Relational Refinement Types for Higher-Order Shape Transformers

Suresh Jagannathan
Joint work with Gowtham Kaki
In imperative settings, shape analysis is concerned with discovering/verifying the shape of a pointer into memory

\[ p = \text{LinkedList} \]
In functional languages, we have types

\[
p = \texttt{Cons(.,Cons(.,\texttt{Nil})})
\]

\[
p = \texttt{B(B(L,.,L),.,B(L,.,L))}
\]

\[
p : \alpha \text{ list}
\]

\[
p : \alpha \text{ tree}
\]
In functional languages, we have types

\[ f : \alpha \text{ tree} \rightarrow \alpha \text{ list} \]
In functional languages, we have have types

$$f : \alpha \text{ tree} \rightarrow \alpha \text{ list}$$
In functional languages, we have types

\[ f : \alpha \text{ tree} \rightarrow \alpha \text{ list} \]
In functional languages, we have types

\[ f : \alpha \text{ tree} \rightarrow \alpha \text{ list} \]

How can we use types to express precise shape information?

\[ f : \{ t : \alpha \text{ tree} \} \rightarrow \{ l : \alpha \text{ list} \mid \varphi \} \]

\[ \varphi \iff \text{SomeShape}(l) \equiv \text{SomeOtherShape}(t) \]

type refinement predicate
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

\[
\text{rev} : \{l : \text{'a list}\} \rightarrow \{\nu : \text{'a list} \mid \nu = \text{rev'}(l)\}
\]

fun \text{rev} [\ ] = [\ ]
    | \text{rev} x::xs = \text{concat} (\text{rev} xs) [x]
Example

\[
\text{rev} : \{l : \text{'a list}\} \rightarrow \{\nu : \text{'a list} \mid \nu = \text{rev' (l)}\}
\]

fun \text{rev} [\] = []
    | \text{rev} x::xs = \text{concat (rev xs)} [x]

reasoning about \text{rev'} likely as complex as directly reasoning about \text{rev}
Example

\[
\text{rev : } \{ l : 'a \text{ list} \} \rightarrow \{ \nu : 'a \text{ list} \mid \nu = \text{rev'}(l) \}
\]

\[
\text{fun \ qrev \ []} = []
| \ qrev \ x::xs = \text{concat} (\text{rev} \ xs) \ [x]
\]

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

\[ \text{inOrder} : \{t: \alpha \text{ tree}\} \rightarrow \{l: \alpha \text{ list} | \varphi\} \]

\[ \varphi \Leftrightarrow \text{forward-order}(l) = \text{in-order}(t) \]
Post-Order

x1 -> x2 -> x3 -> x4 -> x5
postOrder : \{ t:α tree \} → \{ l:α list | φ \}

φ ⇔ forward-order(l) = post-order(t)
rotate : \{t_1: \alpha \text{ tree}\} \rightarrow \{t_2: \alpha \text{ tree}|\varphi\}

\varphi \Leftrightarrow \text{in-order}(t_1) = \text{post-order}(t_2)
Reverse

\[ x_1 \ x_2 \ x_3 \ x_4 \ x_5 \]

\[ x_5 \ x_4 \ x_3 \ x_2 \ x_1 \]
Reverse

\[
\text{rev} : \{l_1: \alpha \text{ list}\} \rightarrow \{l_2: \alpha \text{ list} | \varphi\}
\]

\[
\varphi \iff \text{backward-order}(l_2) = \text{forward-order}(l_1)
\]
We need ...

Type refinements ($\varphi$) to be predicates over an expressive language.
Type refinements ($\varphi$) 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?
What is common among pre-order, post-order, forward-order, and backward-order?

All are orders
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: $\text{in-order}(x_i,x_j) \iff i \leq j$
For Example ...

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

\[R_{\text{io}}(t)\]
For Example ... 

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

\[ R_{\text{io}}(t) = \{(x_i, x_j) \mid i \leq j\} \]
For Example ...

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

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

\begin{array}{ccccc}
  x_1 & x_2 & x_3 & x_4 & x_5 \\
\end{array}
For Example ...

