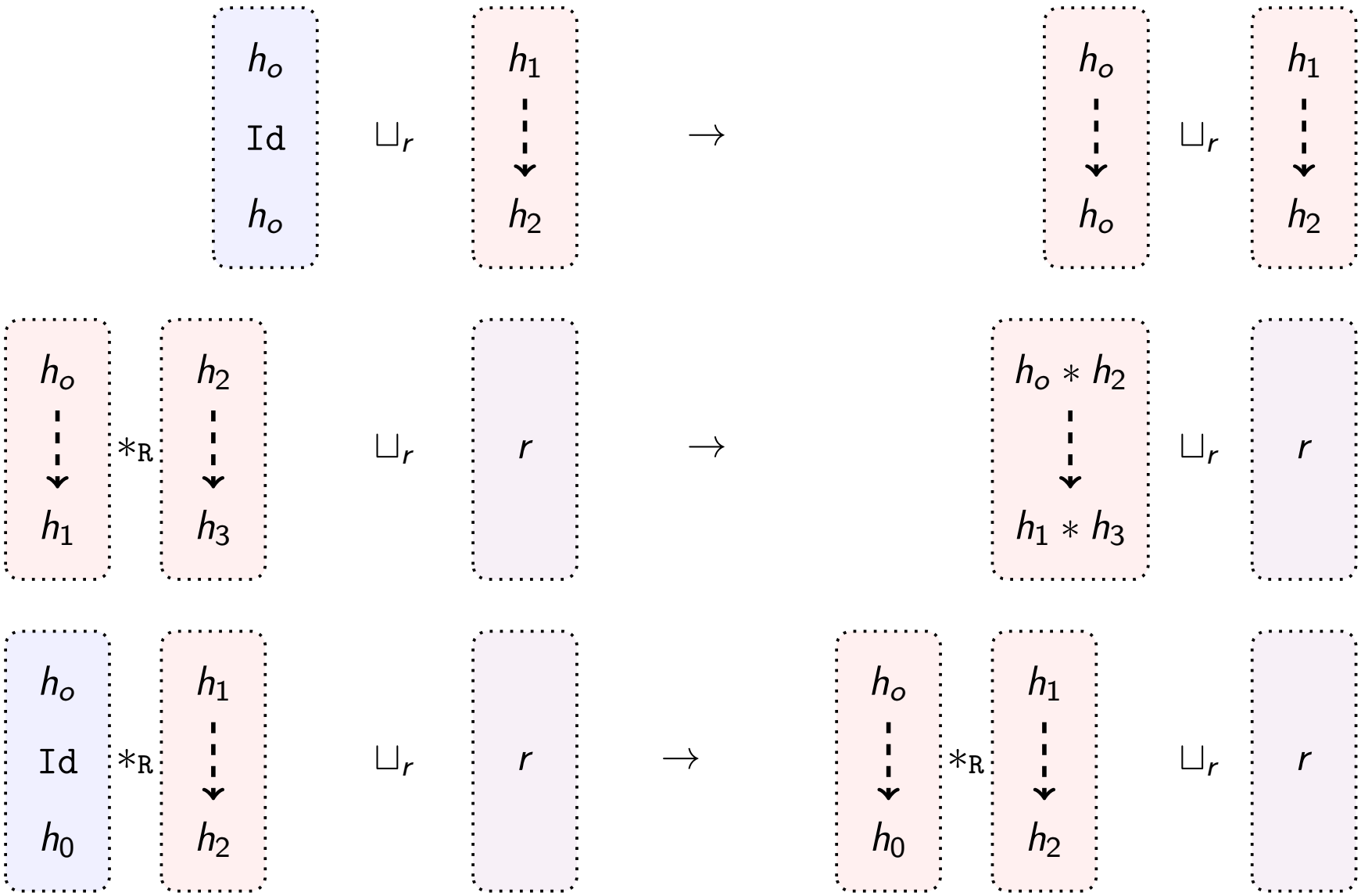
Join/Widening Rules (2/2)



