Relational Refinement Types for Higher-Order Shape Transformers

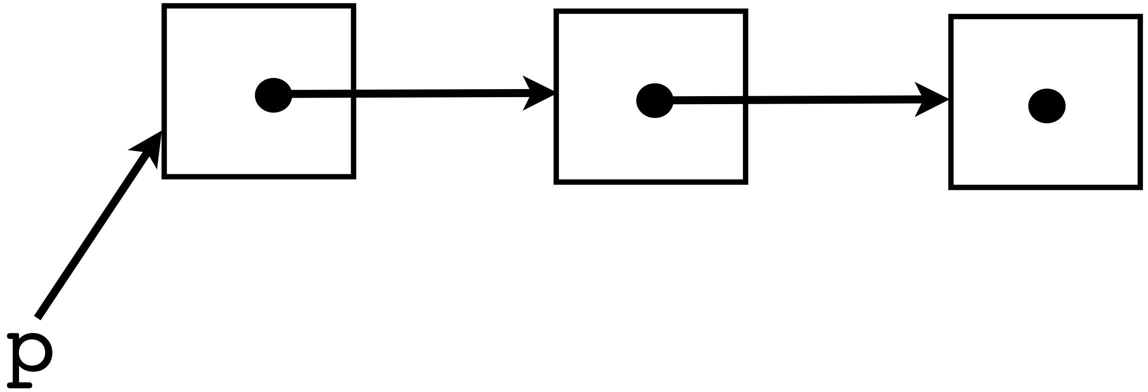
Suresh Jagannathan

Joint work with Gowtham Kaki

PURDUE UNIVERSITY

Shape Analysis

In imperative settings, shape analysis is concerned with discovering/verifying the shape of a pointer into memory

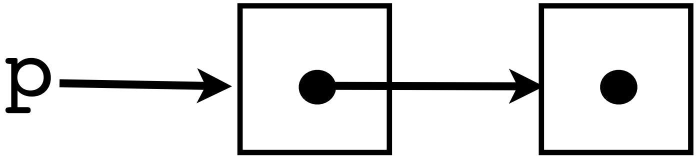


```
p = LinkedList
```

Shape Analysis for Functional (Typed) Programs

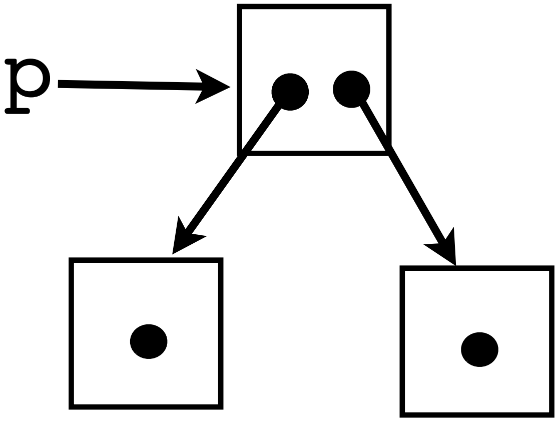
In functional languages, we have have types

```
p = Cons(.,Cons(., Nil))
```



p : α list

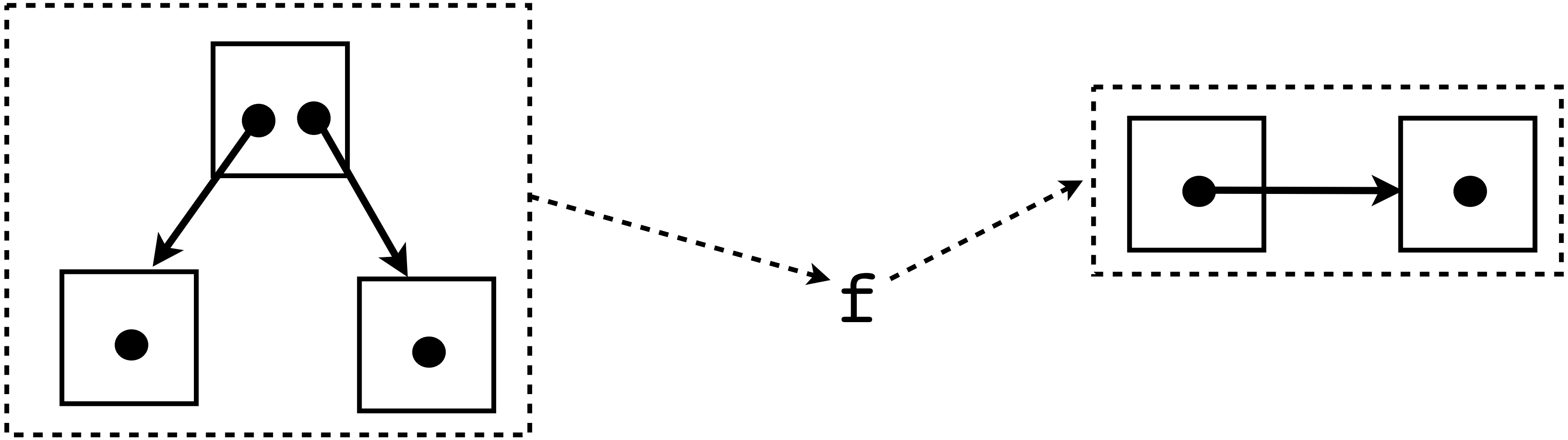
```
p = B(B(L,.,L),.,B(L,.,L)
```



p : α tree

Shape Analysis for Functional (Typed) Programs

In functional languages, we have have types



$$f : \alpha\ \texttt{tree} \longrightarrow \alpha\ \texttt{list}$$

Shape Analysis for Functional (Typed) Programs

In functional languages, we have have types

$$\texttt{f : } \alpha \ \texttt{tree} \longrightarrow \alpha \ \texttt{list}$$

Shape Analysis for Functional (Typed) Programs

In functional languages, we have have types

$$\texttt{f : } \alpha \texttt{ tree} \longrightarrow \alpha \texttt{ list}$$

Shape Analysis for Functional (Typed) Programs

In functional languages, we have have types

$$\texttt{f : } \alpha \texttt{ tree} \longrightarrow \alpha \texttt{ list}$$

How can we use types to express precise shape information?

$$\texttt{f : \{t:}\alpha\texttt{ tree\}} \longrightarrow \texttt{\{l:}\alpha\texttt{ list}|\varphi\texttt{\}}$$

$$\underbrace{\varphi}_{\text{type refinement predicate}} \Leftrightarrow \texttt{SomeShape(l)}\equiv\texttt{SomeOtherShape(t)}$$

Reasoning about shapes

Reasoning about shapes

- Inductively-defined algebraic datatypes are a key feature in modern programming languages
    - ★ Enable the expression of rich data structures - lists, trees, graphs, maps, etc.

Reasoning about shapes

- Inductively-defined algebraic datatypes are a key feature in modern programming languages
  - ★ Enable the expression of rich data structures - lists, trees, graphs, maps, etc.
- But, they also pose challenges for verification
  - ★ Recursive structure
  - ★ Important attributes are often not manifest in a constructor's signature
    - ✦ E.g., length, sorted-ness, height, balance, membership, ordering, dominance, symmetry, etc.
  - ★ Polymorphism and higher-order functions

Reasoning about shapes

- Inductively-defined algebraic datatypes are a key feature in modern programming languages
  - ★ Enable the expression of rich data structures - lists, trees, graphs, maps, etc.
- But, they also pose challenges for verification
  - ★ Recursive structure
  - ★Important attributes are often not manifest in a constructor's signature
    - ✦ E.g., length, sorted-ness, height, balance, membership, ordering, dominance, symmetry, etc.
  - ★ Polymorphism and higher-order functions
- Tension
  - ★ desire expressive specifications over the shape of data
  - ★ but want automated verification of their correctness

Example

```
rev : {l : ’a list} ⟶ {ν: ’a list | ν = rev’(l)}

fun rev [] = []
  | rev x::xs = concat (rev xs) [x]
```

Example

```
rev : {l : 'a list} ⟶ {ν: 'a list | ν = rev'(l)}

fun rev [] = []
  | rev x::xs = concat (rev xs) [x]
```

reasoning about rev' likely as complex as directly reasoning about rev

Example

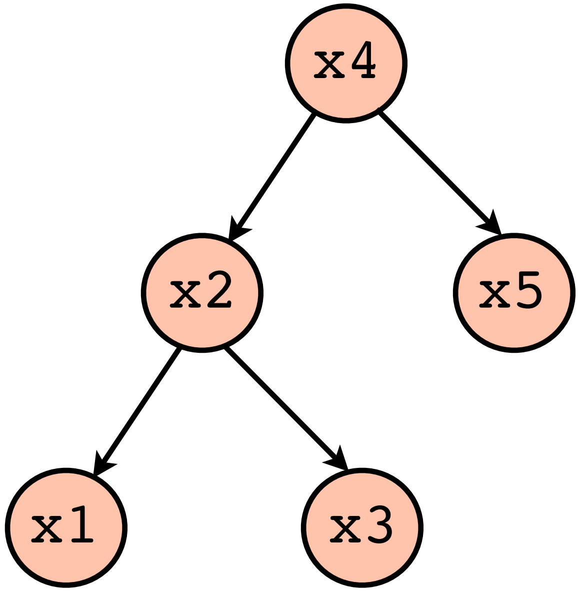
```
rev : {l : 'a list} ⟶ {ν: 'a list | ν = rev'(l)}

fun rev [] = []
  | rev x::xs = concat (rev xs) [x]
```

reasoning about rev' likely as complex as directly reasoning about rev

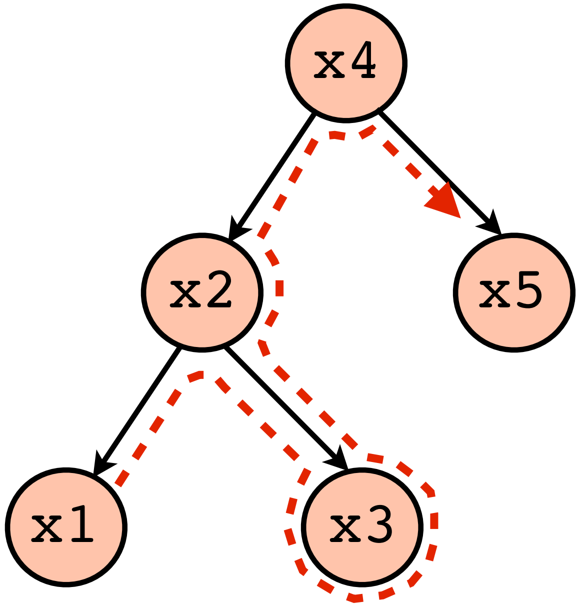
We want

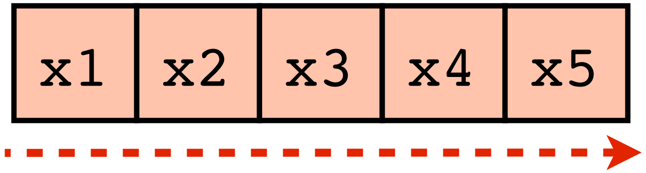
- ★ To reason structurally about the order of elements in the list
- ★ Without appealing to an operational definition of how that ordering is realized

Example



Example

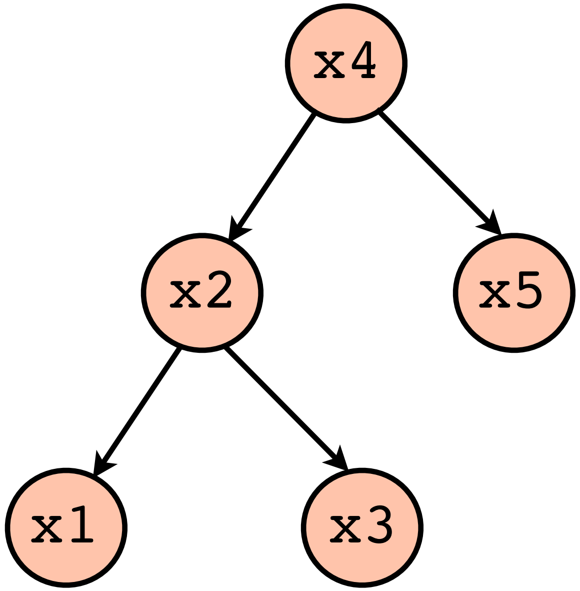




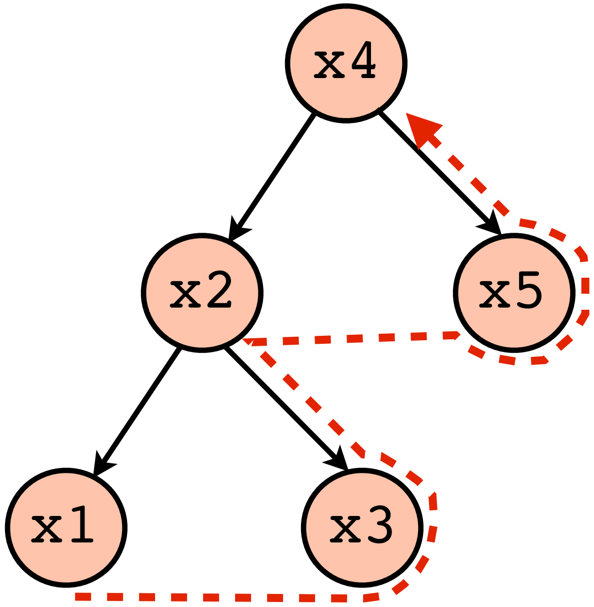
inOrder : {t:α tree} ⟶ {l:α list|φ}

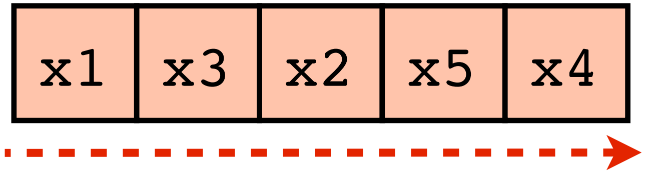
φ ⇔ forward-order(l)=in-order(t)

Post-Order



Post-Order

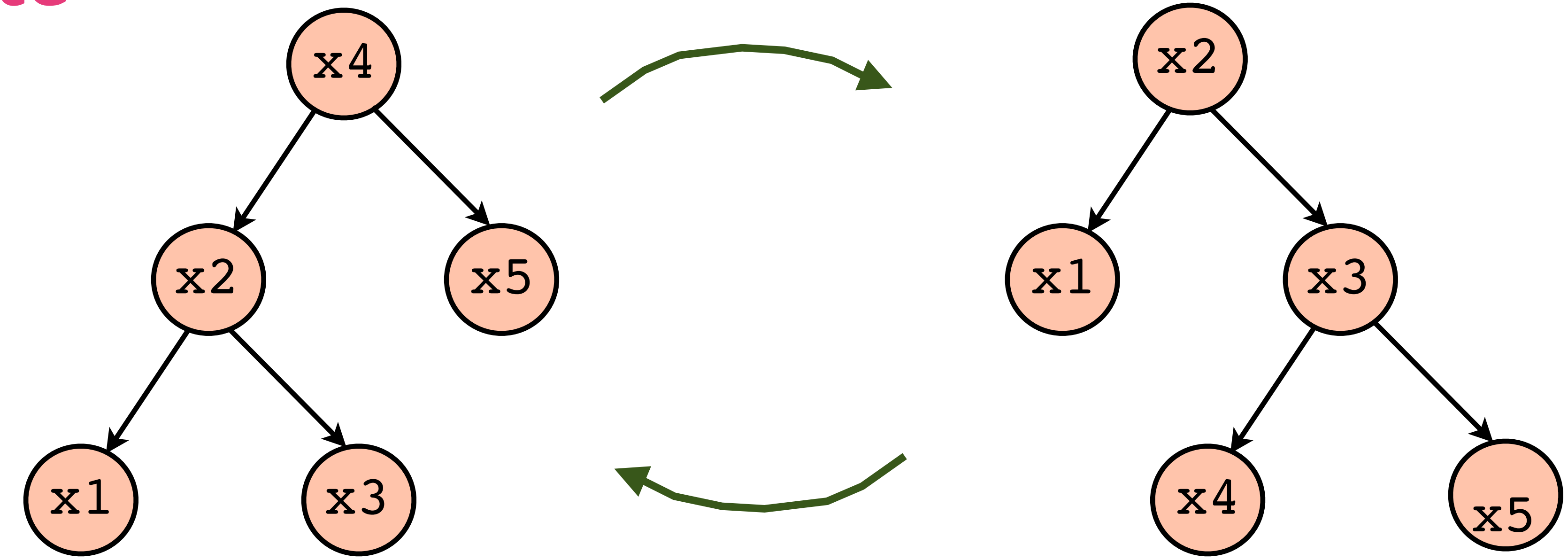




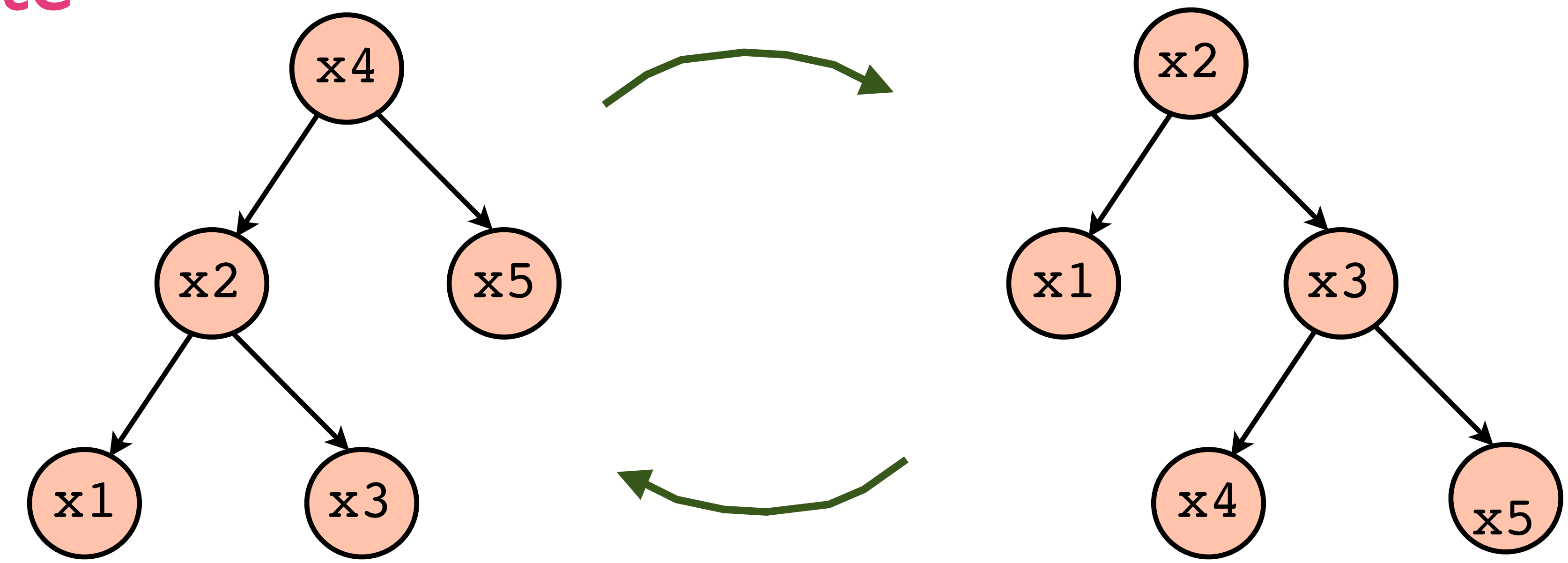
postOrder : {t:α tree} ⟶ {l:α list | φ}

φ ⇔ forward-order(l)= post-order(t)

Rotate



Rotate



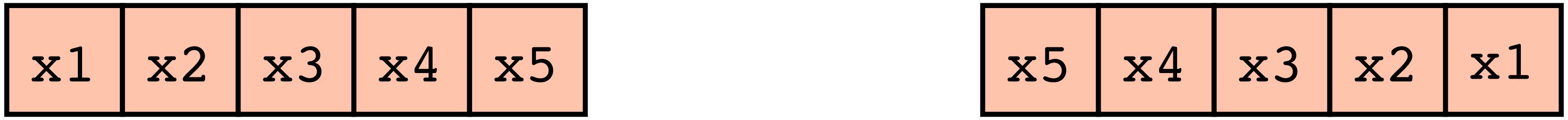
x1	x2	x3	x4	x5

x1	x4	x5	x3	x2

rotate : {t1:α tree} ⟶ {t2:α tree | φ}

φ ⇔ in-order(t1)= post-order(t2)

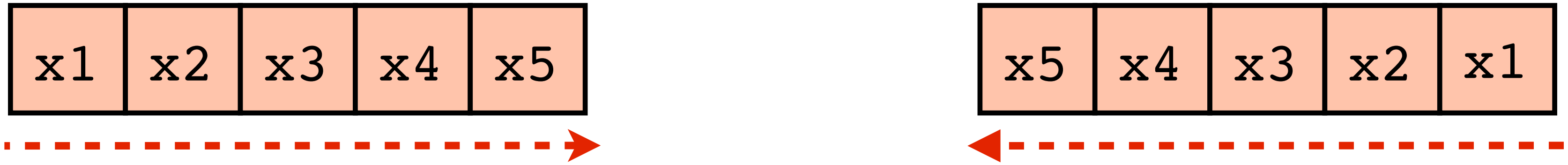
Reverse



x1	x2	x3	x4	x5

x5	x4	x3	x2	x1

Reverse



$$\mathtt{rev} : \{\mathtt{l1}{:}\alpha\ \mathtt{list}\} \longrightarrow \{\mathtt{l2}{:}\alpha\ \mathtt{list} \mid \varphi\}$$

$$\varphi \Leftrightarrow \mathtt{backward\text{-}order(l2)=forward\text{-}order(l1)}$$

We need ...

Type refinements (φ) to be predicates over an expressive language.

We need ...

Type refinements (φ) to be predicates over an expressive language.

Should serve as a common medium to express fine-grained shapes of data structures, such as `in-order`, `pre-order`, `post-order`, `forward-order`, and `backward-order`

Observe ...

What is common among `pre-order`, `post-order`, `forward-order`, and `backward-order`?

Observe ...

What is common among `pre-order`, `post-order`, `forward-order`, and `backward-order`?