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

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

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

$$x_5 \xrightarrow{\text{fwd-order}} x_4 \xrightarrow{\text{fwd-order}} x_2 \xrightarrow{\text{fwd-order}} x_3 \xrightarrow{\text{fwd-order}} x_1$$
For Example ...

**in-order of** $t$ **is binary relation such that:**

$$\text{in-order}(x_i, x_j) \iff i \leq j$$

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

**fwd-order of** $l$ **is binary relation such that:**

$$\text{fwd-order}(x_i, x_j) \iff i \leq j$$

$$R_{\text{fo}}(l)$$
For Example ...

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

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

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

\[ R_{\text{fo}}(l) = \{(x_i, x_j) \mid i \leq j\} \]
For Example ...

The in-order of tree $t$ is a binary relation such that:

\[
\text{in-order}(x_i, x_j) \iff i \leq j
\]

\[
R_{i0}(t) = \{(x_i, x_j) \mid i \leq j\}
\]

The fwd-order of list $l$ is a binary relation such that:

\[
\text{fwd-order}(x_i, x_j) \iff i \leq j
\]

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

$\Rightarrow$ If list $l$ contains elements of tree $t$ in pre-order, then

\[
R_{fo}(l) = R_{i0}(t)
\]
*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**
  \[ R_{tm}(t) = R_{lm}(l) = \{x_1, x_2, x_3, x_4, x_5\} \]

- **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**

  $R_{tm}(t) = R_{lm}(l) = \{x_1, x_2, x_3, x_4, x_5\}$

- **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:

$\text{tree-members}$

$$R_{tm}(t) = R_{lm}(l) = \{x_1, x_2, x_3, x_4, x_5\}$$

$\text{list-members}$

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**
  
  \[
  \mathcal{R}_{tm}(t) = \mathcal{R}_{lm}(l) = \{x_1, x_2, x_3, x_4, x_5\}
  \]

- **list-members**
  
  \[
  \mathcal{R}_{tm}(lt) = \{x_1, x_2, x_3\}
  \]

They let us write assertions over binary relations like \( \mathcal{R}_{po} \)
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
  \( R_{\text{tm}}(t) = R_{\text{lm}}(l) = \{x_1, x_2, x_3, x_4, x_5\} \)
- list-members
  \( R_{\text{tm}}(lt) = \{x_1, x_2, x_3\} \)

They let us write assertions over binary relations like \( R_{\text{po}} \)

\[ R_{\text{tm}}(lt) \times \{x_4\} \subset R_{\text{io}}(t) \]
The Language of Relations ...

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

∪ \( R_{fo}(xs) \)
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.

\[ \bigcup R_{fo}(xs) \]
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:

$$\bigcup R_{fo}(xs)$$
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:

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

relation $R_{io}(Tree(L,n,R)) = (R_{tm}(L) \times \{n\}) \cup (\{n\} \times R_{tm}(R)) \cup R_{io}(L) \cup R_{io}(R)$
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:

relation \( R_{fo}(x::xs) = (\{x\} \times R_{mem}(xs)) \cup R_{fo}(xs) \)

relation \( R_{io}(Tree(L,n,R)) = (R_{tm}(L) \times \{n\}) \cup (\{n\} \times R_{tm}(R)) \cup R_{io}(L) \cup R_{io}(R) \)

inOrder : \{t:\alpha\ tree\} \rightarrow \{l:\alpha\ list\ |\ R_{fo}(l) = R_{io}(t)\}

tail : \{l:\alpha\ list\} \rightarrow \{v:\alpha\ list\ |\ R_{fo}(v) \subset R_{fo}(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:

\[
\begin{align*}
\text{id} & : \alpha \rightarrow \alpha \\
\text{pairMap} & : \alpha \times \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \times \beta
\end{align*}
\]
... to facilitate compositional type checking and verification, we should be able to ascribe relational types to polymorphic and higher-order functions.