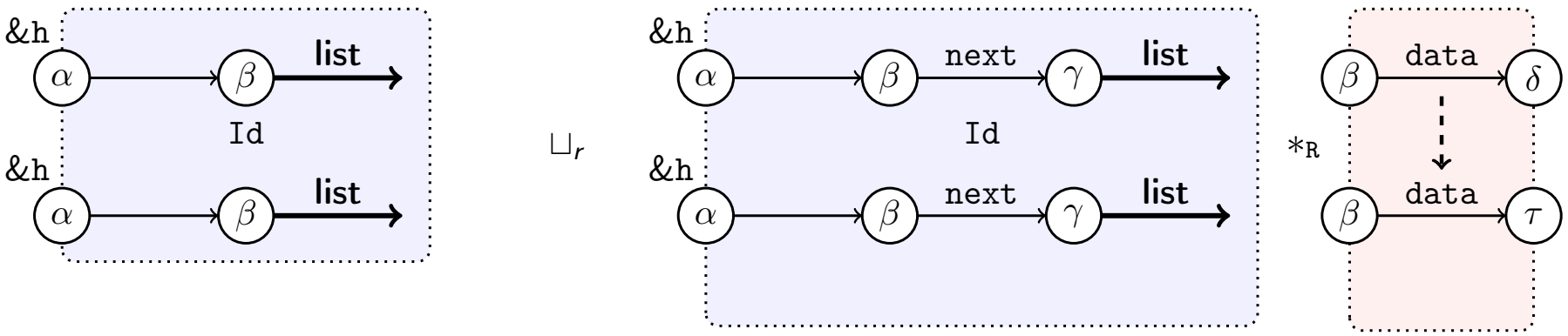
Join/Widening Rules (2/2)



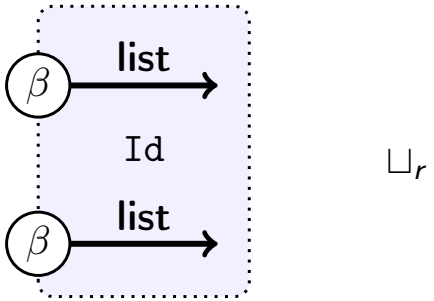
Join/Widening Rules (2/2)