All are orders

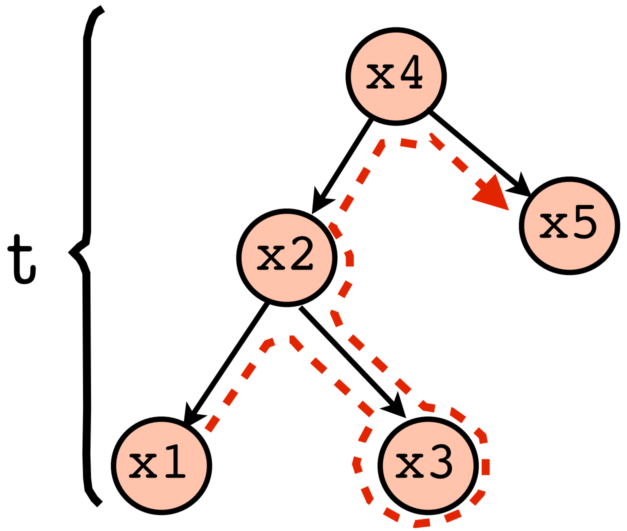
Observe ...

What is common among `pre-order`, `post-order`, `forward-order`, and `backward-order`?

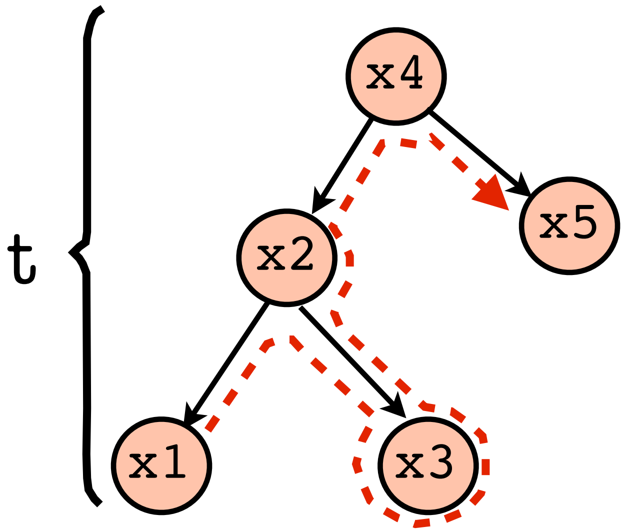
All are orders

Expressible as binary relations

For Example ...

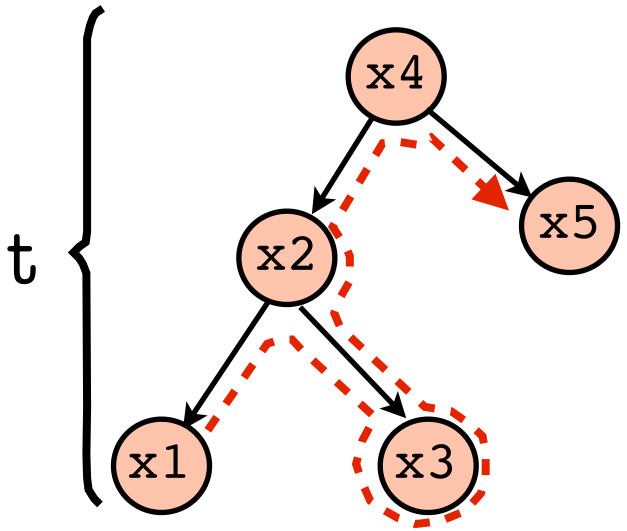


For Example ...



in-order of t is binary relation such that: in-order(x_i,x_j) ⇔ $i \leq j$

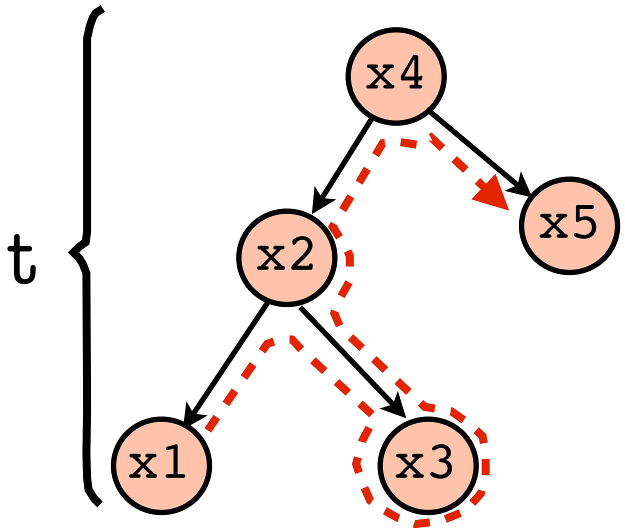
For Example ...



in-order of t is binary relation such that: $\texttt{in-order}(x_i,x_j) \Leftrightarrow i \leq j$

$R_{io}(t)$

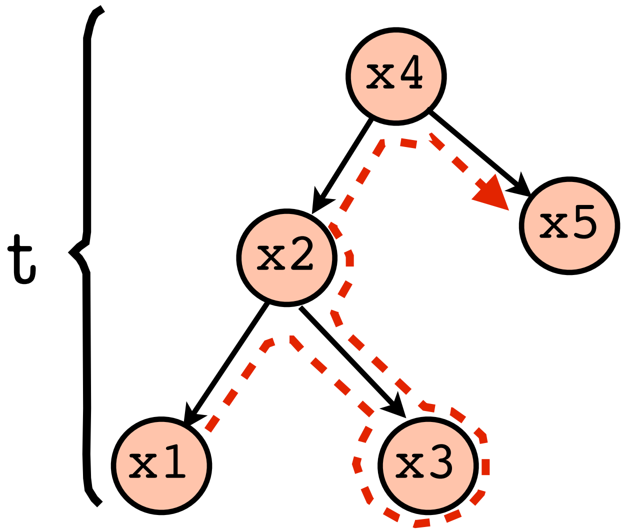
For Example ...



`in-order` of t is binary relation such that: `in-order`$(x_i,x_j) \Leftrightarrow i \leq j$

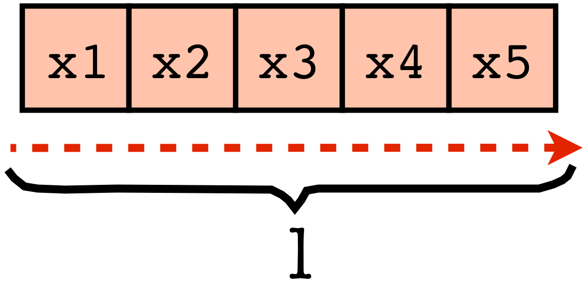
$$R_{io}(t) = \{(x_i,x_j) \mid i \leq j\}$$

For Example ...

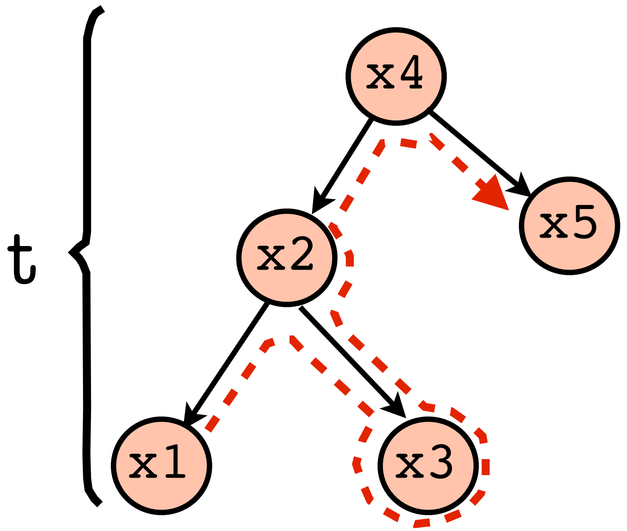


`in-order` of t is binary relation such that: `in-order`$(x_i,x_j) \Leftrightarrow i \leq j$

$$R_{io}(t) = \{(x_i,x_j) \mid i \leq j\}$$

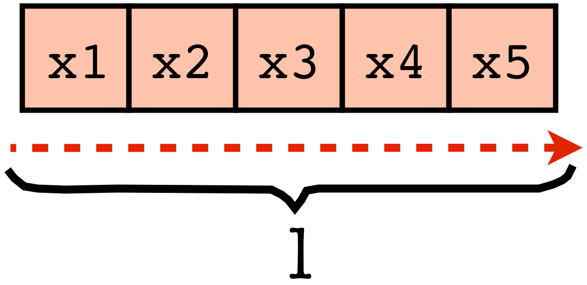


For Example ...



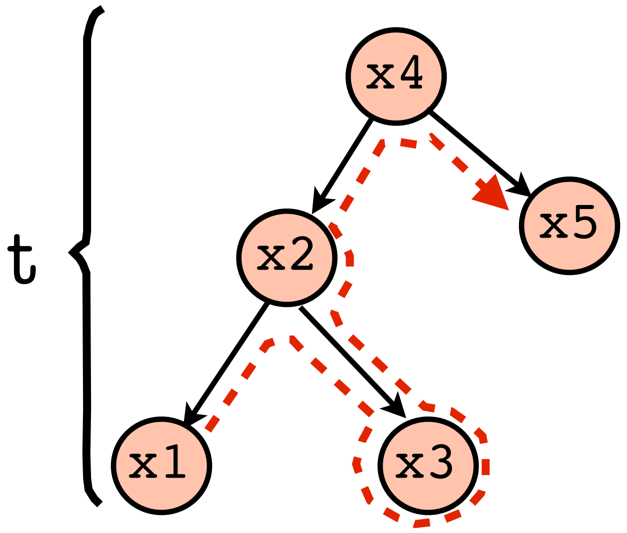
in-order of t is binary relation such that: in-order$(x_i,x_j) \Leftrightarrow i \leq j$

$R_{io}(t) = \{(x_i,x_j) \mid i \leq j\}$



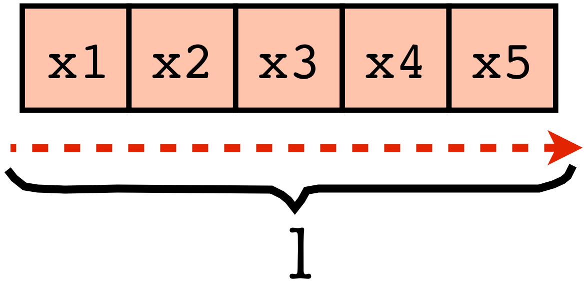
fwd-order of l is binary relation such that: fwd-order$(x_i,x_j) \Leftrightarrow i \leq j$

For Example ...



in-order of t is binary relation such that: in-order(x_i,x_j) ⇔ $i \leq j$

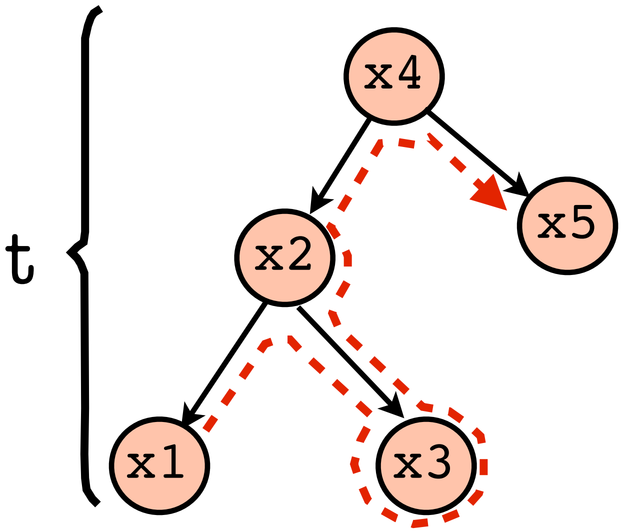
$R_{io}(t) = \{(x_i,x_j) \mid i \leq j\}$



fwd-order of l is binary relation such that: fwd-order(x_i,x_j) ⇔ $i \leq j$

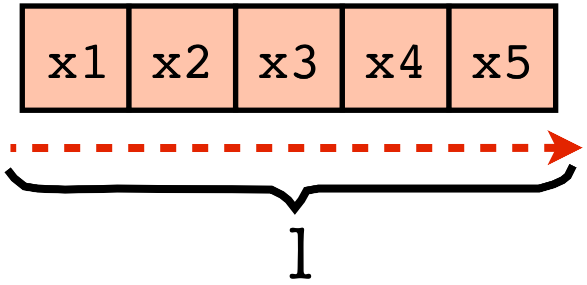
$R_{fo}(l)$

For Example ...



in-order of t is binary relation such that: $\text{in-order}(x_i,x_j) \Leftrightarrow i \leq j$

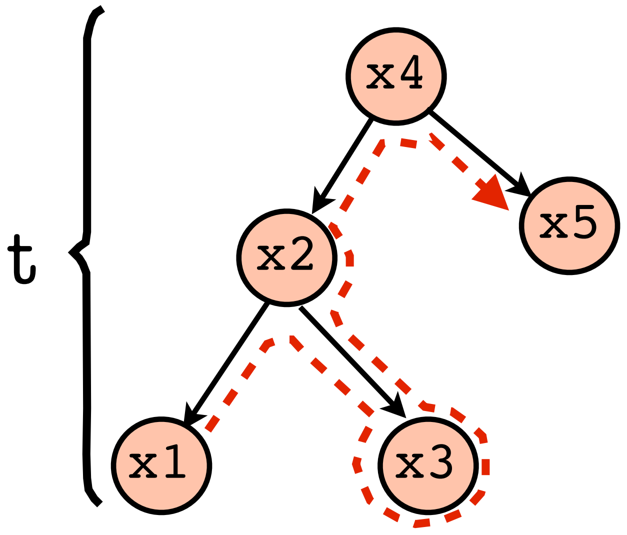
$R_{io}(t) = \{(x_i,x_j) \mid i \leq j\}$



fwd-order of l is binary relation such that: $\text{fwd-order}(x_i,x_j) \Leftrightarrow i \leq j$

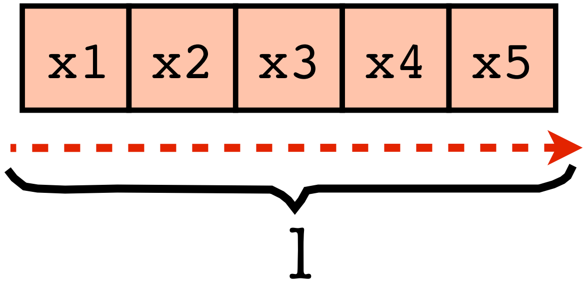
$R_{fo}(l) = \{(x_i,x_j) \mid i \leq j\}$

For Example ...



in-order of t is binary relation such that: $\mathtt{in\text{-}order}(x_i,x_j) \Leftrightarrow i \leq j$

$$R_{io}(t) = \{(x_i,x_j) \mid i \leq j\}$$



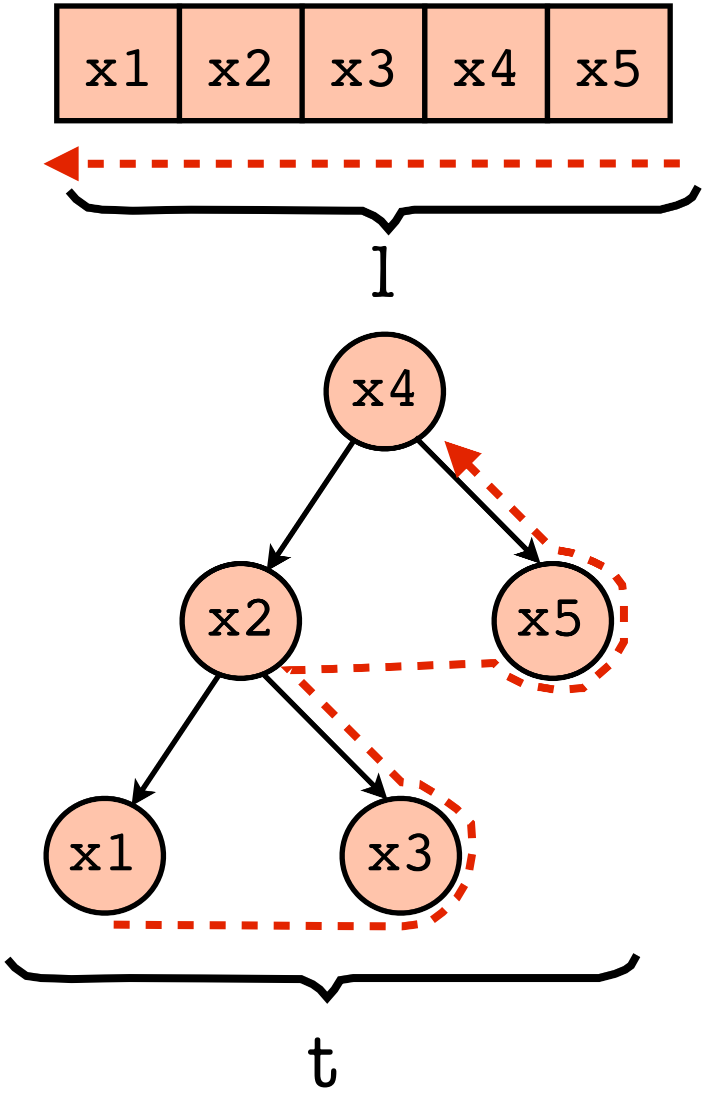
fwd-order of l is binary relation such that: $\mathtt{fwd\text{-}order}(x_i,x_j) \Leftrightarrow i \leq j$

$$R_{fo}(l) = \{(x_i,x_j) \mid i \leq j\}$$

⇒ If list l contains elements of tree t in pre-order, then

$$R_{fo}(l) = R_{io}(t)$$

More Relations



`post-order` on tree `t` and `backward-order` on list `l` are also binary relations, hence set of pairs.

More Relations

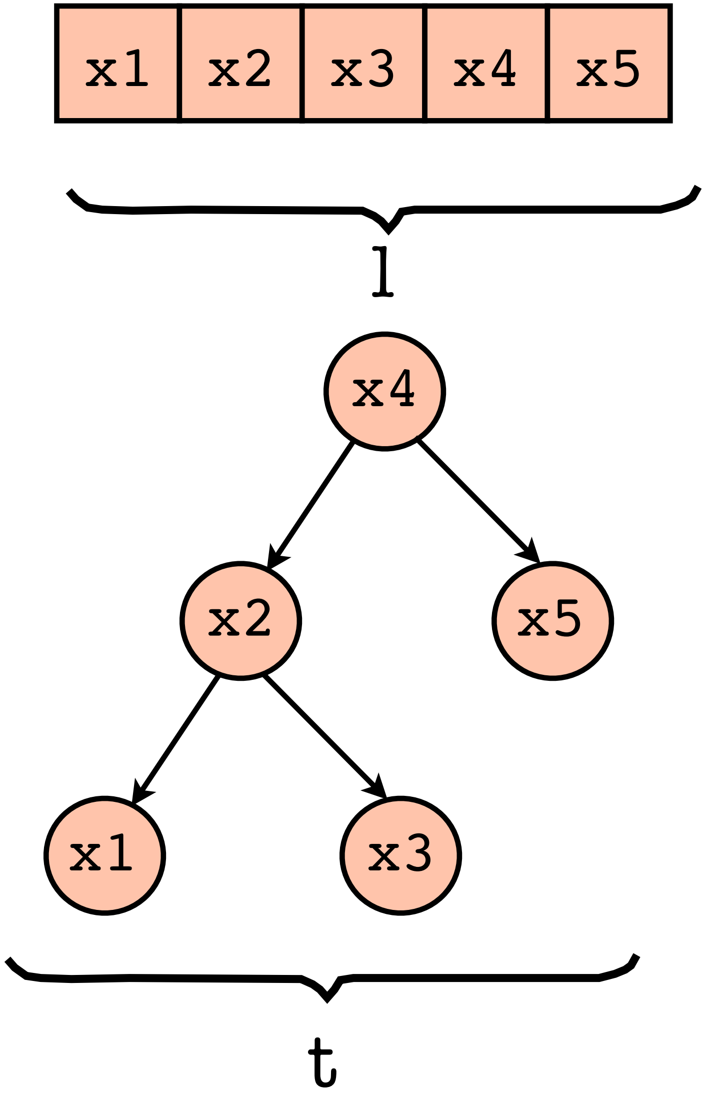
`post-order` on tree `t` and `backward-order` on list `l` are also binary relations, hence set of pairs.

Of supplementary value are unary membership relations:

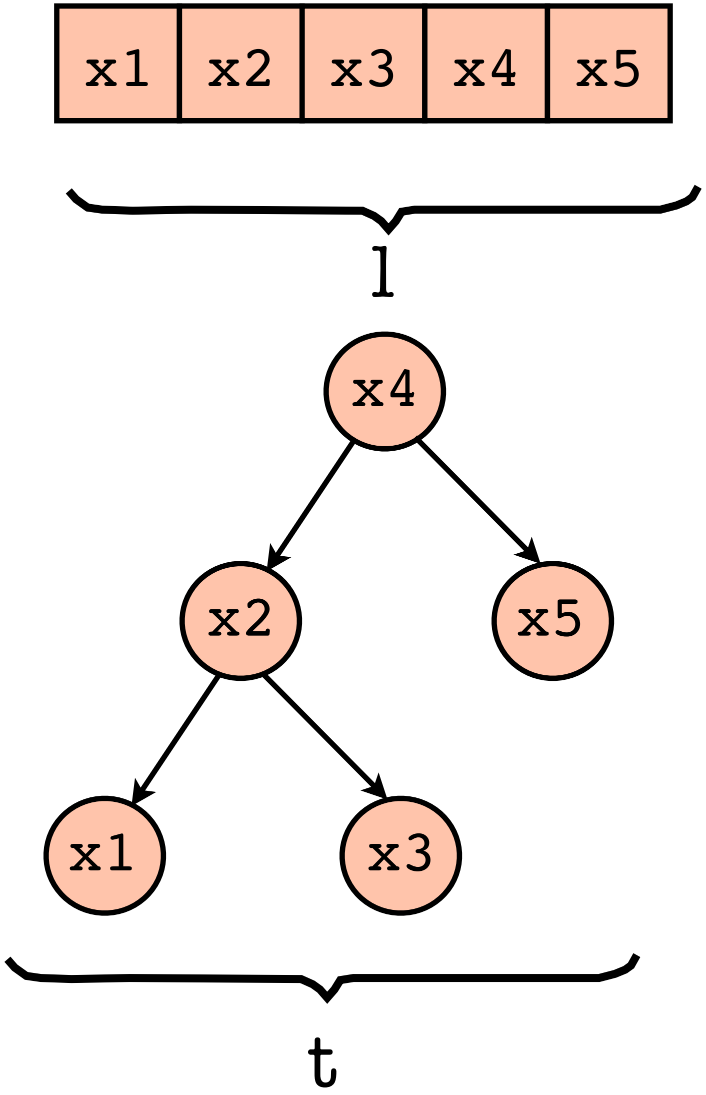
`tree-members`

$R_{tm}(t) = R_{lm}(l) = \{x1, x2, x3, x4, x5\}$

`list-members`



More Relations



`post-order` on tree `t` and `backward-order` on list `l` are also binary relations, hence set of pairs.