For eg:

\[
\begin{align*}
\text{id} : & \, \alpha \rightarrow \alpha \\
\text{pairMap} : & \, \alpha \times \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \times \beta
\end{align*}
\]

Relational types for polymorphic and higher-order functions must be general enough to relate different shapes at different call sites.
id : α → α
id : \alpha \rightarrow \alpha
\text{id} : \alpha \rightarrow \alpha
\beta \text{ list} \quad \beta \text{ tree} \quad \text{id can take arguments of unknown shape}

\text{id} : \alpha \rightarrow \alpha
\[ \beta \text{ list} \rightarrow \beta \text{ tree} \]

\[ \text{id : } \alpha \rightarrow \alpha \]

\text{id can take arguments of unknown shape}

\text{Shape of the argument is also the shape of its result}

\[ \text{id : } \{x : \alpha\} \rightarrow \{y : \alpha \mid \text{Shape}(y) = \text{Shape}(x)\} \]
Relational Parameters

\[ \beta \text{ list} \quad \beta \text{ tree} \]

\[ \text{id} : \alpha \rightarrow \alpha \]

\[ \text{id can take arguments of unknown shape} \]

Shape of the argument is also the shape of its result

\[ \text{id} : \{x:\alpha\} \rightarrow \{y:\alpha \mid \text{Shape}(y) = \text{Shape}(x)\} \]
Relational Parameters

\[ \beta \text{ list } \xrightarrow{\beta} \text{ tree } \]

\[ \text{id : } \alpha \rightarrow \alpha \]

\text{id can take arguments of unknown shape}

Denote with an abstract relation

\text{Shape of the argument is also the shape of its result}

\[ \text{id : } \{ x : \alpha \} \rightarrow \{ y : \alpha \mid \text{Shape}(y) = \text{Shape}(x) \} \]

\[ \rho \]
Relational Parameters

\[ \beta \text{ list} \rightarrow \beta \text{ tree} \]

\[ \text{id} : \alpha \rightarrow \alpha \]

\text{id can take arguments of unknown shape}

Denote with an abstract relation

\[ \text{id} : \{x: \alpha\} \rightarrow \{y: \alpha \mid \text{Shape}(y) = \text{Shape}(x)\} \]

\[ \rho \]

\[ (\rho) \text{Id} : \{x: \alpha\} \rightarrow \{y: \alpha \mid \rho(y) = \rho(x)\} \]
Relational Parameters

\[ \beta \text{ list} \xrightarrow{\beta \text{ tree}} \]
\[ \text{id} : \alpha \rightarrow \alpha \]

\text{id} can take arguments of \textbf{unknown shape}

Denote with an abstract relation

\[ \text{id} : \{ x : \alpha \} \rightarrow \{ y : \alpha \mid \text{Shape}(y) = \text{Shape}(x) \} \]

\[ \rho \]

\[ (\rho) \text{Id} : \{ x : \alpha \} \rightarrow \{ y : \alpha \mid \rho(y) = \rho(x) \} \]

Shape of the argument is also the shape of its result

\textbf{Relationally parametric type of \text{id}}
A Parametric Type of \texttt{pairMap} ...

... by focusing on possible shape invariance between $\alpha$ and $\beta$

$$(\rho_\alpha, \rho_\beta) \ \texttt{pairMap} : \{x_1 : \alpha\} \ast \{x_2 : \alpha\}
\quad \rightarrow (\{x : \alpha\} \rightarrow \{y : \beta \mid \rho_\beta(y) = \rho_\alpha(x)\})
\quad \rightarrow \{y_1 : \beta \mid \rho_\beta(y_1) = \rho_\alpha(x_1)\}
\quad \ast \{y_2 : \beta \mid \rho_\beta(y_2) = \rho_\alpha(x_2)\}$$
A Parametric Type of `pairMap` ...  

... by focusing on possible shape invariance between $\alpha$ and $\beta$

\[ (\rho_\alpha, \rho_\beta) \]

\[ \text{pairMap} : \{x_1 : \alpha\} \times \{x_2 : \alpha\} \]

\[ \rightarrow \left( \left\{ y : \beta \mid \rho_\beta(y) = \rho_\alpha(x) \right\} \right) \]

\[ \rightarrow \left\{ y_1 : \beta \mid \rho_\beta(y_1) = \rho_\alpha(x_1) \right\} \]

\[ \ast \left\{ y_2 : \beta \mid \rho_\beta(y_2) = \rho_\alpha(x_2) \right\} \]

\[ \text{denote shapes of } \alpha \text{ and } \beta, \text{ respectively} \]
A Parametric Type of pairMap ...