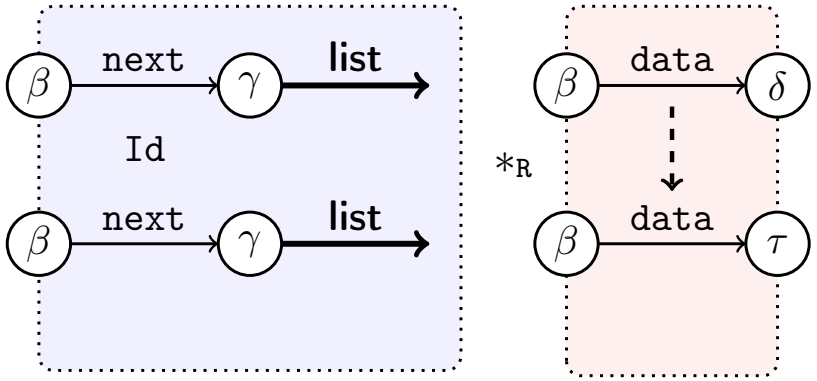
Join/Widening Example

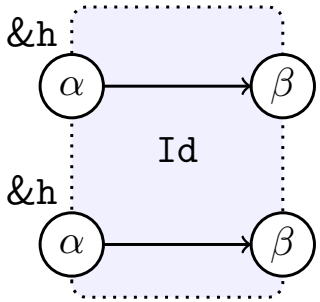


Join/Widening Example



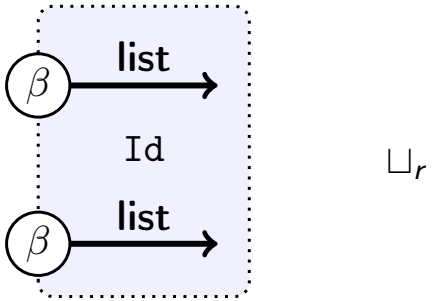
$\sqcup_r$



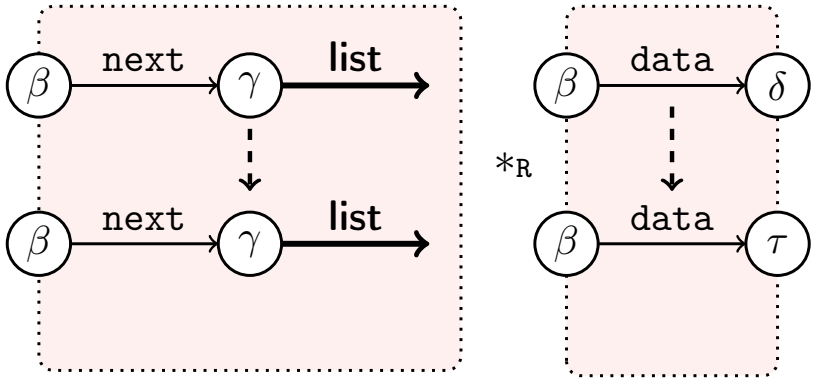


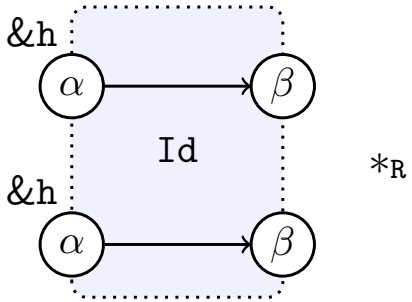
$*_R$

Join/Widening Example



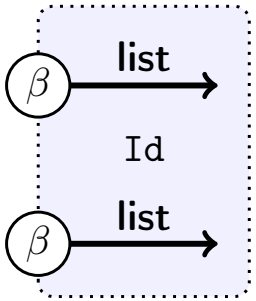
$\sqcup_r$



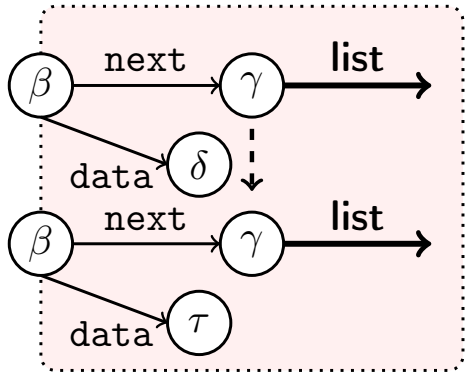


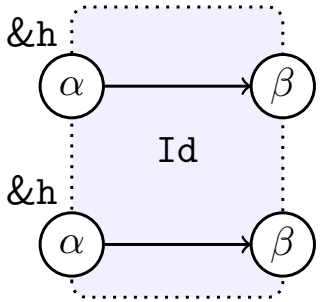
$*_R$

Join/Widening Example



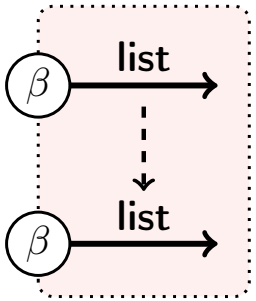
$\sqcup_r$



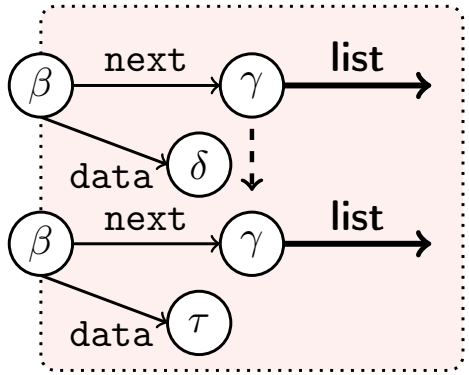


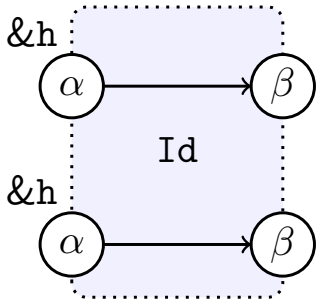
$*_R$

Join/Widening Example



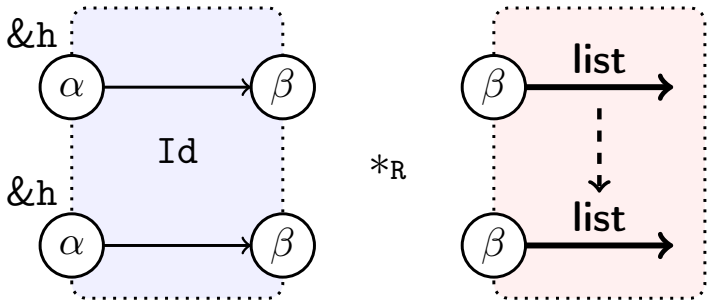
$\sqcup_r$





$*_R$

Join/Widening Example



Composition

- Definition:
  - $r_1 \circ r_2 = \{(h_i, h_o) \mid \exists h, (h_i, h) \in r_1 \wedge (h, h_o) \in r_2\}$
- At each function call, make the analysis compositional using relations as **function summaries**
- Idea:
  - $[h_i \dashrightarrow h] \circ [h \dashrightarrow h_o] = [h_i \dashrightarrow h_o]$
  - $r \circ \mathtt{Id}(h) = r$
  - $(r_0 *_{\mathtt{R}} r_1) \circ (r_2 *_{\mathtt{R}} r_3) = (r_0 \circ r_2) *_{\mathtt{R}} (r_1 \circ r_3)$
- Minimal loss of precision
- In practice more complex (requires $\sqcap$, the over-approximation of intersection)

Compositional Analysis Algorithm

```
1  void f() {
2     ...
3     g();
4     ...
5     g();
6     ...
7  }
```

1. At line 3, current relation in `f`: $[h_1 \dashrightarrow h_2]$

Compositional Analysis Algorithm

```
1  void f() {
2     ...