Of supplementary value are unary membership relations:

`tree-members`

$$R_{tm}(t) = R_{lm}(l) = \{x1, x2, x3, x4, x5\}$$

`list-members`

They let us write assertions over binary relations like R_{po}

More Relations

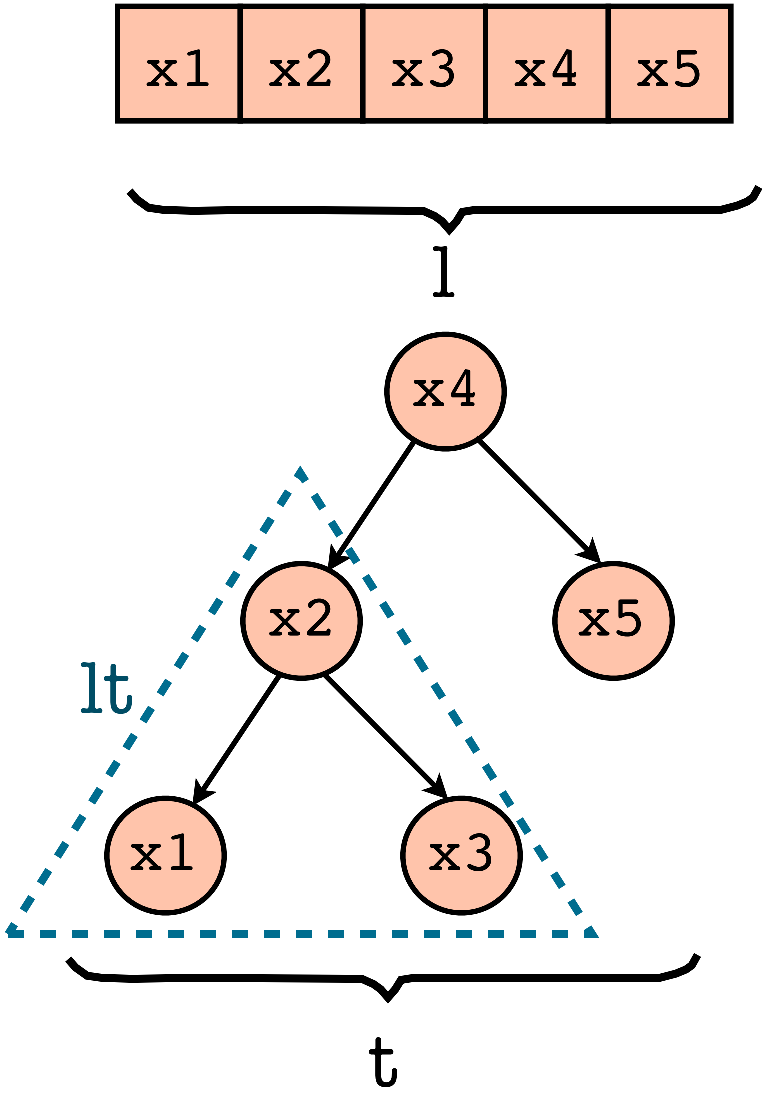
`post-order` on tree `t` and `backward-order` on list `l` are also binary relations, hence set of pairs.

Of supplementary value are unary membership relations:

tree-members

$R_{tm}(t) = R_{lm}(l) = \{x1, x2, x3, x4, x5\}$

list-members



They let us write assertions over binary relations like R_{po}

More Relations

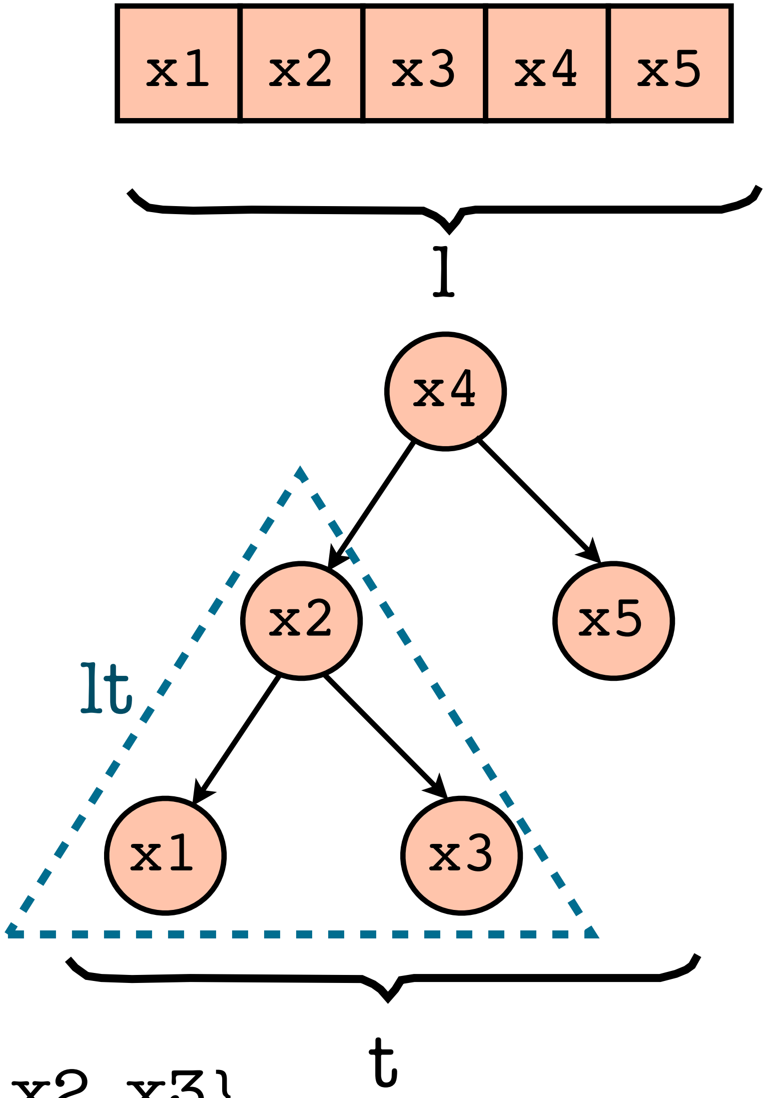
`post-order` on tree `t` and `backward-order` on list `l` are also binary relations, hence set of pairs.

Of supplementary value are unary membership relations:

`tree-members`

$R_{tm}(t) = R_{lm}(l) = \{x1, x2, x3, x4, x5\}$

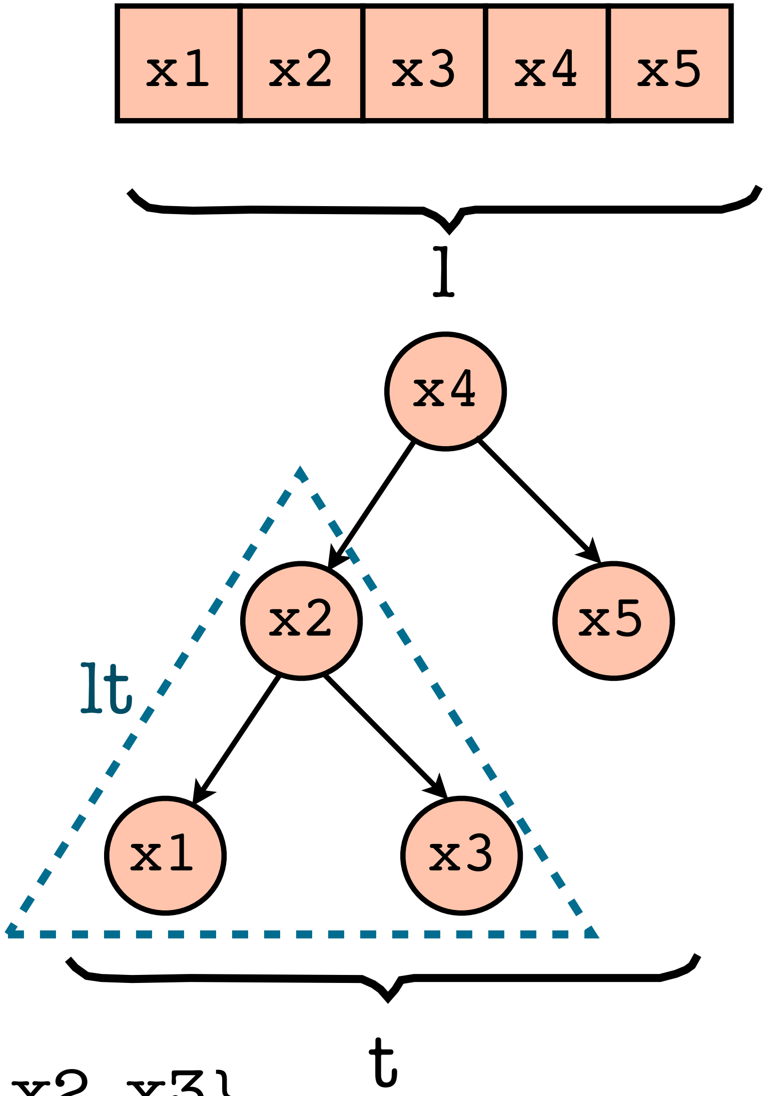
`list-members`



$R_{tm}(lt) = \{x1, x2, x3\}$

They let us write assertions over binary relations like R_{po}

More Relations



`post-order` on tree `t` and `backward-order` on list `l` are also binary relations, hence set of pairs.

Of supplementary value are unary membership relations:

`tree-members`

$$R_{tm}(t) = R_{lm}(l) = \{x1, x2, x3, x4, x5\}$$

`list-members`

$$R_{tm}(lt) = \{x1, x2, x3\}$$

They let us write assertions over binary relations like R_{po}

$$R_{tm}(lt) \times \{x4\} \subset R_{io}(t)$$

The Language of Relations ...

... with relational operators, such as union and cross-product, is capable of expressing fine-grained shapes.

$\bigcup \mathtt{R}_{\mathtt{fo}}\mathtt{(xs)}$

The Language of Relations ...

... with relational operators, such as union and cross-product, is capable of expressing fine-grained shapes.

Equality (=) and Subset inclusion (⊂) predicates over relations let us relate shapes of data structures.

$\bigcup \mathtt{R}_{\mathtt{fo}}(\mathtt{xs})$

The Language of Relations ...

... with relational operators, such as union and cross-product, is capable of expressing fine-grained shapes.

Equality (=) and Subset inclusion (⊂) predicates over relations let us relate shapes of data structures.

For Eg:

$$\bigcup \mathtt{R}_{\mathtt{fo}}(\mathtt{xs})$$

The Language of Relations ...

... with relational operators, such as union and cross-product, is capable of expressing fine-grained shapes.

Equality (=) and Subset inclusion (⊂) predicates over relations let us relate shapes of data structures.

For Eg:

$$\texttt{relation } \texttt{R}_{\texttt{fo}}\texttt{(x::xs)} \ = \ \texttt{(\{x\}} \times \texttt{R}_{\texttt{mem}}\texttt{(xs))} \cup \texttt{R}_{\texttt{fo}}\texttt{(xs)}$$

$$\texttt{relation } \texttt{R}_{\texttt{io}}\texttt{(Tree(L,n,R))} \ =$$

$$\texttt{(R}_{\texttt{tm}}\texttt{(L) X \{n\})} \cup \texttt{(\{n\} X R}_{\texttt{tm}}\texttt{(R)) U R}_{\texttt{io}}\texttt{(L) U R}_{\texttt{io}}\texttt{(R)}$$

The Language of Relations ...

... with relational operators, such as union and cross-product, is capable of expressing fine-grained shapes.

Equality (=) and Subset inclusion (⊂) predicates over relations let us relate shapes of data structures.

For Eg:

$$\texttt{relation } R_{\texttt{fo}}\texttt{(x::xs)} \;\; = \texttt{(\{x\}} \times R_{\mathrm{mem}}\texttt{(xs))} \cup R_{\texttt{fo}}\texttt{(xs)}$$

$$\texttt{relation } R_{\texttt{io}}\texttt{(Tree(L,n,R))} \; =$$

$$\texttt{(}R_{\texttt{tm}}\texttt{(L) X \{n\})} \cup \texttt{(\{n\} X } R_{\texttt{tm}}\texttt{(R)) U } R_{\texttt{io}}\texttt{(L) U } R_{\texttt{io}}\texttt{(R)}$$

$$\texttt{inOrder : \{t:}\alpha\texttt{ tree\}} \longrightarrow \texttt{\{l:}\alpha\texttt{ list} \mid R_{\mathrm{fo}}(\mathrm{l}) = R_{\mathrm{io}}(\mathrm{t})\}$$

$$\texttt{tail : \{l:}\alpha\texttt{ list\}} \longrightarrow \texttt{\{v:}\alpha\texttt{ list} \mid R_{\mathrm{fo}}(\mathrm{v}) \subset R_{\mathrm{fo}}(\mathrm{l})\}$$

However ...

... to facilitate compositional type checking and verification, we should be able to ascribe relational types to polymorphic and higher-order functions.

However ...

... to facilitate compositional type checking and verification, we should be able to ascribe relational types to polymorphic and higher-order functions.

For eg:

$$\texttt{id} : \alpha \longrightarrow \alpha$$

$$\texttt{pairMap} : \alpha * \alpha \longrightarrow (\alpha \longrightarrow \beta) \longrightarrow \beta * \beta$$

However ...

... to facilitate compositional type checking and verification, we should be able to ascribe relational types to polymorphic and higher-order functions.

For eg:

$$\texttt{id} : \alpha \longrightarrow \alpha$$

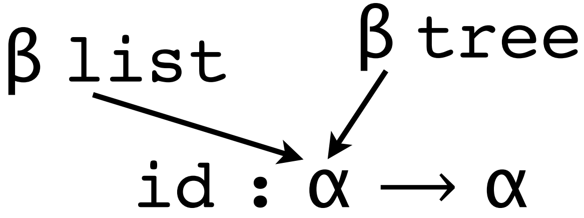
$$\texttt{pairMap} : \alpha * \alpha \longrightarrow (\alpha \longrightarrow \beta) \longrightarrow \beta * \beta$$

Relational types for polymorphic and higher-order functions must be general enough to relate different shapes at different call sites.

$\texttt{id} : \alpha \longrightarrow \alpha$

$\texttt{id} : \alpha \longrightarrow \alpha$

$$\texttt{id} : \alpha \longrightarrow \alpha$$

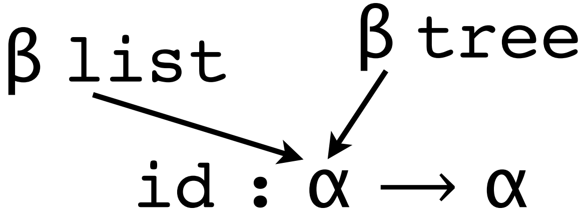


β `list`

β `tree`

`id : α ⟶ α`

`id` can take arguments of unknown shape



β list

β tree

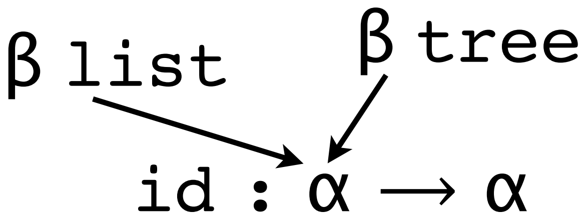
id : $\alpha \longrightarrow \alpha$

`id` can take arguments of unknown shape

Shape of the argument is also the shape of its result

```
id : {x:α} ⟶ {y:α | Shape(y) = Shape(x)}
```

Relational Parameters



β list

β tree

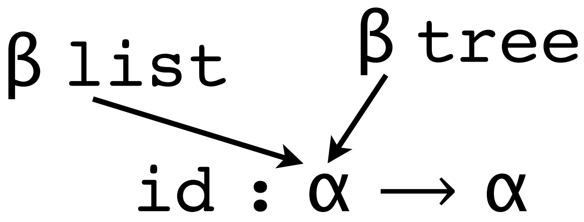
`id : α ⟶ α`

`id` can take arguments
of unknown shape

Shape of the argument is also the shape of its result

```
id : {x:α} ⟶ {y:α | Shape(y) = Shape(x)}
```

Relational Parameters


β list

β tree

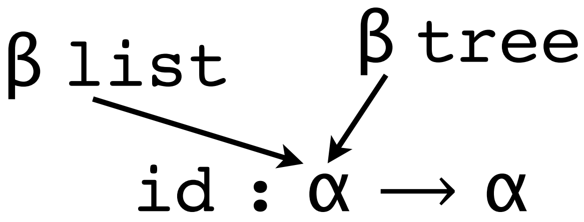
id : α ⟶ α

`id` can take arguments of $\underbrace{\text{unknown shape}}_{\text{Denote with an abstract relation}}$

Shape of the argument is also the shape of its result

$$\texttt{id} : \{\texttt{x}:\alpha\} \longrightarrow \{\texttt{y}:\alpha \mid \underbrace{\texttt{Shape}}_{\rho}(\texttt{y}) = \texttt{Shape}(\texttt{x})\}$$

Relational Parameters



β list

β tree

id : α ⟶ α

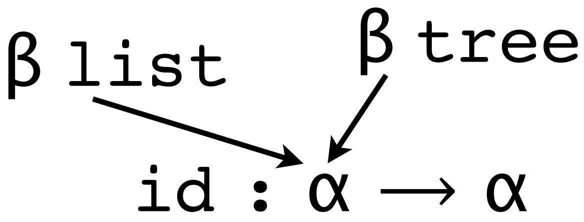
`id` can take arguments of $\underbrace{\text{unknown shape}}_{\text{Denote with an abstract relation}}$

Shape of the argument is also the shape of its result

$$\texttt{id} : \{x{:}\alpha\} \longrightarrow \{y{:}\alpha \mid \underbrace{\texttt{Shape}}_{\rho}(y) = \texttt{Shape}(x)\}$$

$$(\rho)\,\texttt{Id} : \{x{:}\alpha\} \longrightarrow \{y{:}\alpha \mid \rho(y) = \rho(x)\}$$

Relational Parameters



id can take arguments of $\underbrace{\text{unknown shape}}_{\text{Denote with an abstract relation}}$

Shape of the argument is also the shape of its result

$$\texttt{id : \{x:}\alpha\texttt{\}} \longrightarrow \texttt{\{y:}\alpha \mid \underbrace{\texttt{Shape}}_{\rho}\texttt{(y) = Shape(x)\}}$$

$$(\rho)\,\texttt{Id : \{x:}\alpha\texttt{\}} \longrightarrow \texttt{\{y:}\alpha \mid \rho\texttt{(y)} = \rho\texttt{(x)\}}$$

Relationally parametric type of id

A Parametric Type of pairMap ...

... by focusing on possible shape invariance between α and β

$$(\rho_\alpha,\rho_\beta)\ \texttt{pairMap} : \{x_1{:}\alpha\}*\{x_2{:}\alpha\}$$

$$\longrightarrow (\{x{:}\alpha\} \longrightarrow \{y{:}\beta \mid \rho_\beta(y) = \rho_\alpha(x)\})$$

$$\longrightarrow \{y_1{:}\beta \mid \rho_\beta(y_1) = \rho_\alpha(x_1)\}$$
$$* \{y_2{:}\beta \mid \rho_\beta(y_2) = \rho_\alpha(x_2)\}$$

A Parametric Type of pairMap ...

... by focusing on possible shape invariance between α and β

$$\underbrace{(\rho_\alpha,\rho_\beta)}_{\text{denote shapes of }\alpha\text{ and }\beta\text{, respectively}}\ \texttt{pairMap} : \{x_1:\alpha\}*\{x_2:\alpha\}$$

$$\longrightarrow (\{x:\alpha\} \longrightarrow \{y:\beta \mid \rho_\beta(y) = \rho_\alpha(x)\})$$

$$\longrightarrow \{y_1:\beta \mid \rho_\beta(y_1) = \rho_\alpha(x_1)\}$$

$$* \{y_2:\beta \mid \rho_\beta(y_2) = \rho_\alpha(x_2)\}$$

A Parametric Type of pairMap ...

... by focusing on possible shape invariance between α and β

$$\underbrace{(\rho_\alpha,\rho_\beta)}_{\text{denote shapes of } \alpha \text{ and } \beta\text{, respectively}}\ \ \texttt{pairMap} : \{x_1:\alpha\}*\{x_2:\alpha\}$$

$$\longrightarrow (\{x:\alpha\} \longrightarrow \{y:\beta \mid \rho_\beta(y) = \rho_\alpha(x)\})$$

$$\longrightarrow \{y_1:\beta \mid \rho_\beta(y_1) = \rho_\alpha(x_1)\}$$

$$* \ \{y_2:\beta \mid \rho_\beta(y_2) = \rho_\alpha(x_2)\}$$

A Parametric Type of pairMap ...