... by focusing on possible shape invariance between $\alpha$ and $\beta$

\[(\rho_\alpha, \rho_\beta) \text{ pairMap : } \{x_1: \alpha\} \ast \{x_2: \alpha\} \rightarrow (\{x: \alpha\} \rightarrow \{y: \beta \mid \rho_\beta(y) = \rho_\alpha(x)\}) \rightarrow \{y_1: \beta \mid \rho_\beta(y_1) = \rho_\alpha(x_1)\} \ast \{y_2: \beta \mid \rho_\beta(y_2) = \rho_\alpha(x_2)\}\]
A Parametric Type of `pairMap` ...

... by focusing on possible shape invariance between $\alpha$ and $\beta$

$(\rho_\alpha, \rho_\beta)$

`pairMap : \{x_1: \alpha\}*\{x_2: \alpha\} → (\{x: \alpha\} → \{y: \beta \mid \rho_\beta(y) = \rho_\alpha(x)\})`

 EXPRESS

gets propagated to result type
A Parametric Type of `pairMap` ...

\[(\rho_\alpha, \rho_\beta) \text{ pairMap} : \{x_1: \alpha\} \ast \{x_2: \alpha\} \rightarrow (\{x: \alpha\} \rightarrow \{y: \beta \mid \rho_\beta(y) = \rho_\alpha(x)\}) \rightarrow \{y_1: \beta \mid \rho_\beta(y_1) = \rho_\alpha(x_1)\} \ast \{y_2: \beta \mid \rho_\beta(y_2) = \rho_\alpha(x_2)\}\]

For eg:

\((l_1, l_2) = \text{pairMap (}R_{i_0}, R_{f_0}) (t_1, t_2) \text{ inOrder} \langle \alpha \text{ lists}, \alpha \text{ trees} \rangle)\)
A Parametric Type of `pairMap` ...

\[(\rho_\alpha, \rho_\beta)\] `pairMap` : \(\{x_1 : \alpha\} \times \{x_2 : \alpha\}\)

\[\rightarrow (\{x : \alpha\} \rightarrow \{y : \beta \mid \rho_\beta(y) = \rho_\alpha(x)\})\]

\[\rightarrow \{y_1 : \beta \mid \rho_\beta(y_1) = \rho_\alpha(x_1)\}\]
\[\times \{y_2 : \beta \mid \rho_\beta(y_2) = \rho_\alpha(x_2)\}\]

For eg:

\[(l_1, l_2) = \text{pairMap } (R_{i0}, R_{f0}) (t_1, t_2) \text{ inOrder}\]

\(\alpha\) lists

\(\alpha\) trees

explicit instantiation of relational parameters
```
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])
```
treefoldl

\[
\text{treefoldl } f \ i \ (\text{Node } n) = f \ i \ n \\
| f \ i \ (\text{Tree left node right}) = \\
\quad \text{treefoldl } f \ (f \ (\text{treefoldl } f \ i \ \text{left}) \ \text{node}) \ \text{right}
\]

\[
\text{val inOrder } = \text{fn } t \Rightarrow \text{treefoldl } t \ [ ] \\
\quad (\text{fn } acc \Rightarrow \text{fn } x \Rightarrow acc ++ [x])
\]
treefoldl

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

```
val inOrder t = f [x1]
```

```
Tree x4 (Node x2 x3) x5
```
treefoldl

\[
\text{treefoldl } f \ i \ (\text{Node } n) = f \ i \ n
\]
\[
\quad | f \ i \ (\text{Tree left node right}) = \\
\qquad \text{treefoldl } f \ (f \ (\text{treefoldl } f \ i \ \text{left}) \ \text{node}) \ \text{right}
\]

val inOrder = fn t => treefoldl t []