3     g();
4     ...
5     g();
6     ...
7  }
```

1. At line 3, current relation in `f`: $[h_1 \dashrightarrow h_2]$
2. Start the analysis of g with $\texttt{Id}(h_2)$ and obtain $[h_2 \dashrightarrow h_3]$

Compositional Analysis Algorithm

```
1  void f() {
2     ...
3     g();
4     ...
5     g();
6     ...
7  }
```

1. At line 3, current relation in `f`: $[h_1 \dashrightarrow h_2]$
2. Start the analysis of g with $\texttt{Id}(h_2)$ and obtain $[h_2 \dashrightarrow h_3]$
3. Save $[h_2 \dashrightarrow h_3]$ as summary for g

Compositional Analysis Algorithm

```
1  void f() {
2     ...
3     g();
4     ...
5     g();
6     ...
7  }
```

1. At line 3, current relation in `f`: $[h_1 \dashrightarrow h_2]$
2. Start the analysis of g with $\mathtt{Id}(h_2)$ and obtain $[h_2 \dashrightarrow h_3]$
3. Save $[h_2 \dashrightarrow h_3]$ as summary for g
4. Apply $[h_1 \dashrightarrow h_2] \circ [h_2 \dashrightarrow h_3] = [h_1 \dashrightarrow h_3]$

Compositional Analysis Algorithm

```
1  void f() {
2     ...
3     g();
4     ...
5     g();
6     ...
7  }
```

1. At line 3, current relation in `f`: $[h_1 \dashrightarrow h_2]$
2. Start the analysis of g with $\mathtt{Id}(h_2)$ and obtain $[h_2 \dashrightarrow h_3]$
3. Save $[h_2 \dashrightarrow h_3]$ as summary for g
4. Apply $[h_1 \dashrightarrow h_2] \circ [h_2 \dashrightarrow h_3] = [h_1 \dashrightarrow h_3]$
5. At line 5, current relation in `f`: $[h_1 \dashrightarrow h_n]$

Compositional Analysis Algorithm

```
1  void f() {
2     ...
3     g();
4     ...
5     g();
6     ...
7  }
```