... by focusing on possible shape invariance between α and β

$$\underbrace{(\rho_\alpha,\rho_\beta)}_{\substack{\text{denote shapes}\\ \text{of } \alpha \text{ and } \beta,\\ \text{respectively}}}\ \texttt{pairMap} : \{x_1:\alpha\} * \{x_2:\alpha\}$$

$$\longrightarrow (\{x:\alpha\} \longrightarrow \{y:\beta \mid \rho_\beta(y) = \rho_\alpha(x)\})$$

$$\longrightarrow \{y_1:\beta \mid \rho_\beta(y_1) = \rho_\alpha(x_1)\}$$

$$* \ \{y_2:\beta \mid \rho_\beta(y_2) = \rho_\alpha(x_2)\}$$

gets propagated to result type

A Parametric Type of pairMap ...

$$
\begin{aligned}
(\rho_\alpha,\rho_\beta)\ \texttt{pairMap} :\ & \{x_1{:}\alpha\} * \{x_2{:}\alpha\} \\
& \rightarrow (\{x{:}\alpha\} \rightarrow \{y{:}\beta \mid \rho_\beta(y) = \rho_\alpha(x)\}) \\
& \rightarrow \{y_1{:}\beta \mid \rho_\beta(y_1) = \rho_\alpha(x_1)\} \\
& \quad * \{y_2{:}\beta \mid \rho_\beta(y_2) = \rho_\alpha(x_2)\}
\end{aligned}
$$

For eg:

$$\underbrace{(l_1, l_2)}_{\alpha\ \texttt{lists}} = \texttt{pairMap}\ (R_{io}, R_{fo})\ \underbrace{(t_1, t_2)}_{\alpha\ \texttt{trees}}\ \texttt{inOrder}$$

A Parametric Type of pairMap ...

$$(\rho_\alpha,\rho_\beta)\ \texttt{pairMap} : \{\texttt{x}_1:\alpha\}*\{\texttt{x}_2:\alpha\}$$

$$\longrightarrow (\{\texttt{x}:\alpha\} \longrightarrow \{\texttt{y}:\beta \mid \rho_\beta(\texttt{y}) = \rho_\alpha(\texttt{x})\})$$

$$\longrightarrow \{\texttt{y}_1:\beta \mid \rho_\beta(\texttt{y}_1) = \rho_\alpha(\texttt{x}_1)\}$$

$$* \{\texttt{y}_2:\beta \mid \rho_\beta(\texttt{y}_2) = \rho_\alpha(\texttt{x}_2)\}$$

For eg:

$$\underbrace{(\texttt{l}_1, \texttt{l}_2)}_{\alpha\ \texttt{lists}} = \texttt{pairMap}\ (\mathcal{R}_{io}, \mathcal{R}_{fo})\ \underbrace{(\texttt{t}_1, \texttt{t}_2)}_{\alpha\ \texttt{trees}}\ \texttt{inOrder}$$

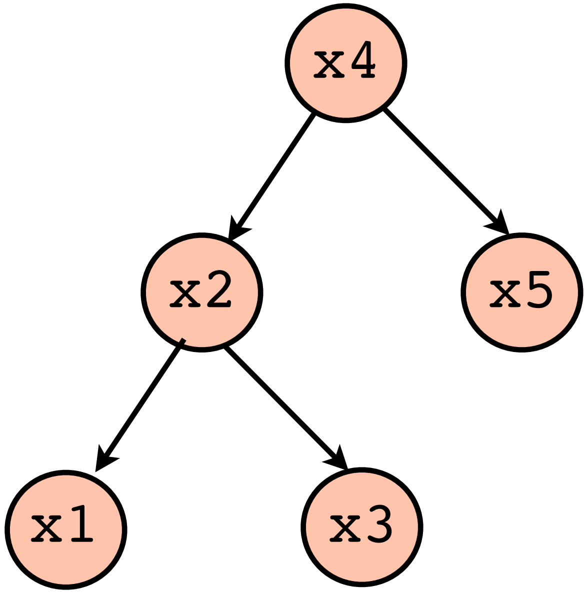
explicit instantiation
of
relational parameters

treefoldl

```
treefoldl f i (Node n) = f i n
        | f i (Tree left node right) =
              treefoldl f (f (treefoldl f i left) node) right
```

```
val inOrder = fn t => treefoldl t []
      (fn acc => fn x => acc ++ [x])
```

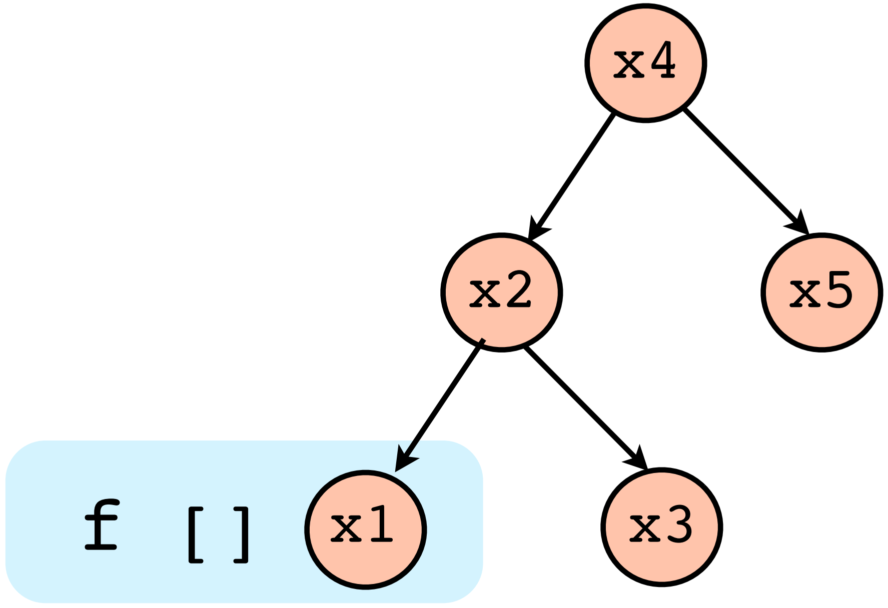
inOrder t =



treefoldl

```
treefoldl f i (Node n) = f i n
        | f i (Tree left node right) =
              treefoldl f (f (treefoldl f i left) node) right

val inOrder = fn t => treefoldl t []
      (fn acc => fn x => acc ++ [x])
```

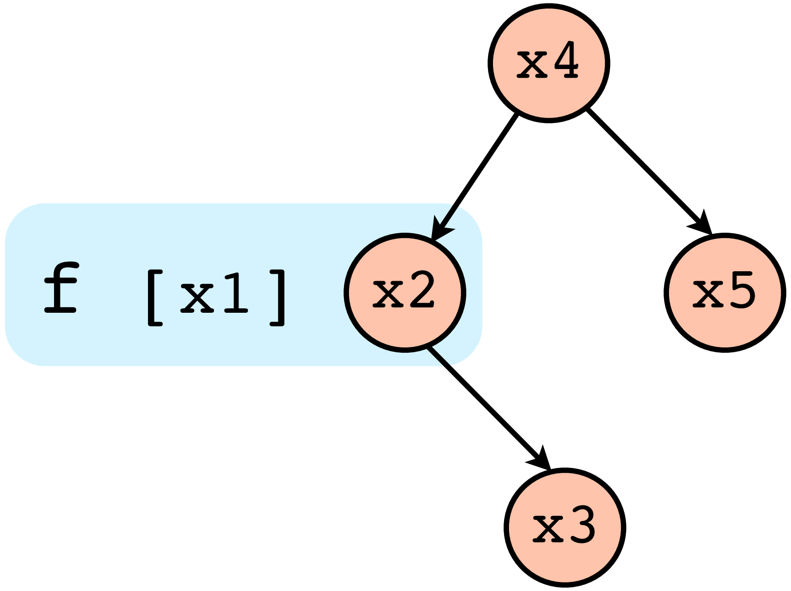
inOrder t =



treefoldl

```
treefoldl f i (Node n) = f i n
        | f i (Tree left node right) =
              treefoldl f (f (treefoldl f i left) node) right

val inOrder = fn t => treefoldl t []
      (fn acc => fn x => acc ++ [x])
```

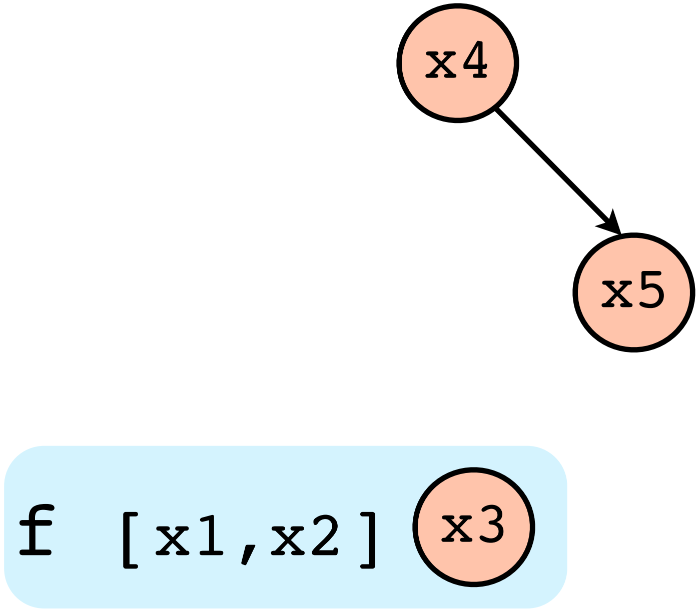
inOrder t =



treefoldl

```
treefoldl f i (Node n) = f i n
        | f i (Tree left node right) =
              treefoldl f (f (treefoldl f i left) node) right

val inOrder = fn t => treefoldl t []
      (fn acc => fn x => acc ++ [x])
```

inOrder t =

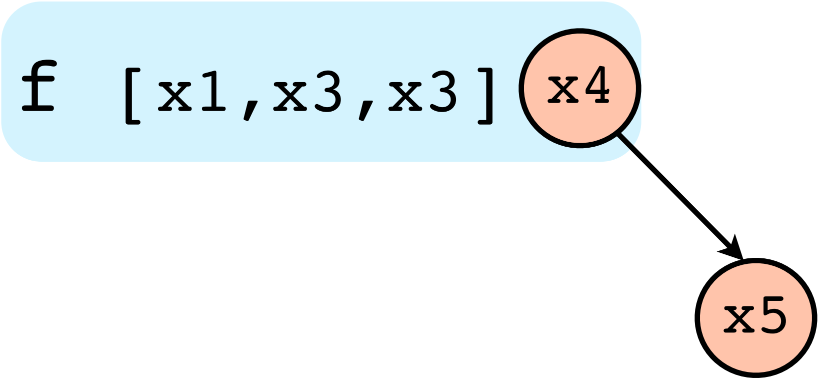


treefoldl

```
treefoldl f i (Node n) = f i n
        | f i (Tree left node right) =
             treefoldl f (f (treefoldl f i left) node) right
```

```
val inOrder = fn t => treefoldl t []
      (fn acc => fn x => acc ++ [x])
```

inOrder t =



treefoldl

```
treefoldl f i (Node n) = f i n
        | f i (Tree left node right) =
              treefoldl f (f (treefoldl f i left) node) right

val inOrder = fn t => treefoldl t []
      (fn acc => fn x => acc ++ [x])
```

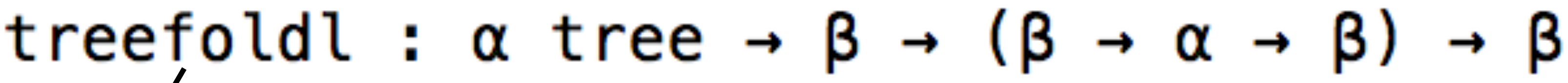
```
inOrder t =        f [x1,x3,x3,x4] x5
```

treefoldl

```
treefoldl f i (Node n) = f i n
        | f i (Tree left node right) =
              treefoldl f (f (treefoldl f i left) node) right

val inOrder = fn t => treefoldl t []
      (fn acc => fn x => acc ++ [x])
```

inOrder t = [x1,x3,x3,x4,x5]

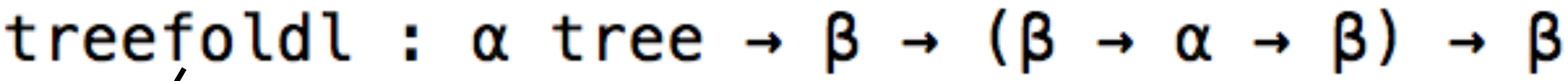
treefoldl



```
treefoldl : α tree → β → (β → α → β) → β
```

folds a tree from left to
right in in-order

treefoldl



```
treefoldl : α tree → β → (β → α → β) → β
```

folds a tree from left to right in in-order

A parametric type can be constructed to relate `in-order` ($\mathcal{R}_{io}$) on α `tree` to some notion of order captured by an abstract relation (ρ_{o}) on β

treefoldl

```
treefoldl : α tree → β → (β → α → β) → β
```

folds a tree from left to right in in-order

A parametric type can be constructed to relate `in-order` ($\mathcal{R}_{io}$) on α `tree` to some notion of order captured by an abstract relation (ρ_o) on β

$$(\rho_o)\ \texttt{treefoldl:}\ \{\texttt{t:}\alpha\ \texttt{tree}\} \longrightarrow \ldots$$

$$\longrightarrow \{\texttt{v:}\ \beta \mid \rho_o(\texttt{v}) = \mathcal{R}_{io}(\texttt{t})\}$$

A Parametric Type of `treefoldl`

$$(\rho_m,\rho_o)\ \mathtt{treefoldl}\colon \{t{:}\alpha\ \mathtt{tree}\} \longrightarrow \{b{:}\beta \mid \rho_m(b)=\emptyset$$

$$\wedge\ \rho_o(b)=\emptyset\}$$

$$\longrightarrow (\{xs{:}\beta\} \longrightarrow \{x{:}\alpha\} \longrightarrow$$

$$\{v{:}\beta \mid \rho_m(v)=\rho_m(xs)\ \cup\ \{x\}$$

$$\wedge\ \rho_o(v)=\rho_m(xs)\times\{x\}\ \cup\ \rho_o(xs)\})$$

$$\longrightarrow \{y{:}\ \beta \mid \rho_o(y)=R_{io}(t)\ \wedge\ \rho_m(y)=R_{tm}(t)\ \}$$

A Parametric Type of `treefoldl`

$$
\begin{aligned}
(\rho_m,\rho_o)\ \texttt{treefoldl}: \{t:\alpha\ \texttt{tree}\} \longrightarrow \{b:\beta \mid\ & \rho_m(b)=\varnothing \\
& \wedge\ \rho_o(b)=\varnothing\} \\
\longrightarrow \big(\{xs:\beta\} \longrightarrow \{x:\alpha\} \longrightarrow\ & \\
\{v:\beta \mid \rho_m(v) = \rho_m(xs)\ & \cup\ \{x\} \\
\wedge\ \rho_o(v) = \rho_m(xs)\times\{x\}\ & \cup\ \rho_o(xs)\}\big) \\
\longrightarrow \{y:\ \beta \mid \rho_o(y) = R_{io}(t)\ \wedge\ & \rho_m(y) = R_{tm}(t)\ \}
\end{aligned}
$$

Order invariant: relates `in-order` on the tree to a notion of order on β

A Parametric Type of `treefoldl`

$$
\begin{aligned}
(\rho_m,\rho_o)\ \texttt{treefoldl}: \{\texttt{t}:\alpha\ \texttt{tree}\} \longrightarrow \{\texttt{b}:\beta \mid\ & \rho_m(\texttt{b})=\emptyset \\
& \wedge\ \rho_o(\texttt{b})=\emptyset\} \\
\longrightarrow \big(\{\texttt{xs}:\beta\} \longrightarrow \{\texttt{x}:\alpha\} \longrightarrow\ & \\
\{\texttt{v}:\beta \mid \rho_m(\texttt{v}) = \rho_m(\texttt{xs})\ &\cup\ \{\texttt{x}\} \\
\wedge\ \rho_o(\texttt{v}) = \rho_m(\texttt{xs})\times\{\texttt{x}\}\ &\cup\ \rho_o(\texttt{xs})\}\big) \\
\longrightarrow \{\texttt{y}:\ \beta \mid \rho_o(\texttt{y}) = \mathcal{R}_{io}(\texttt{t})\ \wedge\ & \rho_m(\texttt{y}) = \mathcal{R}_{tm}(\texttt{t})\ \}
\end{aligned}
$$

Order invariant: relates `in-order` on the tree to a notion of order on β

Membership invariant: relates `membership` of the tree to a notion of membership of β

A Parametric Type of `treefoldl`

$$
\begin{aligned}
&(\rho_m,\rho_o)\ \texttt{treefoldl}: \{\mathtt{t}:\alpha\ \mathtt{tree}\} \longrightarrow \{\mathtt{b}:\beta \mid \rho_m(\mathtt{b})=\varnothing \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \rho_o(\mathtt{b})=\varnothing\} \\
&\quad \longrightarrow \big(\{\mathtt{xs}:\beta\} \longrightarrow \{\mathtt{x}:\alpha\} \longrightarrow \\
&\qquad\qquad \{\mathtt{v}:\beta \mid \rho_m(\mathtt{v}) = \rho_m(\mathtt{xs})\ \cup\ \{\mathtt{x}\} \\
&\qquad\qquad\qquad \wedge\ \rho_o(\mathtt{v}) = \rho_m(\mathtt{xs})\times\{\mathtt{x}\}\ \cup\ \rho_o(\mathtt{xs})\}\big) \\
&\quad \longrightarrow \{\mathtt{y}:\ \beta \mid \rho_o(\mathtt{y}) = R_{io}(\mathtt{t})\ \wedge\ \rho_m(\mathtt{y}) = R_{tm}(\mathtt{t})\ \}
\end{aligned}
$$

Order invariant: relates `in-order` on the tree to a notion of order on β

Membership invariant: relates `membership` of the tree to a notion of membership of β

inOrder using treefoldl

```
val inOrder = fn t => treefoldl ($R_{lm}$,$R_{fo}$) t []
                  (fn acc => fn x => acc ++ [x])
```

inOrder using treefoldl

$$\texttt{val inOrder = fn t => treefoldl}\ \overbrace{(R_{\texttt{lm}},R_{\texttt{fo}})}^{\text{Explicit relational parameter instantiation}}\ \texttt{t [ ]}$$

$$\texttt{(fn acc => fn x => acc ++ [x])}$$

inOrder using treefoldl

Explicit relational parameter instantiation

```
val inOrder = fn t => treefoldl (R_lm,R_fo) t []
            (fn acc => fn x => acc ++ [x])
```

has type

$$\{t{:}\alpha\ \texttt{tree}\} \longrightarrow \ldots \longrightarrow \{v{:}\ \alpha\ \texttt{list} \mid R_{fo}(v) = R_{io}(t) \wedge R_{lm}(v) = R_{tm}(t)\ \}$$

Parametric Relations

`id` and `pairMap` are functions parameterized over relations

Parametric Relations

`id` and `pairMap` are functions parameterized over relations

Relations can also be parameterized over relations

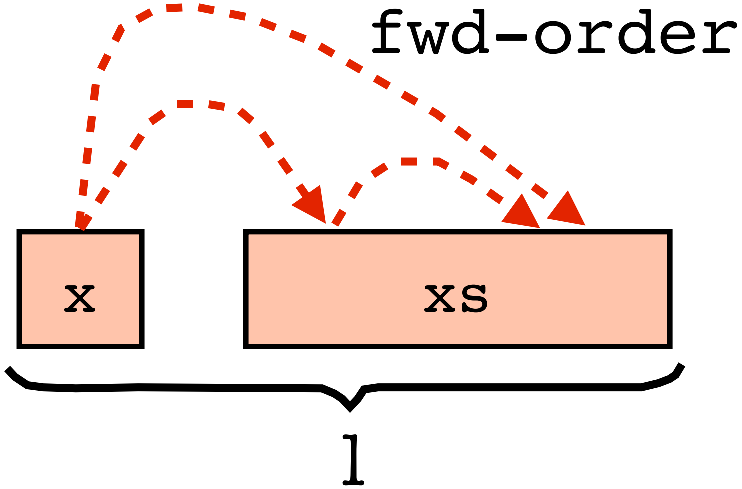
Parametric Relations

`id` and `pairMap` are functions parameterized over relations

Relations can also be parameterized over relations

For Eg:

$$R_{fo}(l) = \{x\} \times R_{lm}(xs) \cup R_{fo}(xs)$$



Parametric Relations

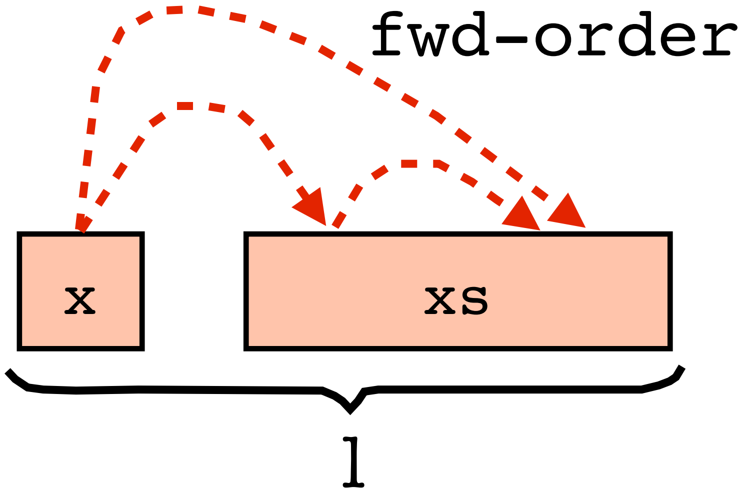
`id` and `pairMap` are functions parameterized over relations

Relations can also be parameterized over relations

For Eg:

Relates elements of l

$$R_{fo}(l) = \{x\} \times R_{lm}(xs) \cup R_{fo}(xs)$$



Parametric Relations

`id` and `pairMap` are functions parameterized over relations

Relations can also be parameterized over relations

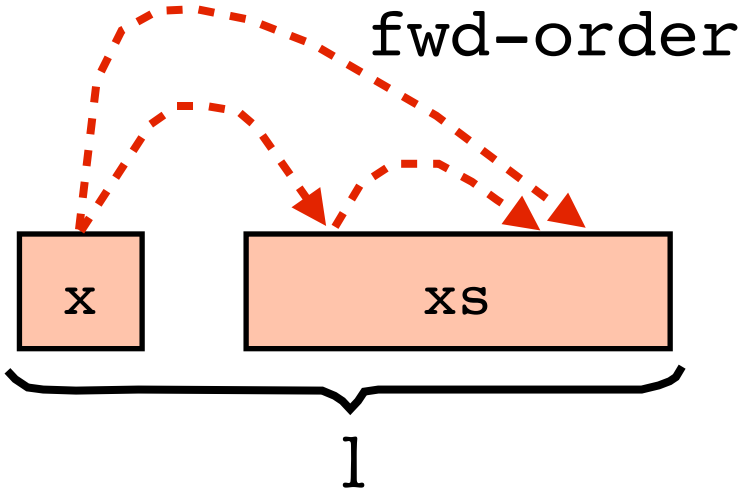
For Eg:

Relates elements of `l`

$R_{fo}(l) = \{x\} \times R_{lm}(xs) \cup R_{fo}(xs)$

Generalize

$R_{fo}[\rho](l) = \rho(x) \times R_{lm}[\rho](xs) \cup R_{fo}[\rho](xs)$



Parametric Relations

`id` and `pairMap` are functions parameterized over relations

Relations can also be parameterized over relations

For Eg:

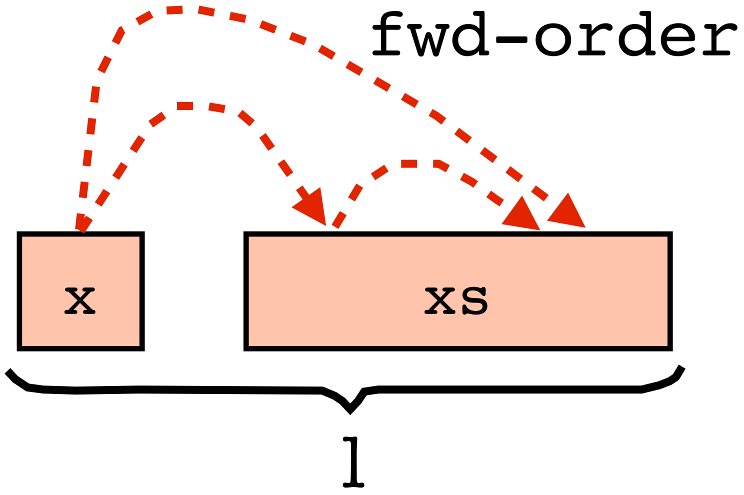
Relates elements of `l`

$$R_{fo}(l) = \{x\} \times R_{lm}(xs) \cup R_{fo}(xs)$$

Generalize

$$R_{fo}[\rho](l) = \rho(x) \times R_{lm}[\rho](xs) \cup R_{fo}[\rho](xs)$$

Relates different things for different instantiations of ρ



Parametric Relations

`id` and `pairMap` are functions parameterized over relations

Relations can also be parameterized over relations

For Eg:

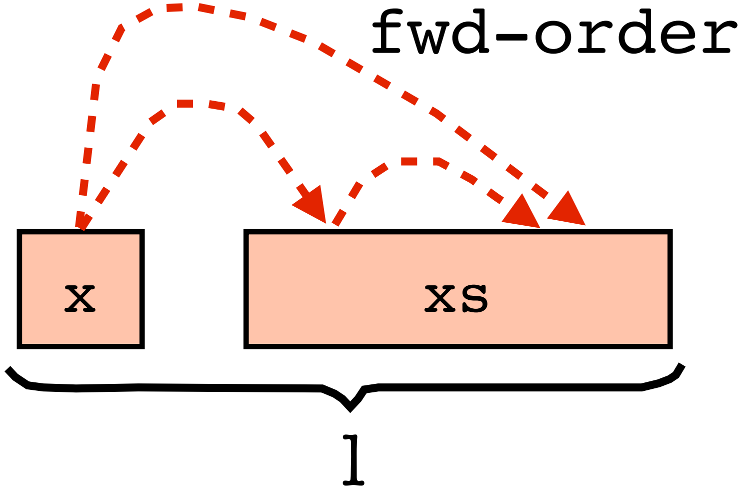
Relates elements of l

$$\mathrm{R}_{fo}(l) = \{x\} \times \mathrm{R}_{lm}(xs) \cup \mathrm{R}_{fo}(xs)$$

Generalize

$$\mathrm{R}_{fo}[\rho](l) = \rho(x) \times \mathrm{R}_{lm}[\rho](xs) \cup \mathrm{R}_{fo}[\rho](xs)$$

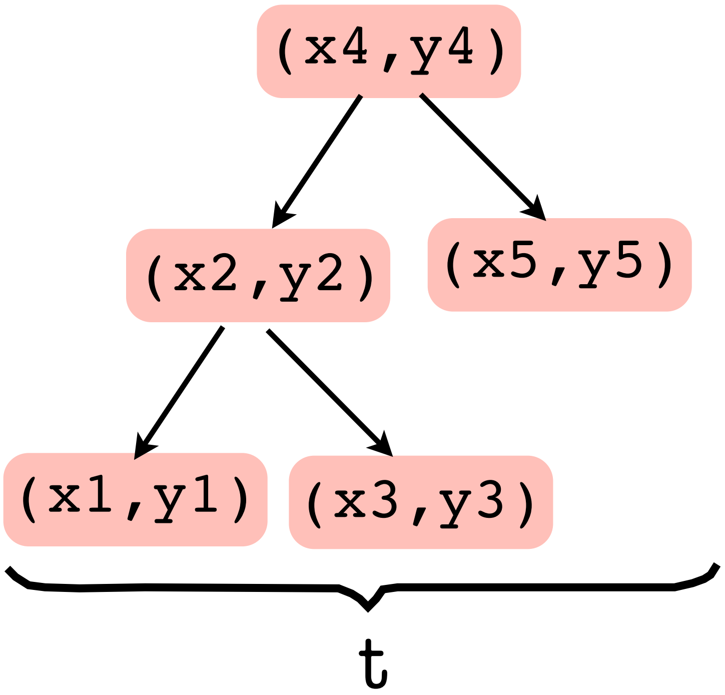
Relates different things for different instantiations of ρ



Note: If $\mathrm{R}_{id}(x)=\{x\}$ then $\mathrm{R}_{fo}[\mathrm{R}_{id}](l)$ relates elements like non-parametric $\mathrm{R}_{fo}(l)$

For Example …

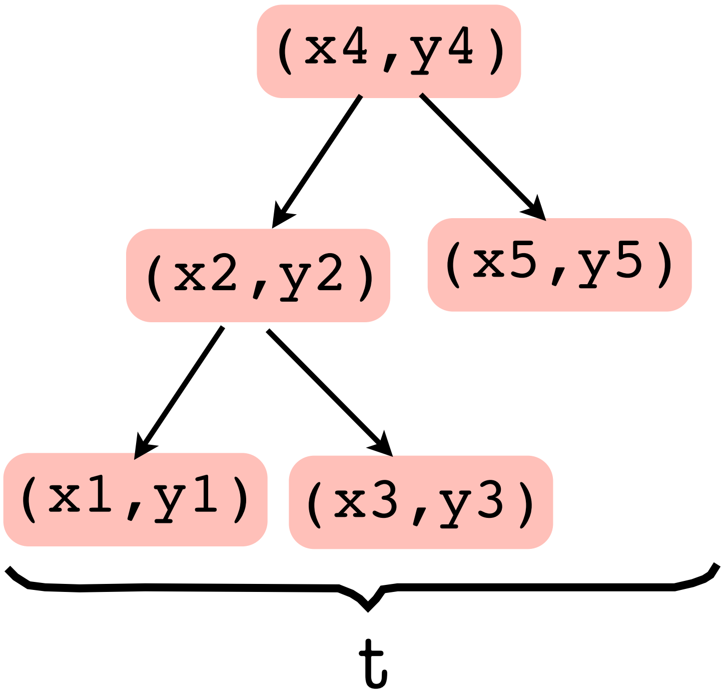
For Example ...



We know:

$$R_{io}(t) = \{((x_i, y_i), (x_j, y_j)) \mid i \leq j\}$$

For Example ...



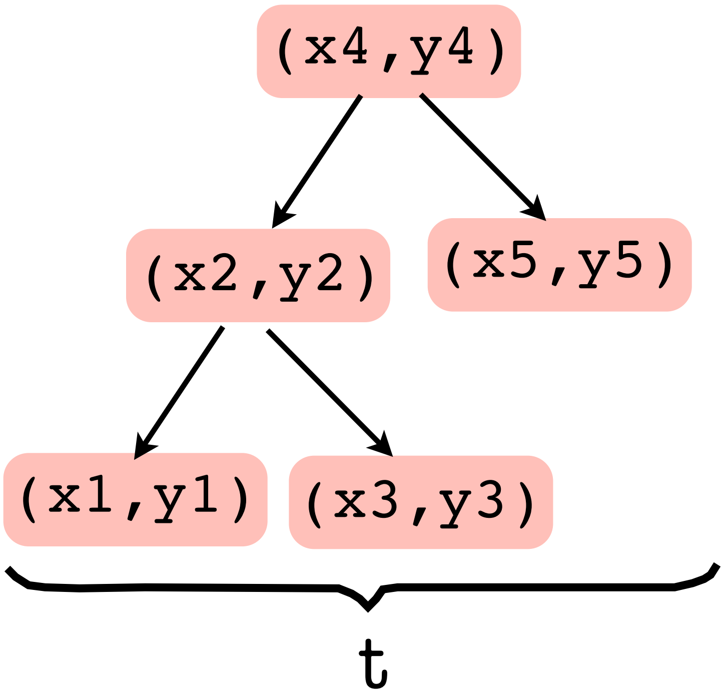
We know:

$$R_{io}(t) = \{((x_i, y_i), (x_j, y_j)) \mid i \leq j\}$$

By Definition:

$$R_{io}[\rho](t) = \{(\rho(x_i, y_i), \rho\,(x_j, y_j)) \mid i \leq j\}$$

For Example ...



We know:

$$R_{io}(t) = \{((x_i, y_i), (x_j, y_j)) \mid i \leq j\}$$

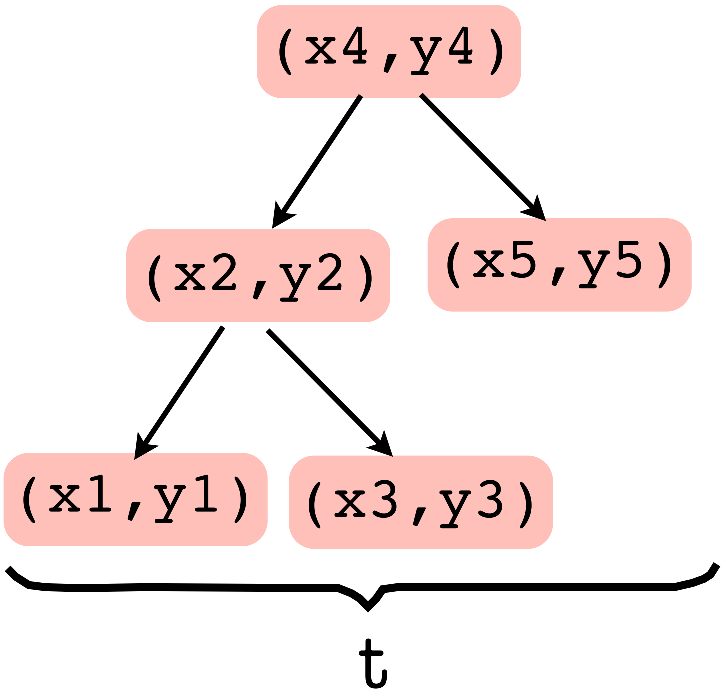
By Definition:

$$R_{io}[\rho](t) = \{(\rho(x_i, y_i), \rho\,(x_j, y_j)) \mid i \leq j\}$$

Let R_{fst} be a relation on pairs, such that

$$R_{fst}(x,y) = \{x\}$$

For Example ...



We know:

$$R_{io}(t) = \{((x_i, y_i), (x_j, y_j)) \mid i \leq j\}$$

By Definition:

$$R_{io}[\rho](t) = \{(\rho(x_i, y_i), \rho\,(x_j, y_j)) \mid i \leq j\}$$

Let R_{fst} be a relation on pairs, such that

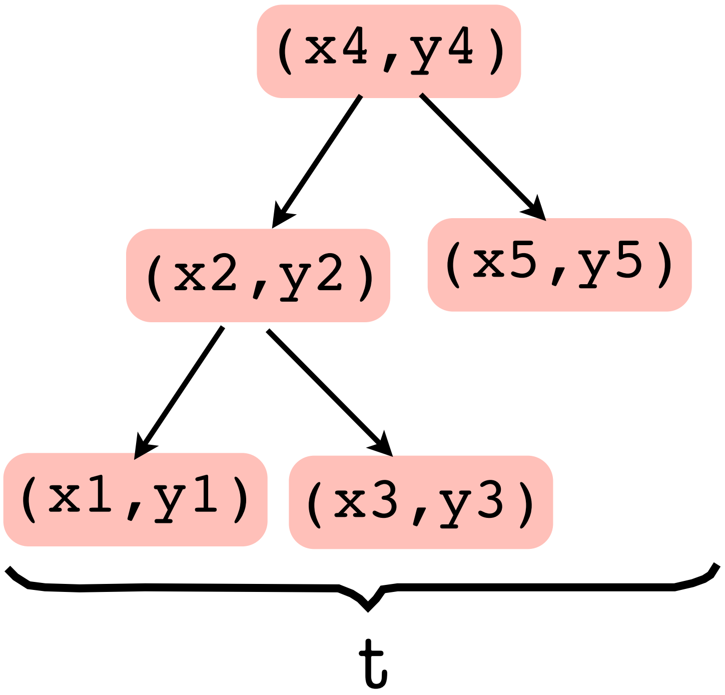
$$R_{fst}(x,y) = \{x\}$$

Now:

$$R_{io}[R_{fst}](t) = \{R_{fst}(x_i, y_i), R_{fst}\,(x_j, y_j)) \mid i \leq j\}$$

$$\Leftrightarrow \quad R_{io}[R_{fst}](t) = \{(x_i,x_j) \mid i \leq j\}$$

For Example ...



We know:

$$R_{io}(t) = \{((x_i, y_i), (x_j, y_j)) \mid i \leq j\}$$

By Definition:

$$R_{io}[\rho](t) = \{(\rho(x_i, y_i), \rho\,(x_j, y_j)) \mid i \leq j\}$$

Let R_{fst} be a relation on pairs, such that

$$R_{fst}(x,y) = \{x\}$$

Now:

$$R_{io}[R_{fst}](t) = \{R_{fst}(x_i, y_i), R_{fst}\,(x_j, y_j)) \mid i \leq j\}$$

$$\Leftrightarrow \quad R_{io}[R_{fst}](t) = \{(x_i,x_j) \mid i \leq j\}$$

in-order among first-components of pairs in t

For Example ...

For Example ...

$$\texttt{treeMap : } \alpha\ \texttt{tree} \longrightarrow (\alpha \longrightarrow \beta) \longrightarrow \beta\ \texttt{tree}$$

For Example ...

$$\texttt{treeMap} : \alpha\ \texttt{tree} \longrightarrow (\alpha \longrightarrow \beta) \longrightarrow \beta\ \texttt{tree}$$

Relational type ...

... by focusing on possible shape invariance between α and β (a la pairMap)

$$(\rho_\alpha,\rho_\beta)\ \texttt{treeMap} : \{t_1 : \alpha\ \texttt{tree}\}$$

$$\longrightarrow (\{x:\alpha\} \longrightarrow \{y:\beta \mid \rho_\beta(y) = \rho_\alpha(x)\})$$

For Example ...

$$\texttt{treeMap} : \alpha\ \texttt{tree} \longrightarrow (\alpha \longrightarrow \beta) \longrightarrow \beta\ \texttt{tree}$$

Relational type ...

... by focusing on possible shape invariance between α and β (a la `pairMap`)

$$(\rho_\alpha, \rho_\beta)\ \texttt{treeMap} : \{t_1 : \alpha\ \texttt{tree}\}$$

$$\longrightarrow (\{x : \alpha\} \longrightarrow \{y : \beta \mid \rho_\beta(y) = \rho_\alpha(x)\})$$

For Example ...

$$\texttt{treeMap} : \alpha\ \texttt{tree} \longrightarrow (\alpha \longrightarrow \beta) \longrightarrow \beta\ \texttt{tree}$$

$$\downarrow$$

Relational type ...

... by focusing on possible shape invariance between α and β (a la `pairMap`)

$$(\rho_\alpha,\rho_\beta)\ \texttt{treeMap} : \{\texttt{t}_1 : \alpha\ \texttt{tree}\}$$

$$\longrightarrow (\{\texttt{x} : \alpha\} \longrightarrow \{\texttt{y} : \beta \mid \rho_\beta(\texttt{y}) = \rho_\alpha(\texttt{x})\})$$

$$\longrightarrow \{\texttt{t}_2 : \beta\ \texttt{tree} \mid ?\ \}$$

For Example ...

$$\texttt{treeMap} : \alpha\ \texttt{tree} \longrightarrow (\alpha \longrightarrow \beta) \longrightarrow \beta\ \texttt{tree}$$

Relational type ... $\Big\downarrow$ ... by focusing on possible shape invariance between α and β (a la `pairMap`)

$$(\rho_\alpha, \rho_\beta)\ \texttt{treeMap} : \{t_1 : \alpha\ \texttt{tree}\}$$
$$\longrightarrow (\{x:\alpha\} \longrightarrow \{y:\beta \mid \rho_\beta(y) = \rho_\alpha(x)\})$$
$$\longrightarrow \{t_2:\beta\ \texttt{tree} \mid ?\ \}$$

$$R_{io}(t_2) \cancel{=} R_{io}(t_1)$$

$R_{io}(t_i)$ is a relation on elements of t_i

and elements of $t_1 \neq$ elements of t_2

For Example ...

$$\texttt{treeMap} : \alpha\ \texttt{tree} \longrightarrow (\alpha \longrightarrow \beta) \longrightarrow \beta\ \texttt{tree}$$

Relational type ...

... by focusing on possible shape invariance between α and β (a la `pairMap`)

$$\begin{aligned}
(\rho_\alpha,\rho_\beta)\ \texttt{treeMap} :\ & \{t_1 : \alpha\ \texttt{tree}\} \\
& \longrightarrow (\{x:\alpha\} \longrightarrow \{y:\beta \mid \rho_\beta(y) = \rho_\alpha(x)\}) \\
& \longrightarrow \{t_2:\beta\ \texttt{tree} \mid \mathcal{R}_{io}[\rho_\beta](t_2) = \mathcal{R}_{io}[\rho_\alpha](t_1)\}
\end{aligned}$$

For Example ...

$$\texttt{treeMap} : \alpha\ \texttt{tree} \longrightarrow (\alpha \longrightarrow \beta) \longrightarrow \beta\ \texttt{tree}$$

Relational type ... ↓ ... by focusing on possible shape invariance between α and β (a la $\texttt{pairMap}$)

$$\begin{aligned}(\rho_\alpha,\rho_\beta)\ \texttt{treeMap} : &\ \{t_1 : \alpha\ \texttt{tree}\}\\ &\longrightarrow (\{x:\alpha\} \longrightarrow \{y:\beta \mid \rho_\beta(y) = \rho_\alpha(x)\})\\ &\longrightarrow \{t_2:\beta\ \texttt{tree} \mid \mathcal{R}_{io}[\rho_\beta](t_2) = \mathcal{R}_{io}[\rho_\alpha](t_1)\}\end{aligned}$$

Parametric in-order relation ($\mathcal{R}_{io}[\rho]$) is not necessarily a relation over elements.

For Example ...

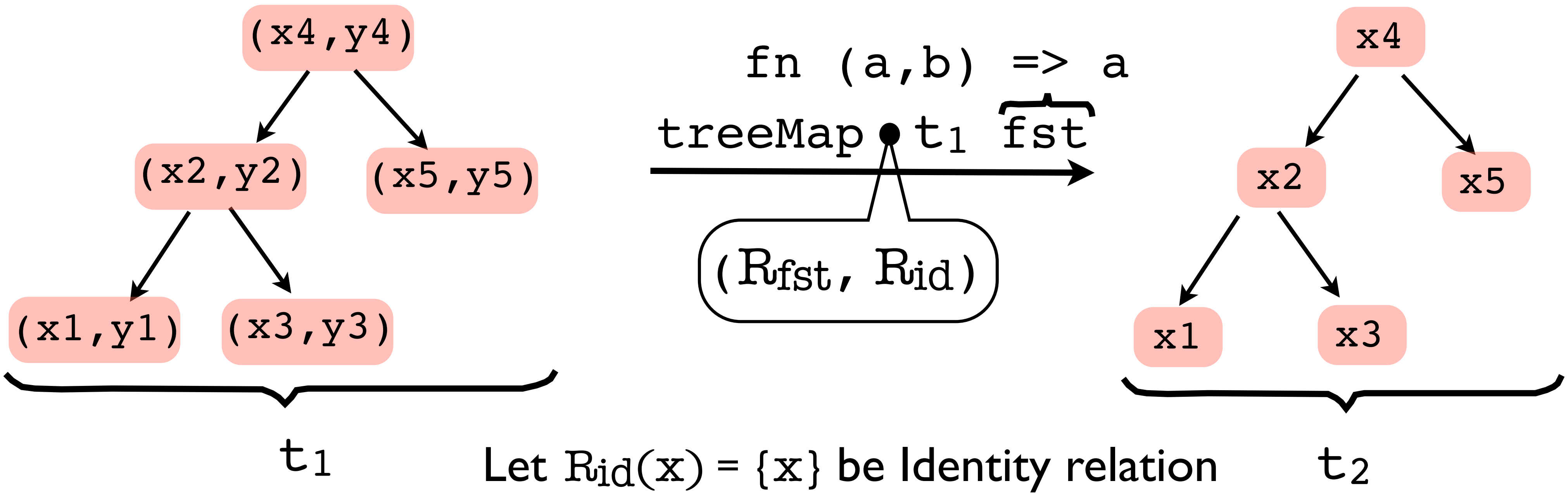
$$
\begin{aligned}
&(\rho_\alpha,\rho_\beta)\ \texttt{treeMap} : \{\texttt{t}_1{:}\alpha\ \texttt{tree}\}\\
&\qquad\qquad \longrightarrow \left(\{\texttt{x}{:}\alpha\} \longrightarrow \{\texttt{y}{:}\beta \mid \rho_\beta(\texttt{y}) = \rho_\alpha(\texttt{x})\}\right)\\
&\qquad\qquad\quad \longrightarrow \{\texttt{t}_2{:}\beta\ \texttt{tree} \mid \mathcal{R}_{\mathrm{io}}[\rho_\beta](\texttt{t}_2) = \mathcal{R}_{\mathrm{io}}[\rho_\alpha](\texttt{t}_1)\}
\end{aligned}
$$

For Example ...

$$
\begin{aligned}
&(\rho_\alpha,\rho_\beta)\ \mathtt{treeMap} : \{t_1{:}\alpha\ \mathtt{tree}\}\\
&\qquad\longrightarrow \big(\{x{:}\alpha\} \longrightarrow \{y{:}\beta \mid \rho_\beta(y) = \rho_\alpha(x)\}\big)\\
&\qquad\quad\longrightarrow \{t_2{:}\beta\ \mathtt{tree} \mid R_{io}[\rho_\beta](t_2) = R_{io}[\rho_\alpha](t_1)\}
\end{aligned}
$$

For Example ...