(fn acc => fn x => acc ++ [x])
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

\[
\text{treefoldl}\ f\ i\ (\text{Node}\ n) = f\ i\ n
\]
\[
|\ f\ i\ (\text{Tree}\ \text{left}\ \text{node}\ \text{right}) =
\]
\[
\text{treefoldl}\ f\ (f\ (\text{treefoldl}\ f\ i\ \text{left})\ \text{node})\ \text{right}
\]

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

\[
inOrder\ t = \ f\ [x_1,x_3,x_3,x_4]\ x_5
\]
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
A parametric type can be constructed to relate \( R_{\text{io}} \) on \( \alpha \) tree to some notion of order captured by an abstract relation \( \rho_\circ \) on \( \beta \).
A parametric type can be constructed to relate in-order \((R_{\text{io}})\) on \(\alpha\) tree to some notion of order captured by an abstract relation \((\rho_o)\) on \(\beta\).

\[(\rho_o)\ \text{treecfoldl} : \{t : \alpha\ \text{tree}\} \rightarrow \ldots \rightarrow \{v : \beta \mid \rho_o(v) = R_{\text{io}}(t)\}\]
A Parametric Type of `treefoldl`

\[(\rho_m, \rho_o) \text{ treefoldl}: \{t: \alpha \text{ tree}\} \rightarrow \{b: \beta \mid \rho_m(b) = \emptyset \\
\quad \land \rho_o(b) = \emptyset\}\]

\[\rightarrow (\{xs: \beta\} \rightarrow \{x: \alpha\} \rightarrow \]

\[\quad \{v: \beta \mid \rho_m(v) = \rho_m(xs) \cup \{x\} \\
\quad \land \rho_o(v) = \rho_m(xs) \times \{x\} \cup \rho_o(xs)\})\]

\[\rightarrow \{y: \beta \mid \rho_o(y) = R_{\text{vio}}(t) \land \rho_m(y) = R_{\text{tm}}(t) \}\]
A Parametric Type of \texttt{treefoldl}

\[(\rho_m, \rho_o) \texttt{treefoldl}: \{t:\alpha \text{ tree}\} \rightarrow \{b:\beta \mid \rho_m(b) = \emptyset \wedge \rho_o(b) = \emptyset\}\]

\[\rightarrow (\{xs:\beta\} \rightarrow \{x:\alpha\} \rightarrow \{v:\beta \mid \rho_m(v) = \rho_m(xs) \cup \{x\} \wedge \rho_o(v) = \rho_m(xs) \times \{x\} \cup \rho_o(xs)\})\]

\[\rightarrow \{y: \beta \mid \rho_o(y) = R_{\text{io}}(t) \wedge \rho_m(y) = R_{\text{tm}}(t)\}\]

Order invariant: relates in-order on the tree to a notion of order on \(\beta\)
(\(\rho_m, \rho_o\) treefoldl: \{t: \alpha \text{ tree}\} \rightarrow \{b: \beta \mid \rho_m(b) = \emptyset \land \rho_o(b) = \emptyset\})

\[\rightarrow (\{xs: \beta\} \rightarrow \{x: \alpha\} \rightarrow \{v: \beta \mid \rho_m(v) = \rho_m(xs) \cup \{x\} \land \rho_o(v) = \rho_m(xs) \times \{x\} \cup \rho_o(xs)\})\]

\[\rightarrow \{y: \beta \mid \rho_o(y) = \text{R}_{\text{vio}}(t) \land \rho_m(y) = \text{R}_{\text{tm}}(t)\}\]

Order invariant: relates in-order on the tree to a notion of order on \(\beta\)

Membership invariant: relates membership of the tree to a notion of membership of \(\beta\)
A Parametric Type of `treefoldl`

\[(\rho_m,\rho_o)\] `treefoldl`: \{\(t:\alpha\) tree\} \rightarrow \{\(b:\beta\) \mid \rho_m(b)=\emptyset \wedge \rho_o(b)=\emptyset\} \\
\rightarrow (\{xs:\beta\} \rightarrow \{x:\alpha\} \rightarrow \{v:\beta \mid \rho_m(v)=\rho_m(xs) \cup \{x\} \wedge \rho_o(v)=\rho_m(xs) \times \{x\} \cup \rho_o(xs)\}) \\
\rightarrow \{y: \beta \mid \rho_o(y)=R_{io}(t) \wedge \rho_m(y)=R_{tm}(t)\}