1. At line 3, current relation in `f`: $[h_1 \dashrightarrow h_2]$
2. Start the analysis of g with $\mathtt{Id}(h_2)$ and obtain $[h_2 \dashrightarrow h_3]$
3. Save $[h_2 \dashrightarrow h_3]$ as summary for g
4. Apply $[h_1 \dashrightarrow h_2] \circ [h_2 \dashrightarrow h_3] = [h_1 \dashrightarrow h_3]$
5. At line 5, current relation in `f`: $[h_1 \dashrightarrow h_n]$
6. - if $h_n \sqsubseteq_h h_2$, Apply $[h_1 \dashrightarrow h_n] \circ [h_2 \dashrightarrow h_3]$

Compositional Analysis Algorithm

```
1  void f() {
2     ...
3     g();
4     ...
5     g();
6     ...
7  }
```

1. At line 3, current relation in `f`: $[h_1 \dashrightarrow h_2]$
2. Start the analysis of g with $\texttt{Id}(h_2)$ and obtain $[h_2 \dashrightarrow h_3]$
3. Save $[h_2 \dashrightarrow h_3]$ as summary for g
4. Apply $[h_1 \dashrightarrow h_2] \circ [h_2 \dashrightarrow h_3] = [h_1 \dashrightarrow h_3]$
5. At line 5, current relation in `f`: $[h_1 \dashrightarrow h_n]$
6. - if $h_n \sqsubseteq_h h_2$, Apply $[h_1 \dashrightarrow h_n] \circ [h_2 \dashrightarrow h_3]$
   - otherwise, **forget** $[h_2 \dashrightarrow h_3]$ and **re-analyze** g from $\texttt{Id}(h_n \sqcup_h h_2)$

Prototype and Experimental Evaluation

- Static analyzer of C programs
- Implemented as a Frama-C plug-in (16 000 lines of OCaml)
- Supports both non-relational and relational analysis, parametrized by inductive definitions (linked lists, binary trees, emacs Lisp Objects...)
- Goals:
  - Check if the computed relations were the expected
  - Evaluate the practicality and scalability of the analysis
  - Assess the cost and benefits of a relational analysis
- Kind of analyzed programs
  - Small functions
  - War (card game, $\approx$ 250 lines of C)
  - Real program (part of emacs 25.3, $\approx$ 2000 lines of C)

Experimental Evaluation of small functions

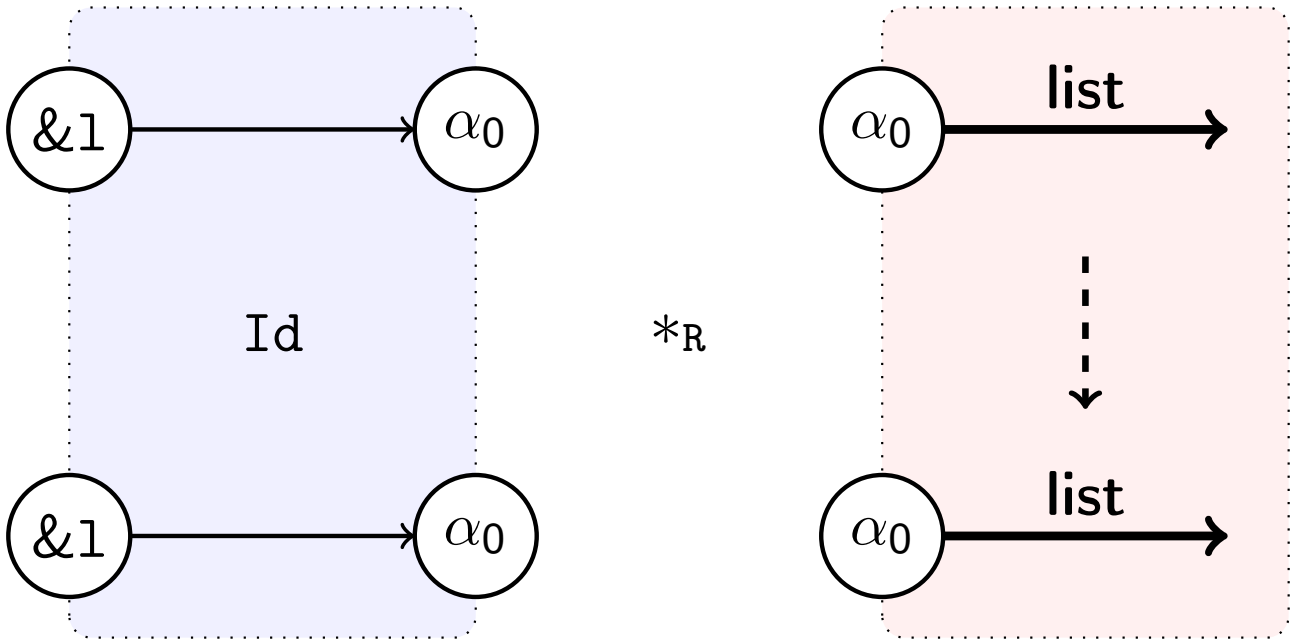
Structure	Function	Time (in ms)		Relational
		Reach.	Relat.	Property
sll	allocation	0.53	1.27	yes
sll	deallocation	0.34	0.99	yes
sll	traversal	0.53	0.83	yes
sll	insertion (head)	0.32	0.33	yes
sll	insertion (random pos)	1.98	2.75	yes
sll	insertion (random)	2.33	3.94	yes
sll	reverse (in place)	0.52	2.36	partial
sll	map	0.66	1.17	partial
tree	allocation	0.94	2.21	yes
tree	search	1.06	1.76	yes