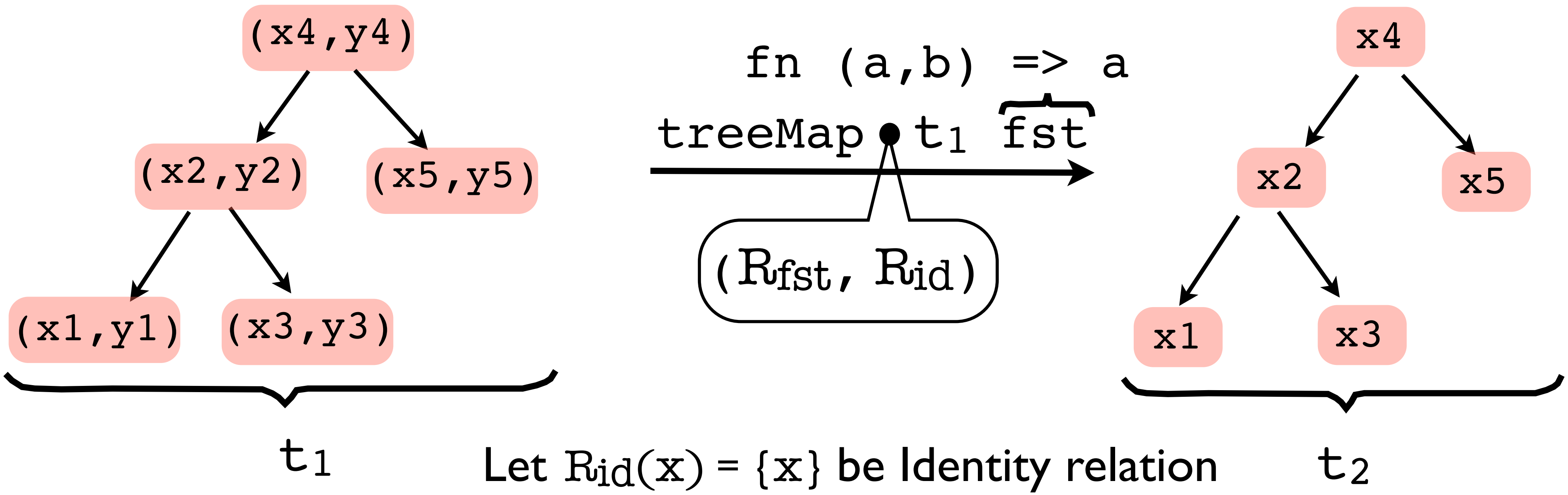
$$\texttt{treeMap}\ (\mathcal{R}_{fst},\ \mathcal{R}_{id}) : \{t_1 : \alpha\ \texttt{tree}\}$$
$$\longrightarrow (\{x:\alpha\} \longrightarrow \{y:\beta \mid \mathcal{R}_{id}(y) = \mathcal{R}_{fst}(x)\})$$
$$\longrightarrow \{t_2:\beta\ \texttt{tree} \mid \mathcal{R}_{io}[\mathcal{R}_{id}](t_2) = \mathcal{R}_{io}[\mathcal{R}_{fst}](t_1)\}$$



Let $\mathcal{R}_{id}(x) = \{x\}$ be Identity relation

For Example ...

$$\begin{aligned}&\texttt{treeMap}\ (\mathcal{R}_{fst},\ \mathcal{R}_{id}) : \{t_1 : \alpha\ \texttt{tree}\}\\ &\quad \longrightarrow (\{x:\alpha\} \longrightarrow \{y:\beta \mid \mathcal{R}_{id}(y) = \mathcal{R}_{fst}(x)\})\\ &\qquad \longrightarrow \{t_2:\beta\ \texttt{tree} \mid \mathcal{R}_{io}[\mathcal{R}_{id}](t_2) = \mathcal{R}_{io}[\mathcal{R}_{fst}](t_1)\}\end{aligned}$$



Let $\mathcal{R}_{id}(x) = \{x\}$ be Identity relation

in-order among elements of t_2 = in-order among first components of pairs in t_1

So far ...

So far ...

- Relational language to express shapes

So far ...

- Relational language to express shapes
- Functions parameterized on relations

So far ...

- Relational language to express shapes
- Functions parameterized on relations
- Relations parameterized on relations

So far ...

- Relational language to express shapes
- Functions parameterized on relations
- Relations parameterized on relations

Expressive type language

So far ...

- Relational language to express shapes
- Functions parameterized on relations
- Relations parameterized on relations

Expressive type language

For type-based shape analysis to be effective, we need type checking with such expressive types to be decidable and practical

Decidability

Type checking is decidable if type refinements can be encoded in a decidable logic

$$\frac{\Gamma \vdash \{\nu : T \mid \phi_1\} \qquad \Gamma \vdash \{\nu : T \mid \phi_2\} \qquad [\![\Gamma_R]\!] \models [\![\Gamma, \nu : T]\!] \Rightarrow [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!]}{\Gamma \vdash \{\nu : T \mid \phi_1\} <: \{\nu : T \mid \phi_2\}}$$

i.e., if ϕ is a type refinement, then $[\![\phi]\!]$ must be an expression in a decidable logic

For the language of relational type refinements,
there exists such an encoding into a decidable
subset of many-sorted first-order logic (MSFOL)

$\Rightarrow$

Type checking is decidable

MSFOL

Many-sorted first-order logic is a syntactic extension of first-order logic with sorts (types)

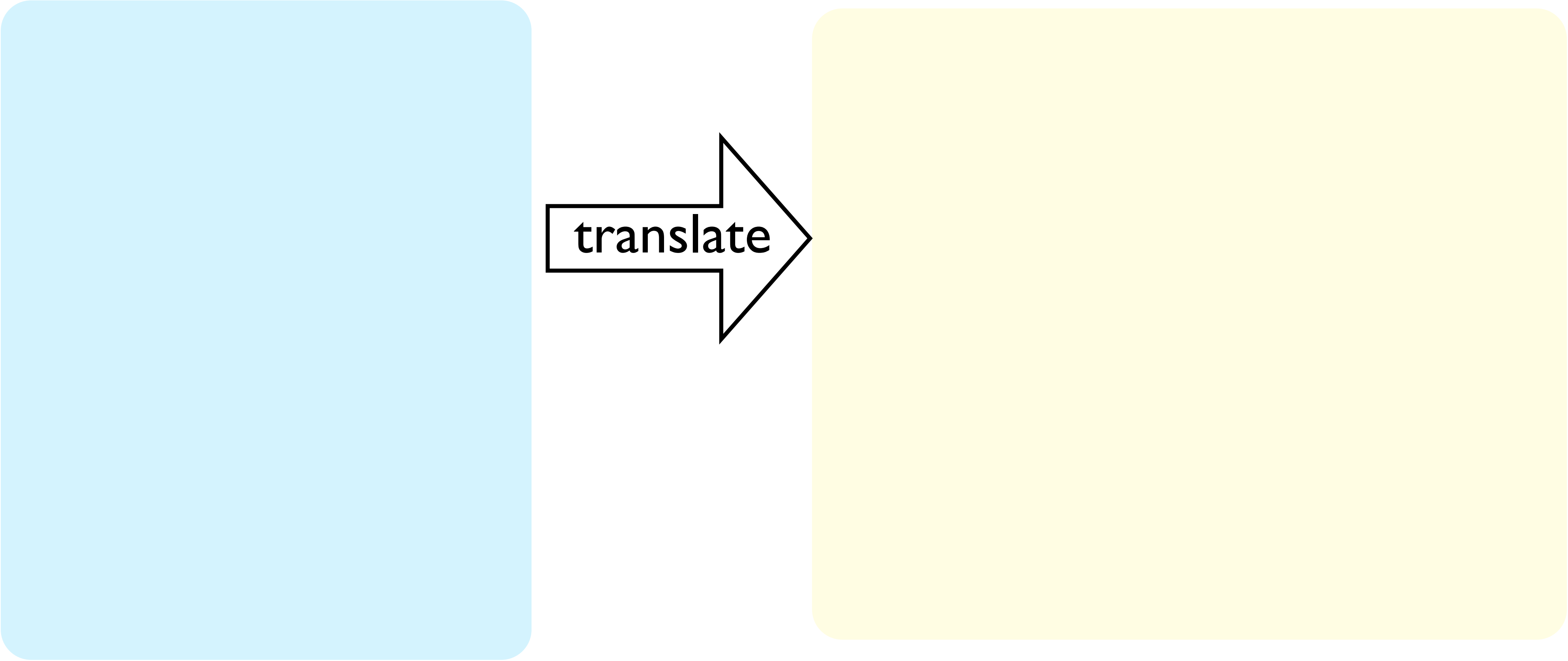
We consider a decidable subset with ...

Effectively Propositional (EPR) MSFOL

Uninterpreted sorts	$\mathtt{T_0,\ T_1,\ \ldots}$
Sorted variables	$\mathtt{x{:}T_0,\ y{:}T_1,\ \ldots}$
Sorted uninterpreted boolean functions (relations)	$\mathtt{R{:}T_0 \longrightarrow bool\ \ \ldots}$
Prenex quantification over sorted variables	$\mathtt{\forall(k{:}T_0).R(x,k) \Leftrightarrow x{=}k,}$ $\mathtt{\exists(j{:}T_0).f(y) = j}$

Encoding ...

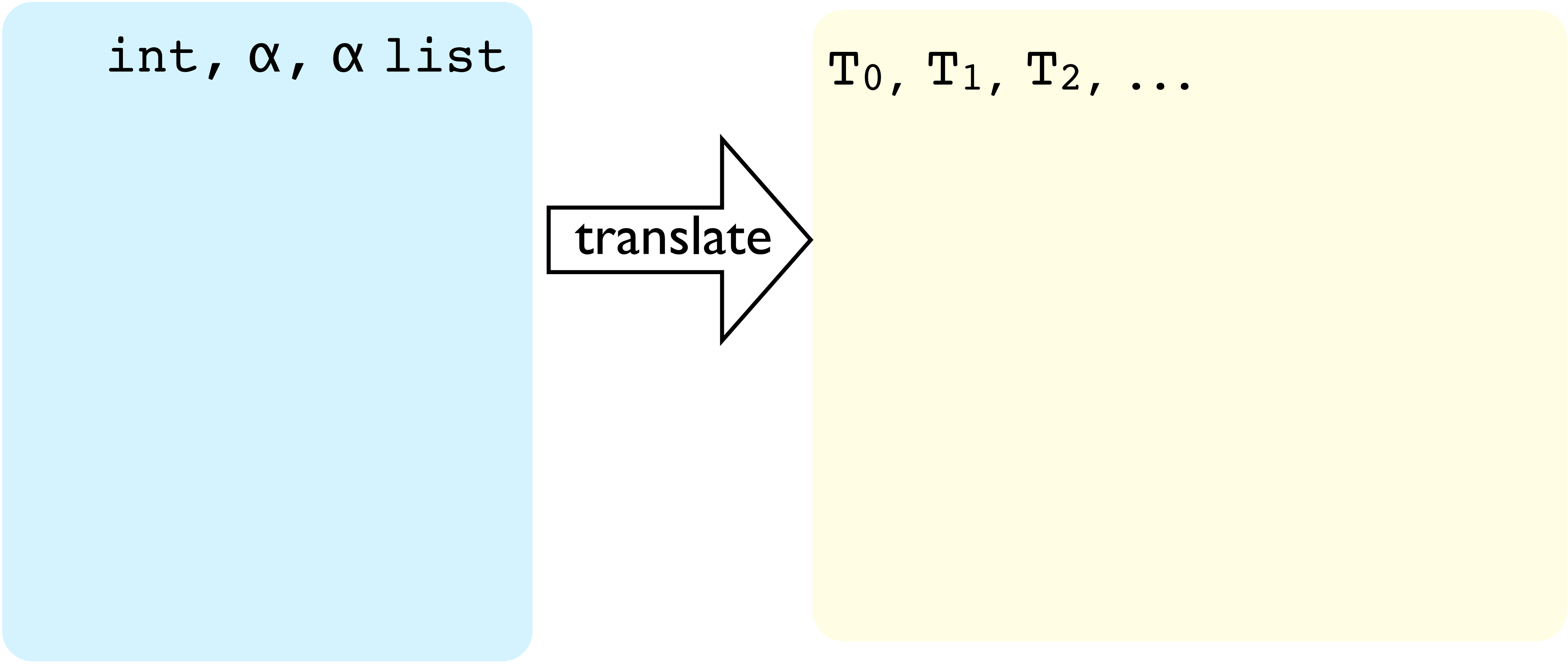
... is translation of artifacts of type refinement language into the EPR fragment of MSFOL.



translate

Encoding ...

... is translation of artifacts of type refinement language into the EPR fragment of MSFOL.



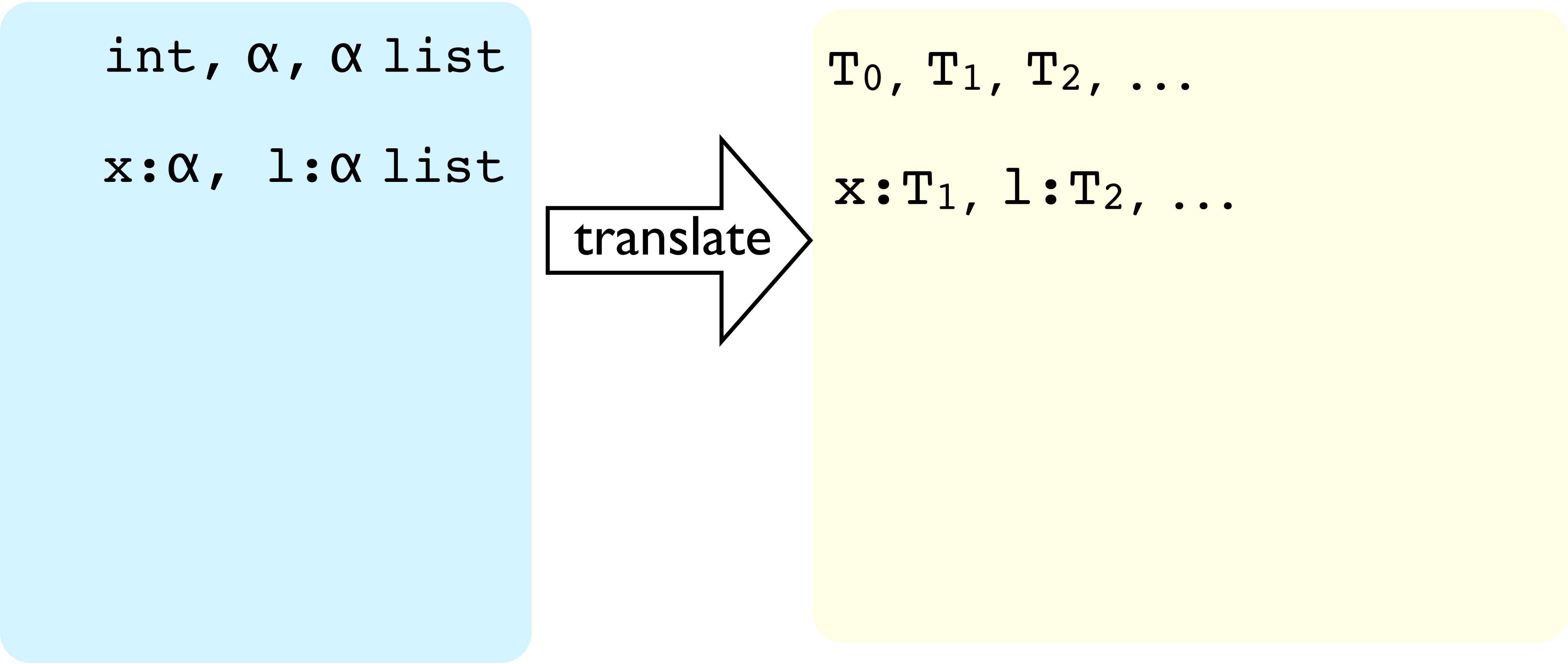
`int, α, α list`

translate

T_0, T_1, T_2, ...

Encoding ...

... is translation of artifacts of type refinement language into the EPR fragment of MSFOL.



`int`, α, α `list`

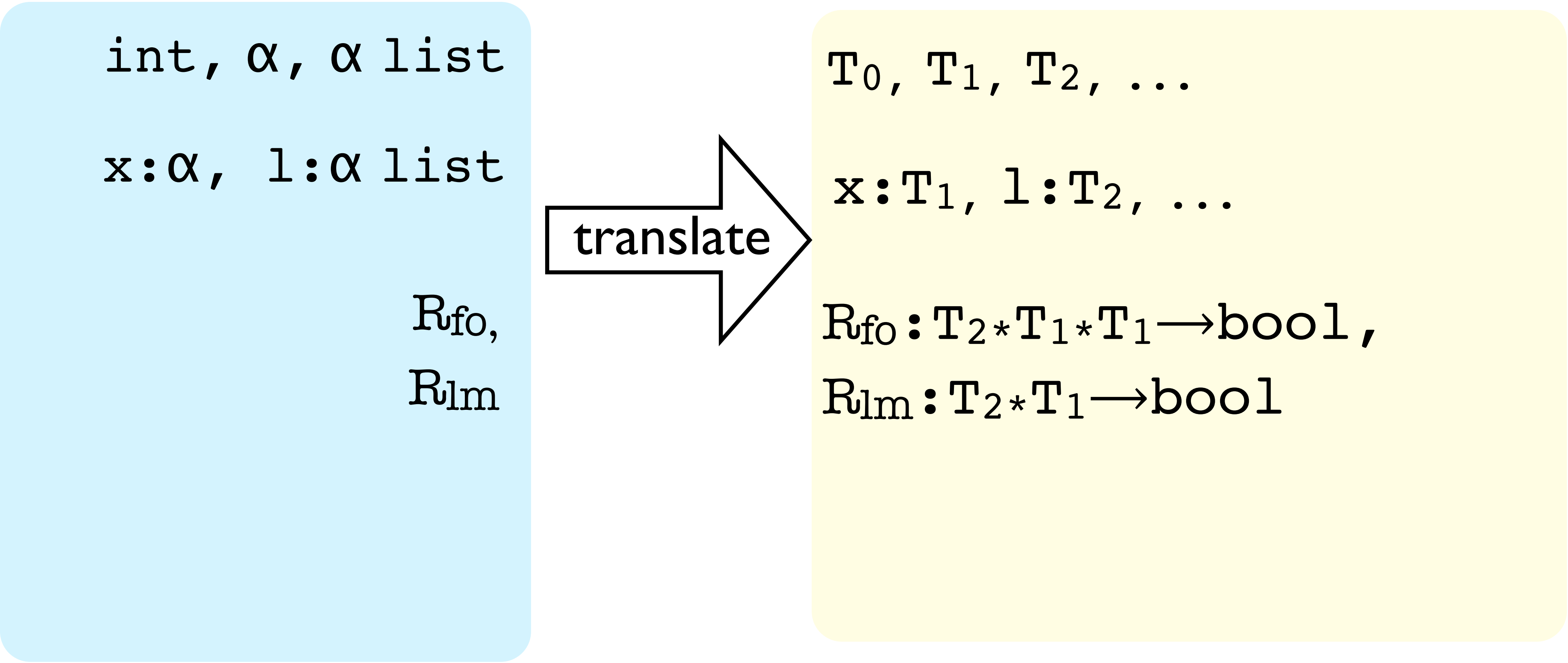
`x`:α, `l`:α `list`

$\mathtt{T}_0$, $\mathtt{T}_1$, $\mathtt{T}_2$, ...

$\mathtt{x}$:$\mathtt{T}_1$, $\mathtt{l}$:$\mathtt{T}_2$, ...

Encoding ...

... is translation of artifacts of type refinement language into the EPR fragment of MSFOL.



`int, α, α list`

`x:α, l:α list`

R_{fo},

R_{lm}

translate

`T0, T1, T2, ...`

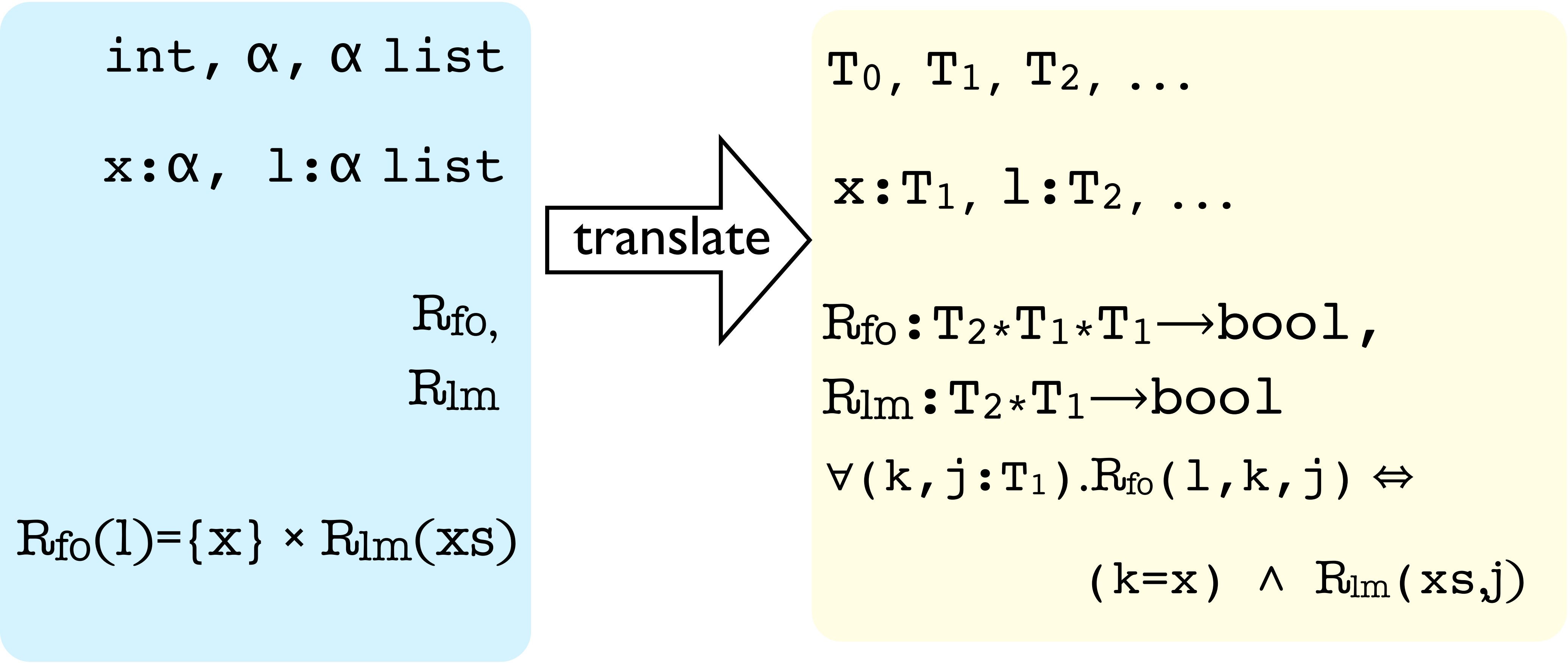
`x:T1, l:T2, ...`

R_{fo} `:T2*T1*T1`$\longrightarrow$`bool,`

R_{lm} `:T2*T1`$\longrightarrow$`bool`

Encoding ...