Order invariant: relates in-order on the tree to a notion of order on \(\beta\)

Membership invariant: relates membership of the tree to a notion of membership of \(\beta\)
inOrder using treefoldl

```plaintext
val inOrder = fn t => treefoldl (Rlm,Rfo) t []
  (fn acc => fn x => acc ++ [x])
```
inOrder using treefoldl

val inOrder = fn t => treefoldl (Rlm, Rfo) t []
(fn acc => fn x => acc ++ [x])
inOrder using treefoldl

val inOrder = fn t =>
  treefoldl ((Rlm, Rfo) => t []
               (fn acc => fn x => acc ++ [x]))
  (Rlm, Rfo)

Explicit relational parameter instantiation

\{t: \alpha \text{ tree}\} \rightarrow \ldots \rightarrow \{v: \alpha \text{ list} \mid R_{fo}(v) = R_{io}(t) \land R_{lm}(v) = R_{tm}(t)\}
Parametric Relations

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

\( \text{id} \) and \( \text{pairMap} \) are functions parameterized over relations

Relations can also be parameterized over relations
Parametric Relations

\( \text{id} \) and \( \text{pairMap} \) are functions parameterized over relations

Relations can also be parameterized over relations

For Eg:

\[
R_{\text{foo}}(l) = \{x\} \times R_{\text{lm}}(xs) \cup R_{\text{foo}}(xs)
\]
Parametric Relations

\text{id and pairMap} are functions parameterized over relations

Relations can also be parameterized over relations

\text{For Eg:}
\[ R_{fo}(l) = \{x\} \times R_{lm}(xs) \cup R_{fo}(xs) \]

Relates elements of \( l \)

fwd-order

\( l \)
id and pairMap are functions parameterized over relations.

Relations can also be parameterized over relations. For Eg:

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

Relates elements of \( l \)

Generalize

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

\text{id} \text{ and } \text{pairMap} \text{ are functions parameterized over relations}

Relations can also be parameterized over relations

For Eg:

\begin{align*}
R_{fo}(l) &= \{x\} \times R_{lm}(xs) \cup R_{fo}(xs) \\
\text{Relates elements of } l \\
R_{fo}[\rho](l) &= \rho(x) \times R_{lm}[\rho](xs) \cup R_{fo}[\rho](xs) \\
\text{Generalize} \\
\text{Relates different things for different instantiations of } \rho
\end{align*}
Parametric Relations

id and pairMap are functions parameterized over relations.

Relations can also be parameterized over relations.

For Eg:

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

Generalize

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

Relates different things for different instantiations of \( \rho \).

Note: If \( R_{id}(x) = \{x\} \) then \( R_{fo}[R_{id}](l) \) relates elements like non-parametric \( R_{fo}(l) \).
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:

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

By Definition:

\[ R_{\text{io}}[\rho](t) = \{ (\rho(x_i, y_i), \rho(x_j, y_j)) \mid i \leq j \} \]
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:
\[ R_{io}[R_{fst}](t) = \{ R_{fst}(x_i, y_i), R_{fst}(x_j, y_j) \mid i \leq j \} \]
\[ \Leftrightarrow R_{io}[R_{fst}](t) = \{ (x_i, x_j) \mid i \leq j \} \]
For Example ...

Let $R_{fst}$ be a relation on pairs, such that $R_{fst}(x, y) = \{x\}$

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:

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

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

in-order among first-components of pairs in $t$
For Example ...
For Example ...

treeMap : \(\alpha\) tree \(\rightarrow\) \((\alpha \rightarrow \beta) \rightarrow \beta\) tree
For Example ...

\[
\text{treeMap} : \alpha \text{ tree} \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \text{ tree}
\]