Experimental Evaluation of small functions

Structure	Function	Time (in ms)		Relational
		Reach.	Relat.	Property
sll	allocation	0.53	1.27	yes
sll	deallocation	0.34	0.99	yes
sll	traversal	0.53	0.83	yes
sll	insertion (head)	0.32	0.33	yes
sll	insertion (random pos)	1.98	2.75	yes
sll	insertion (random)	2.33	3.94	yes
sll	reverse (in place)	0.52	2.36	partial
sll	map	0.66	1.17	partial
tree	allocation	0.94	2.21	yes
tree	search	1.06	1.76	yes

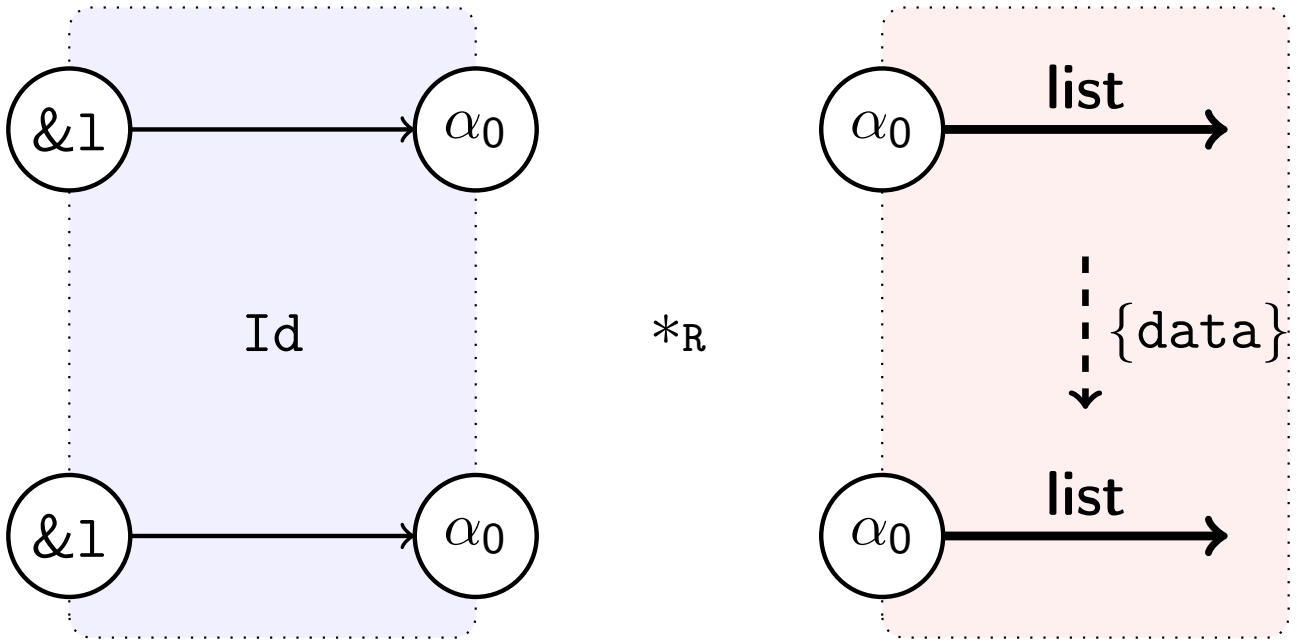
Relational Heap Predicates

```
void list_map(list *l) {
  list *c = l;
  while(c != NULL) {
    c->data++;
    c = c->next;
  }
}
```