... is translation of artifacts of type refinement language into the EPR fragment of MSFOL.



$\texttt{int, α, α list}$

$\texttt{x:α, l:α list}$

$\mathcal{R}_{fo}$,

$\mathcal{R}_{lm}$

$\mathcal{R}_{fo}(l)=\{x\} \times \mathcal{R}_{lm}(xs)$

translate

$T_0, T_1, T_2, \ldots$

$x{:}T_1,\ l{:}T_2, \ldots$

$\mathcal{R}_{fo} : T_2 * T_1 * T_1 \longrightarrow \texttt{bool},$

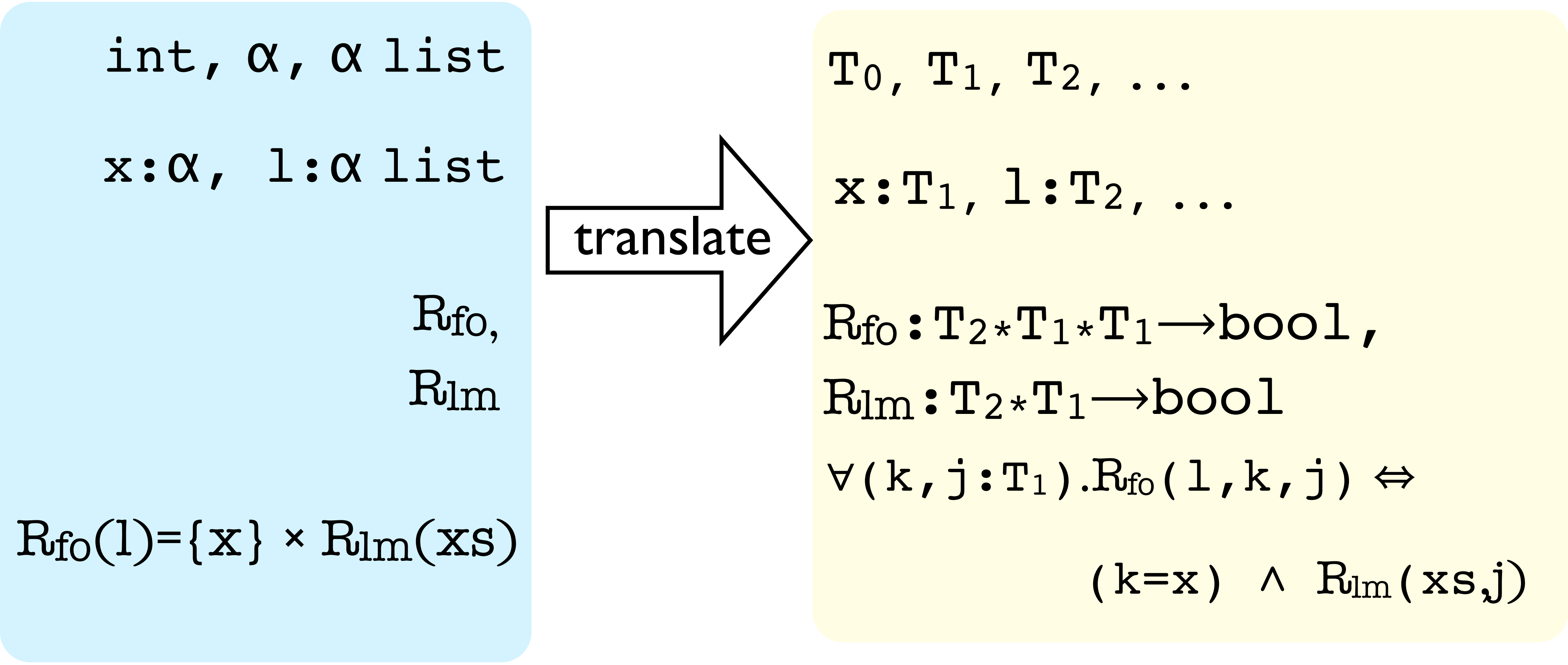
$\mathcal{R}_{lm} : T_2 * T_1 \longrightarrow \texttt{bool}$

$\forall(k,j{:}T_1).\mathcal{R}_{fo}(l,k,j) \Leftrightarrow$

$(k=x)\ \wedge\ \mathcal{R}_{lm}(xs,j)$

Encoding ...

... is translation of artifacts of type refinement language into the EPR fragment of MSFOL.



int, α, α list

x:α, l:α list

$\mathcal{R}_{fo}$,

$\mathcal{R}_{lm}$

$\mathcal{R}_{fo}(l)=\{x\} \times \mathcal{R}_{lm}(xs)$

translate

T_0, T_1, T_2, ...

x:T_1, l:T_2, ...

$\mathcal{R}_{fo} : T_2 * T_1 * T_1 \longrightarrow$ bool,

$\mathcal{R}_{lm} : T_2 * T_1 \longrightarrow$ bool

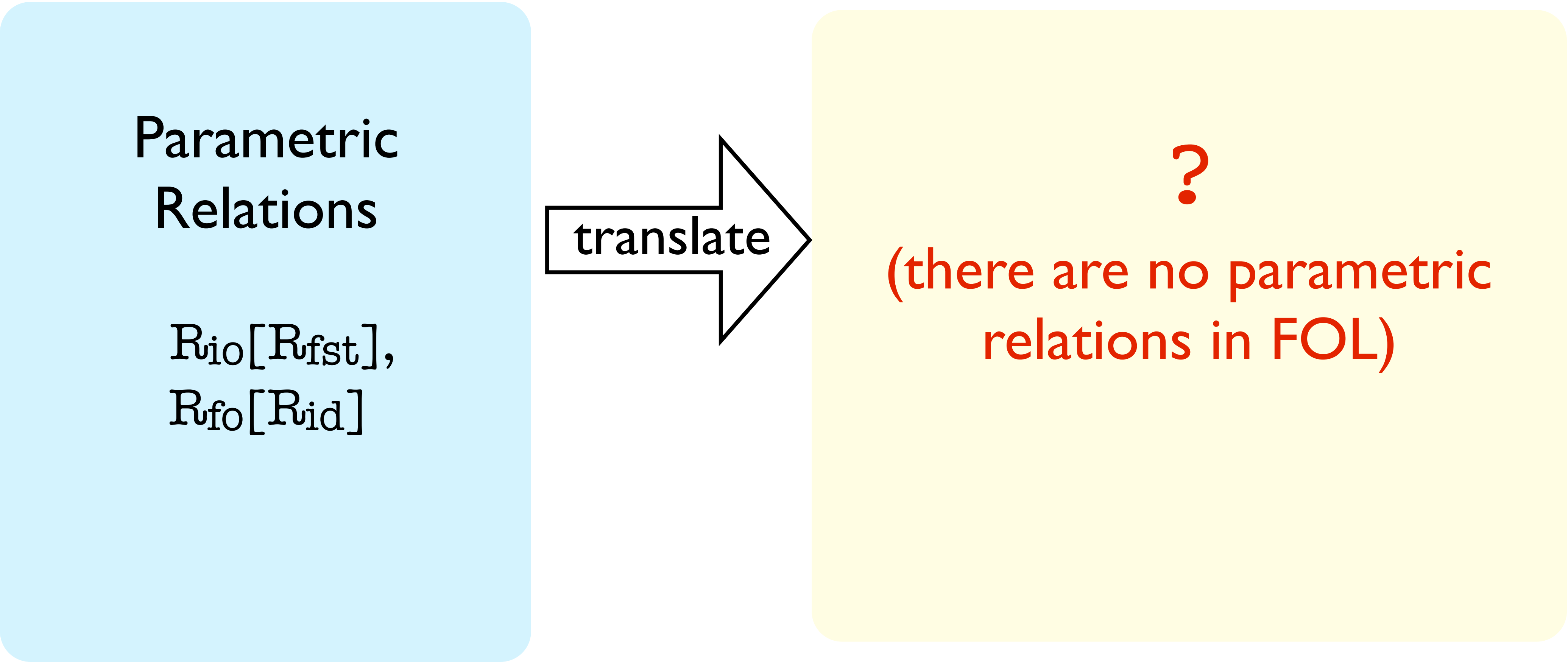
$\forall (k, j : T_1).\mathcal{R}_{fo}(l, k, j) \Leftrightarrow$

$(k=x) \wedge \mathcal{R}_{lm}(xs, j)$

but ...

Encoding ...

... parametric relations is not straightforward



Parametric Relations

$R_{io}[R_{fst}]$, $R_{fo}[R_{id}]$

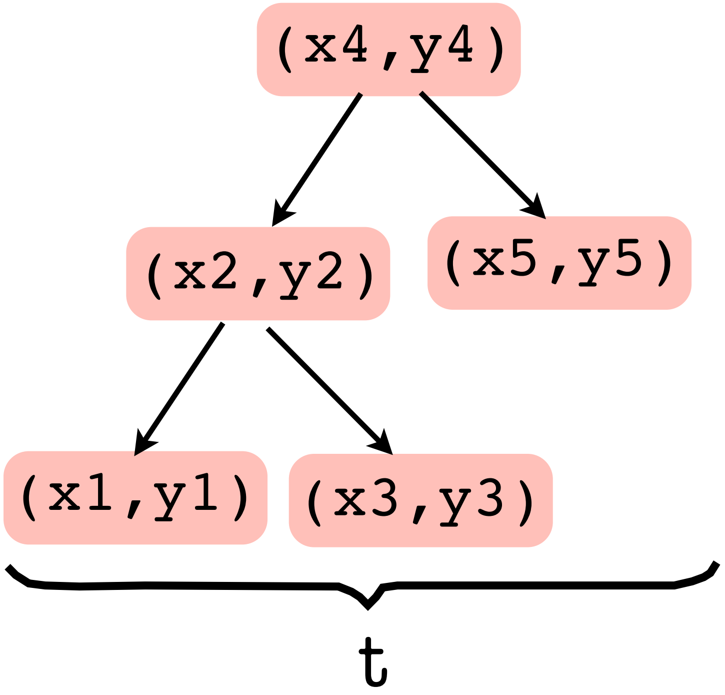
translate

?

(there are no parametric relations in FOL)

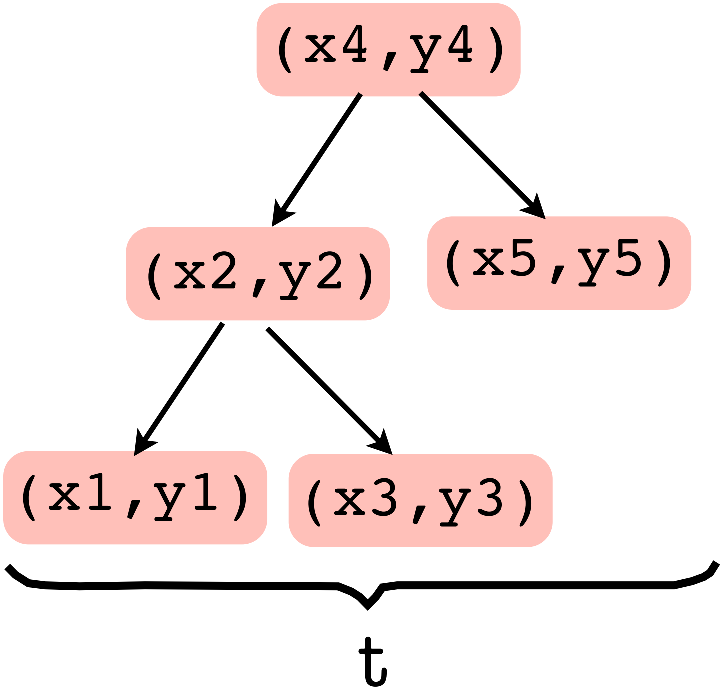
A fully instantiated parametric relation can be defined in terms of its component non-parametric relations

For eg:



A fully instantiated parametric relation can be defined in terms of its component non-parametric relations

For eg:



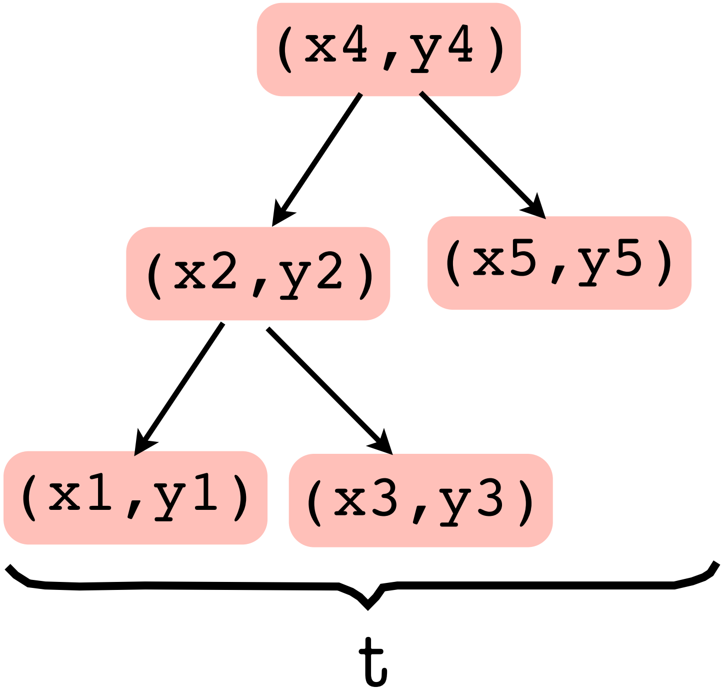
We have already seen:

$$R_{io}(t) = \{((x_i, y_i), (x_j, y_j)) \mid i \leq j\}$$

$$R_{io}[R_{fst}](t) = \{(x_i, x_j) \mid i \leq j\}$$

A fully instantiated parametric relation can be defined in terms of its component non-parametric relations

For eg:



We have already seen:

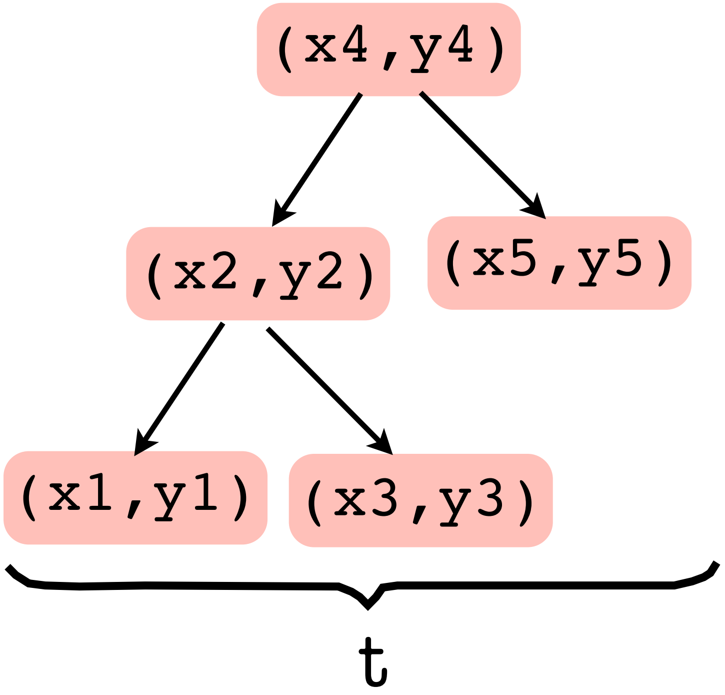
$$R_{io}(t) = \{((x_i, y_i), (x_j, y_j)) \mid i \leq j\}$$

R_{fst}

$$R_{io}[R_{fst}](t) = \{(x_i, x_j) \mid i \leq j\}$$

A fully instantiated parametric relation can be defined in terms of its component non-parametric relations

For eg:



We have already seen:

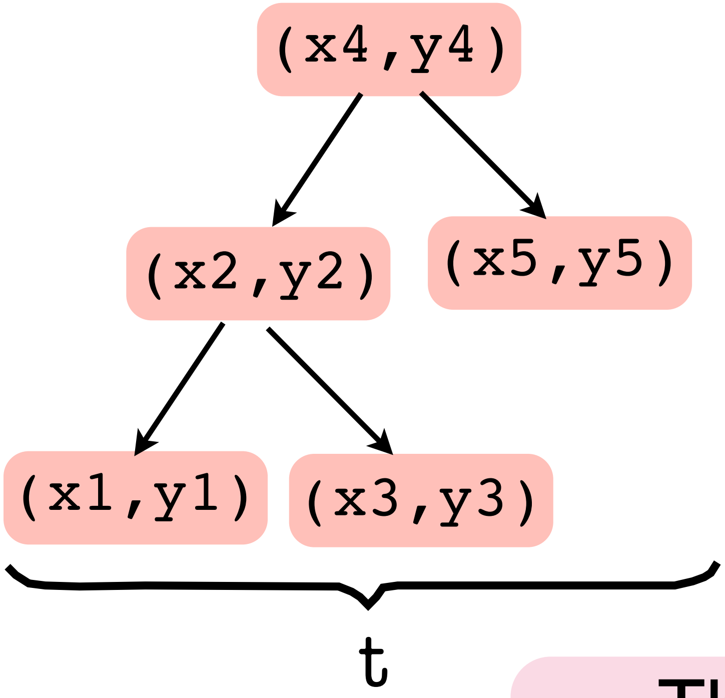
$$R_{io}(t) = \{((x_i, y_i), (x_j, y_j)) \mid i \leq j\}$$

$$\begin{tikzcd} (x_i, y_i) \arrow[d, "R_{fst}"'] & (x_j, y_j) \arrow[d, "R_{fst}"] \\ x_i & x_j \end{tikzcd}$$

$$R_{io}[R_{fst}](t) = \{(x_i, x_j) \mid i \leq j\}$$

A fully instantiated parametric relation can be defined in terms of its component non-parametric relations

For eg:



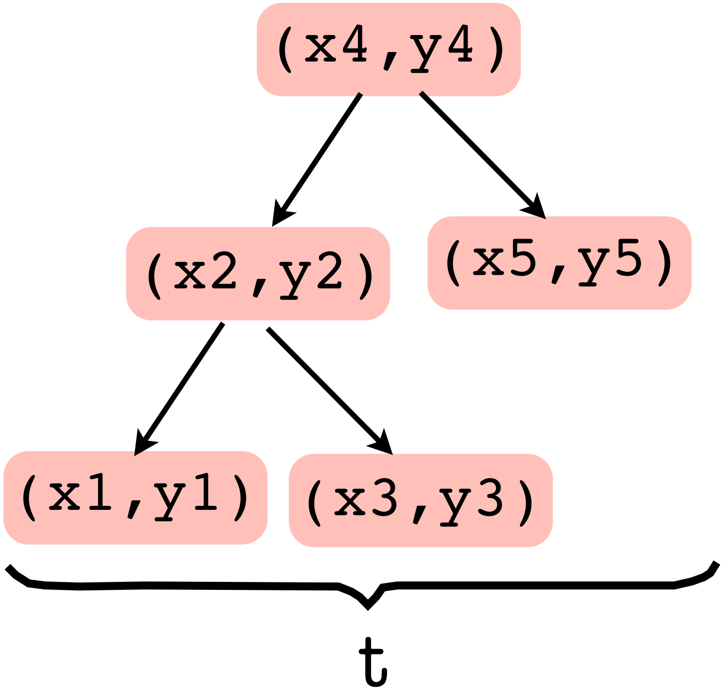
We have already seen:

$$\begin{tikzcd} R_{io}(t) = \{(& (x_i, y_i), \arrow[d, "R_{fst}"'] & (x_j, y_j) \arrow[d, "R_{fst}"] &) \mid i \leq j\} \\ R_{io}[R_{fst}](t) = \{(& x_i, & x_j &) \mid i \leq j\} \end{tikzcd}$$

The set $R_{io}[R_{fst}](t)$ is obtained from the set $R_{io}(t)$ by mapping both components of pairs with R_{fst}

A fully instantiated parametric relation can be defined in terms of its component non-parametric relations

For eg:



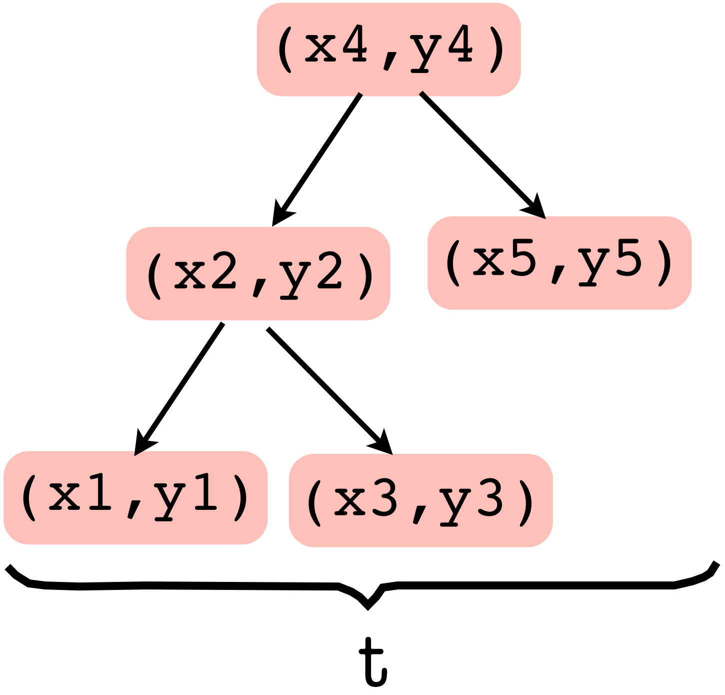
We have already seen:

$$R_{io}(t) = \{((x_i, y_i), (x_j, y_j)) \mid i \leq j\}$$

$$R_{io}[R_{fst}](t) = \{(x_i, x_j) \mid i \leq j\}$$

A fully instantiated parametric relation can be defined in terms of its component non-parametric relations