Relational Heap Predicates

```
void list_map(list *l) {
  list *c = l;
  while(c != NULL) {
    c->data++;
    c = c->next;
  }
}
```



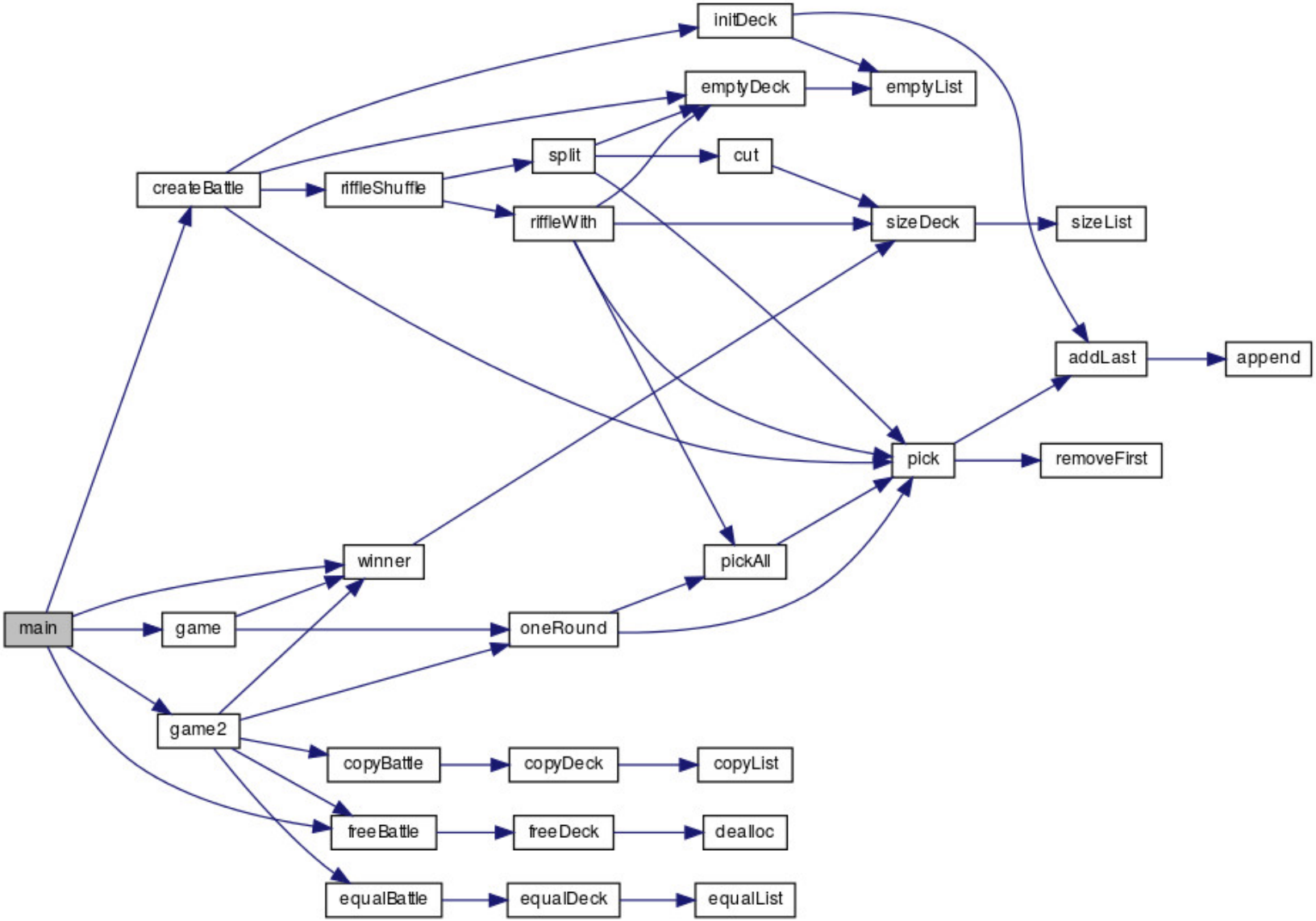
Experimental Evaluation of War, Data Structures

```
typedef struct list {
  int data;
  struct list *next;
} list;

typedef struct deck {
  list *pack_of_cards;
} deck;

typedef struct battle {
  int nbVals;
  deck *trick;
  deck *player1;
  deck *player2;
} battle;
```

Experimental Evaluation of War, static call graph



Experimental Evaluation of War, time in s for 1000 analyzes

Structure	Function	State (in-line)	Relation (composition)
List	removeFirst	0.248	0.268
List	append	0.608	1.124
List	addLast	0.872	1.584
Deck	initDeck	3.888	4.124
Deck	sizeDeck	0.572	0.808
Deck	pick	1.364	2.348
Deck	pickAll	1.616	3.604
Deck	split	4.932	6.972
Deck	riffleWith	15.064	14.548
Deck	riffleShuffle	53.024	22.992
Battle	createBattle	78.816	30.132
Battle	oneRound	28.816	22.736
Battle	winner	3.088	2.412
Battle	copyBattle	4.676	2.852
Battle	equalBattle	13.756	8.736
Battle	game	39.280	26.876
Battle	game2	185.252	51.396
Battle	main (with game)	188.588	78.962
Battle	main (with game2)	5319.648	864.282

Experimental Evaluation of Emacs: Lisp Objects

- Lisp Object (set of primitive types of Lisp):
  - Symbol
  - String
  - Integer
  - Cons pair
  - Vector
  - ...

$$
\begin{array}{lcl}
\textbf{lisp\_object}(\alpha) & := & \textbf{emp} \\
& & \\
& \vee & \alpha \cdot \texttt{car} \mapsto \beta * \textbf{lisp\_object}(\beta) * \\
& & \alpha \cdot \texttt{cdr} \mapsto \gamma * \textbf{lisp\_object}(\gamma) \\
& & \\
& \vee & \ldots
\end{array}
$$

Experimental Evaluation of Emacs: Lisp Objects

- Lisp Object (set of primitive types of Lisp):
  - Symbol
  - String
  - Integer
  - Cons pair
  - Vector
  - ...

$$
\begin{array}{lcl}
\textbf{lisp\_object}(\alpha) & := & \textbf{emp} \\
& & \\
& \vee & \alpha \cdot \texttt{car} \mapsto \beta * \textbf{lisp\_object}(\beta) * \\
& & \alpha \cdot \texttt{cdr} \mapsto \gamma * \textbf{lisp\_object}(\gamma) \\
& & \\
& \vee & \ldots
\end{array}
$$

Experimental Evaluation of Emacs: Results

- Analysis of the function `Fx_show_tip` (src/xfns.c)
- $\approx$ 2000 LOC after pre-processing
- Without relation: analysis stopped after 4 hours
- With relation and composition : 14.988 seconds!
- No loss of precision

Conclusion

- Automatic inference of "intuitive" relational properties on programs (absence of modification, of allocation...)
- Enabler for efficient and precise shape analysis of large benchmarks
- Data-structure agnostic
- Future Works:
  - Handle recursive functions (without manual annotations)
  - Combine with recent progresses in shape analysis
  - To be able to express data or structural relations between separated transformations

Thank you for your attention

Questions?