For eg:



We have already seen:

$$R_{io}(t) = \{((x_i, y_i), (x_j, y_j)) \mid i \leq j\}$$

$$R_{io}[R_{fst}](t) = \{(x_i, x_j) \mid i \leq j\}$$

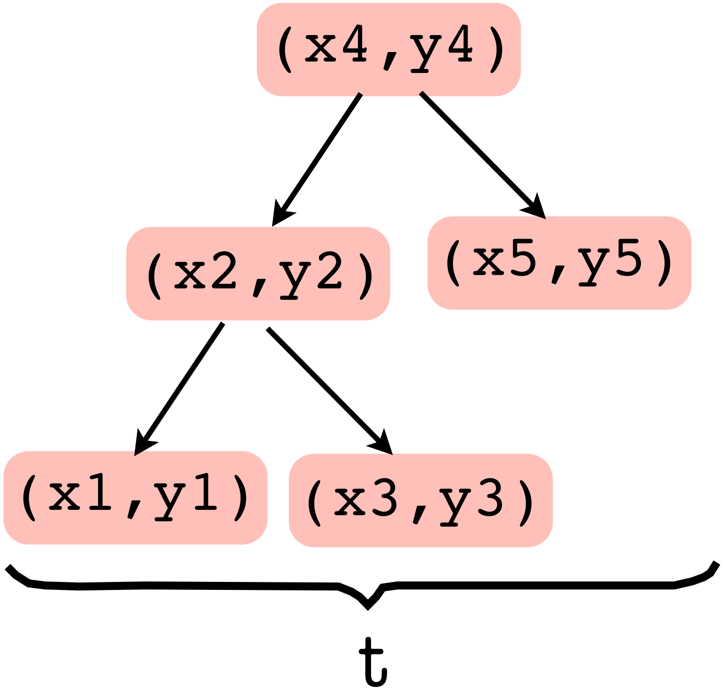
$$\Leftrightarrow$$

$$R_{io}[R_{fst}](t) =$$

$$\{(R_{fst}(a), R_{fst}(b)) \mid (a,b) \in R_{io}(t)\}$$

A fully instantiated parametric relation can be defined in terms of its component non-parametric relations

For eg:



We have already seen:

$$R_{io}(t) = \{((x_i, y_i), (x_j, y_j)) \mid i \leq j\}$$

$$R_{io}[R_{fst}](t) = \{(x_i, x_j) \mid i \leq j\}$$

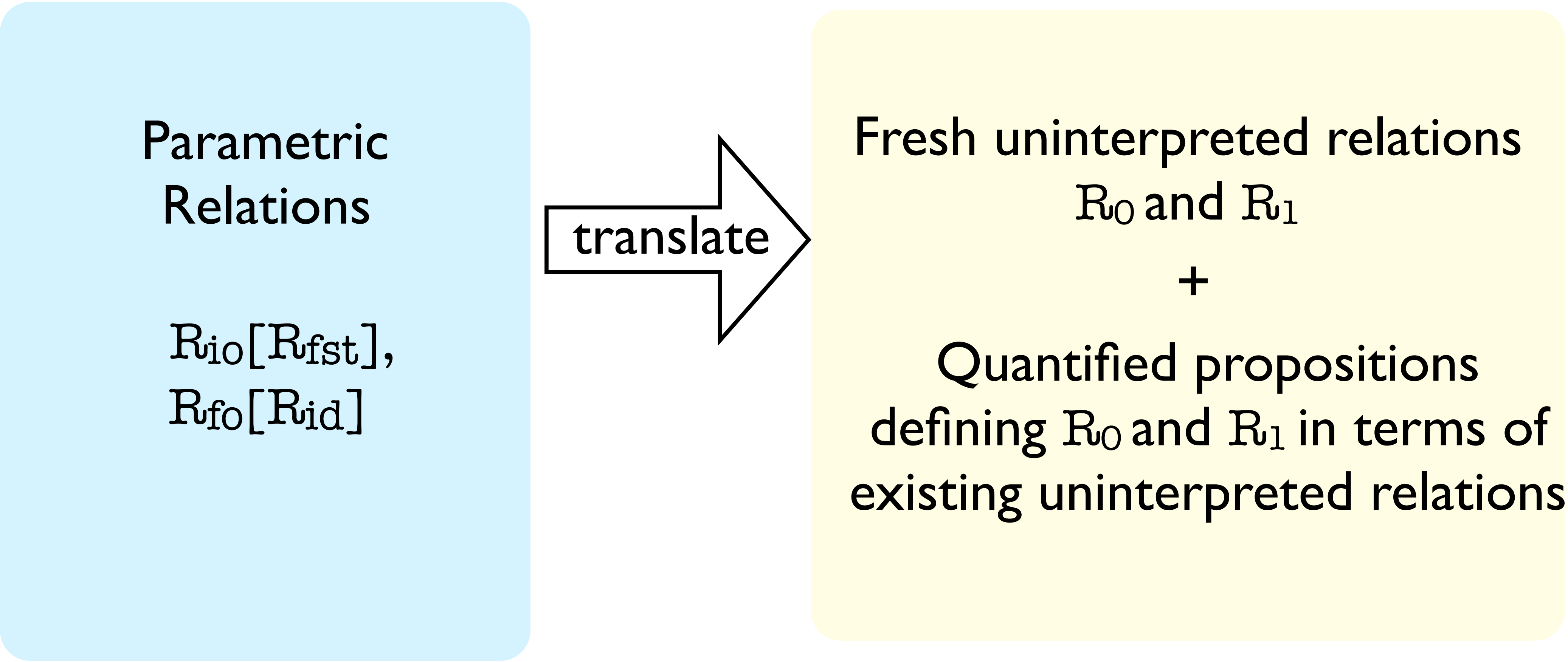
$$\Leftrightarrow$$

$$R_{io}[R_{fst}](t) = \{(R_{fst}(a), R_{fst}(b)) \mid (a,b) \in R_{io}(t)\}$$

Defines $R_{io}[R_{fst}]$ in terms of R_{io} and R_{fst}

Encoding ...

... parametric relations by defining them in terms of their component non-parametric relations



Parametric Relations

$R_{io}[R_{fst}]$, $R_{fo}[R_{id}]$

translate

Fresh uninterpreted relations R_0 and R_1

+

Quantified propositions defining R_0 and R_1 in terms of existing uninterpreted relations

Off-the-shelf SMT solvers (eg: Z3) are efficient decision procedures for the EPR fragment of MSFOL.

Off-the-shelf SMT solvers (eg: Z3) are efficient decision procedures for the EPR fragment of MSFOL.

⇒

A practical type checker can be constructed by encoding type refinements in MSFOL and using SMT solvers for subtype checking.

Off-the-shelf SMT solvers (eg: Z3) are efficient decision procedures for the EPR fragment of MSFOL.

CATALYST ⇒

A $\overbrace{\text{practical type checker}}$ can be constructed by encoding type refinements in MSFOL and using SMT solvers for subtype checking.

CATALYST

Implemented as extended type checking pass in MLton Standard ML compiler

CATALYST

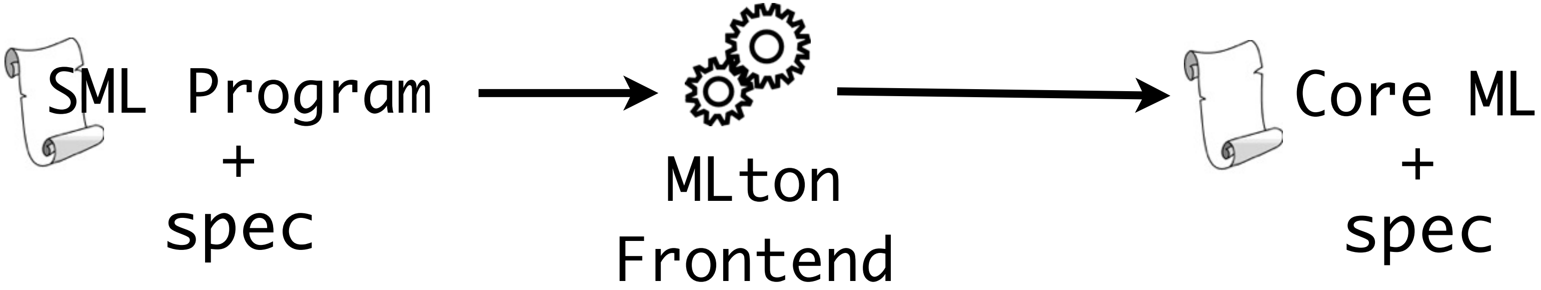
Implemented as extended type checking pass in MLton Standard ML compiler

SML Program

+

spec

Implemented as extended type checking pass in MLton Standard ML compiler



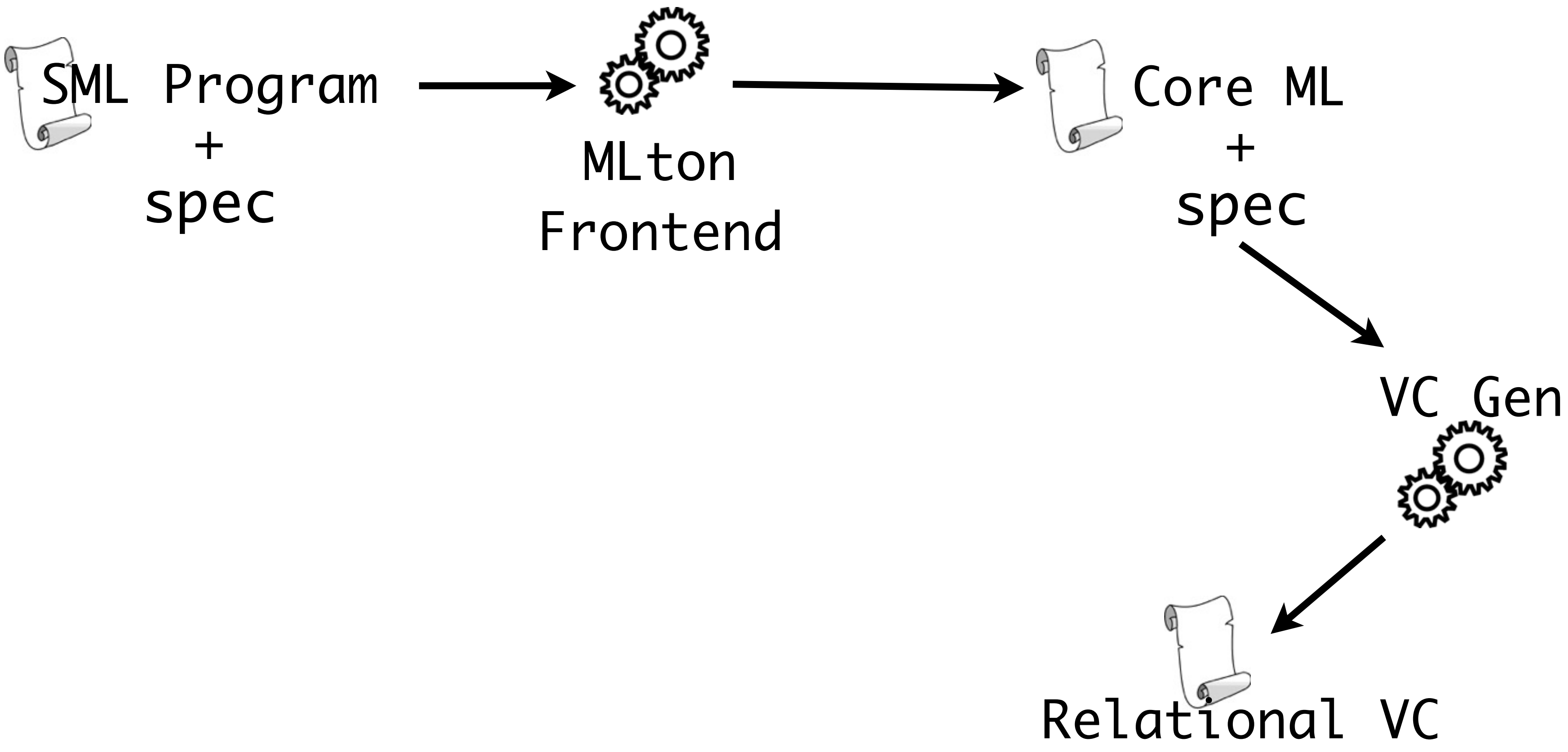
SML Program
+
spec

MLton
Frontend

Core ML
+
spec

CATALYST

Implemented as extended type checking pass in MLton Standard ML compiler



SML Program
+
spec

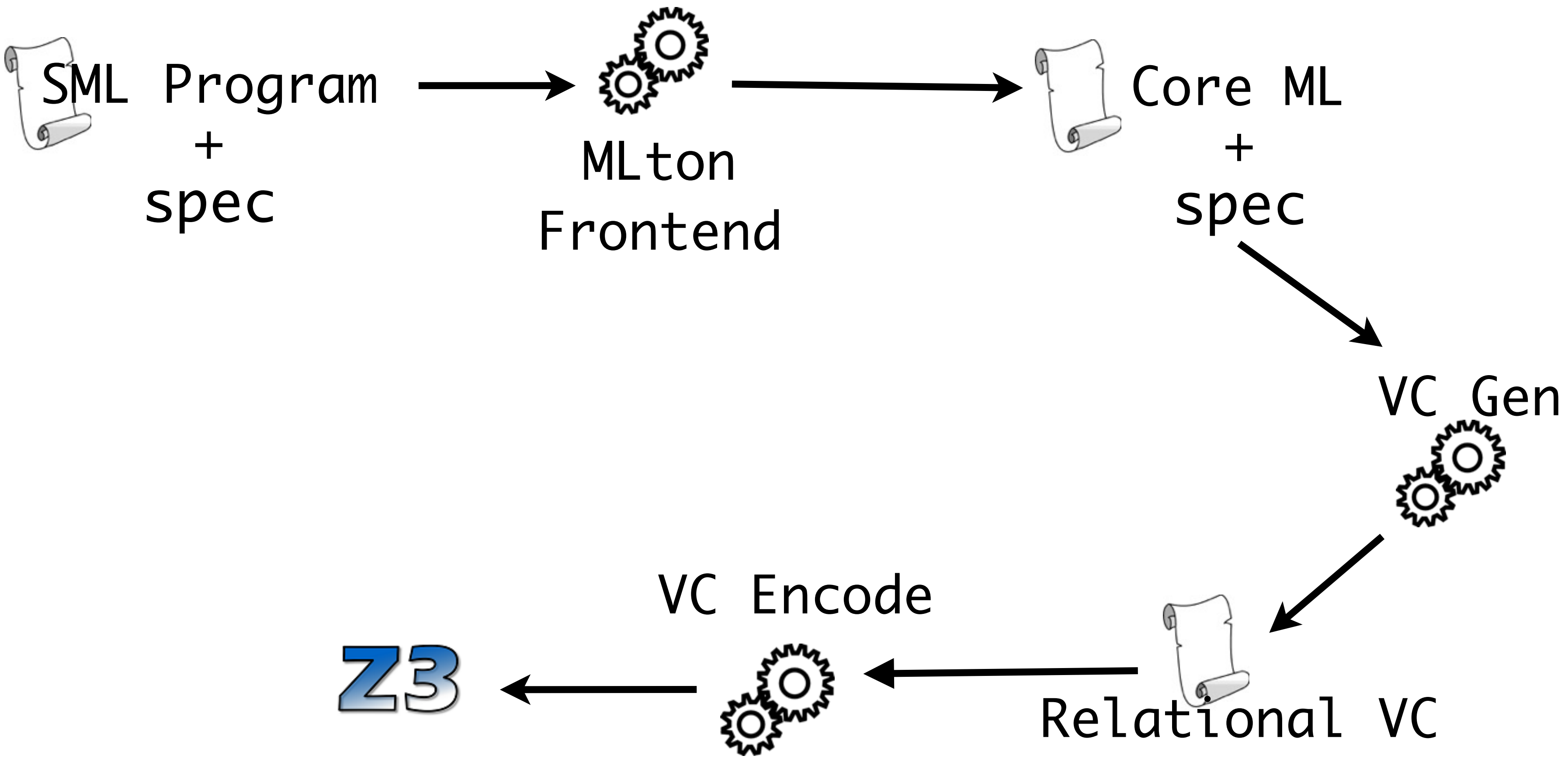
MLton
Frontend

Core ML
+
spec

VC Gen

Relational VC

Implemented as extended type checking pass in MLton Standard ML compiler



SML Program + spec

MLton Frontend

Core ML + spec

VC Gen

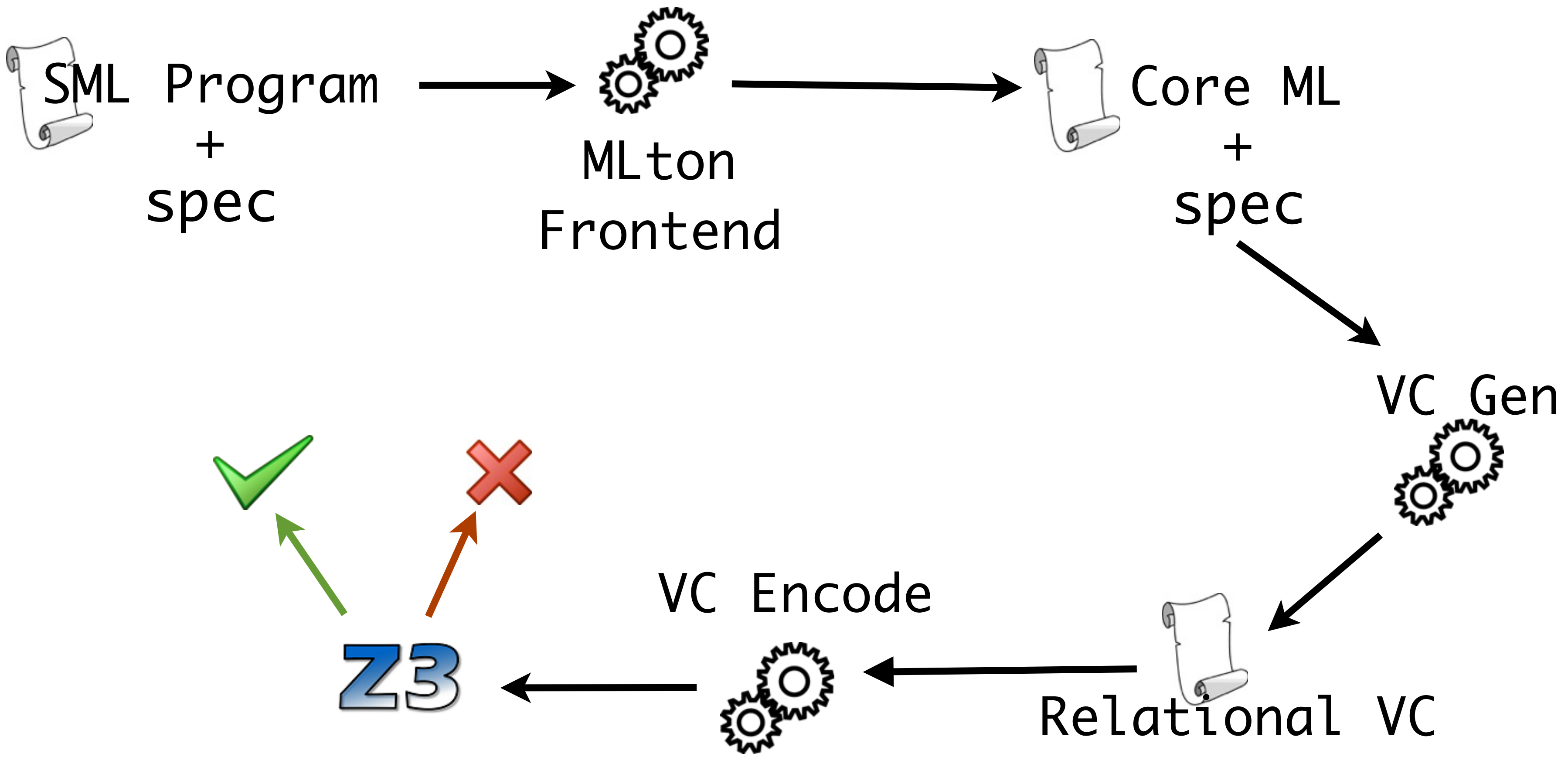
Relational VC

VC Encode

Z3

CATALYST

Implemented as extended type checking pass in MLton Standard ML compiler



SML Program
+
spec

MLton
Frontend

Core ML
+
spec

VC Gen

Relational VC

VC Encode

Z3

Validation

Lists	Okasaki trees	Functional Graphs	MLton functions
rev	inOrder	folds	alpha-rename
concat	preOrder	traversals	substitutions
map	postOrder	maps	SSA
foldl	treefoldl	⋮	
foldr	treefoldr		
exists	balance		
filter	rotate		
⋮	⋮		

Validation

Lists	Okasaki trees	Functional Graphs	MLton functions
rev	inOrder	folds	alpha-rename
concat	preOrder	traversals	substitutions
map	postOrder	maps	SSA
foldl	treefoldl	⋮	
foldr	treefoldr		
exists	balance		
filter	rotate		
⋮	⋮		

✓

Related Work

GADTs in OCaml and Haskell

Type refinements in F*

Abstract refinements in Liquid Types

Logical Relations

Shape analysis for higher-order control flow

Conclusions

Marriage of a relational specification language with a dependent type system capable of describing expressive structural invariants of functional data structures

Future Directions

- Extensions to deal with non-inductive structures
- Automated inference
- Basis for "lightweight" verified compilation

`https://github.com/tycon/catalyst`