Relational type ... ... by focusing on possible shape invariance between \(\alpha\) and \(\beta\) (a la pairMap)

\[
(\rho_\alpha, \rho_\beta) \text{ treeMap} : \{t_1 : \alpha \text{ tree}\}
\rightarrow (\{x : \alpha\} \rightarrow \{y : \beta \mid \rho_\beta(y) = \rho_\alpha(x)\})
\]
For Example ...

\[
\text{treeMap} : \alpha \text{ tree} \to (\alpha \to \beta) \to \beta \text{ tree}
\]

Relational type ... ... by focusing on possible shape invariance between \(\alpha\) and \(\beta\) (a la \text{pairMap})

\[
(\rho_\alpha, \rho_\beta) \text{ treeMap} : \{t_1 : \alpha \text{ tree}\}
\to (\{x : \alpha\} \to \{y : \beta \mid \rho_\beta(y) = \rho_\alpha(x)\})
\]
For Example ...

\[
\text{treeMap} : \alpha \text{ tree} \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \text{ tree}
\]

Relational type ...

... by focusing on possible shape invariance between \(\alpha\) and \(\beta\) (a la pairMap)

\[
(\rho_\alpha, \rho_\beta) \text{ treeMap} : \{ t_1 : \alpha \text{ tree} \}
\rightarrow (\{ x : \alpha \} \rightarrow \{ y : \beta \mid \rho_\beta(y) = \rho_\alpha(x) \})
\rightarrow \{ t_2 : \beta \text{ tree} \mid ? \}
\]
For Example ...

\[
\text{treeMap} : \alpha \text{ tree} \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \text{ tree}
\]

Relational type ... ... by focusing on possible shape invariance between \( \alpha \) and \( \beta \) (a la pairMap)

\[
(\rho_\alpha, \rho_\beta) \text{ treeMap} : \{t_1: \alpha \text{ tree}\}
\rightarrow (\{x: \alpha\} \rightarrow \{y: \beta \mid \rho_\beta(y) = \rho_\alpha(x)\})
\rightarrow \{t_2: \beta \text{ tree} \mid ?\}
\]

\[R_{i0}(t_2) \neq R_{i0}(t_1)\]

\(R_{i0}(t_i)\) is a relation on elements of \( t_i \)
and elements of \( t_1 \neq \) elements of \( t_2 \)
For Example...

\[
\text{treeMap} : \alpha \text{ tree} \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \text{ tree}
\]

Relational type... ...by focusing on possible shape invariance between \(\alpha\) and \(\beta\) (a la pairMap)

\[
(\rho_\alpha, \rho_\beta) \text{ treeMap} : \{t_1: \alpha \text{ tree}\}
\]

\[
\rightarrow (\{x: \alpha\} \rightarrow \{y: \beta \mid \rho_\beta(y) = \rho_\alpha(x)\})
\]

\[
\rightarrow \{t_2: \beta \text{ tree} \mid R_{\text{io}}[\rho_\beta](t_2) = R_{\text{io}}[\rho_\alpha](t_1)\}
\]

\[
\]
For Example ...

\[
\text{treeMap} : \alpha \text{ tree} \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \text{ tree}
\]

Relational type ...  
... by focusing on possible shape invariance between \(\alpha\) and \(\beta\) (a la \text{pairMap})

\[
(\rho_\alpha, \rho_\beta) \text{ treeMap} : \{t_1 : \alpha \text{ tree}\}
\rightarrow (\{x : \alpha\} \rightarrow \{y : \beta \mid \rho_\beta(y) = \rho_\alpha(x)\})
\rightarrow \{t_2 : \beta \text{ tree} \mid R_{io}[\rho_\beta](t_2) = R_{io}[\rho_\alpha](t_1)\}
\]

Parametric in-order relation \((R_{io}[\rho])\) is not necessarily a relation over elements.
For Example ...

\[(\rho_\alpha, \rho_\beta) \text{ treeMap} : \{t_1 : \alpha \text{ tree}\}
\rightarrow (\{x : \alpha\} \rightarrow \{y : \beta \mid \rho_\beta(y) = \rho_\alpha(x)\})
\rightarrow \{t_2 : \beta \text{ tree} \mid R_{i0}[\rho_\beta](t_2) = R_{i0}[\rho_\alpha](t_1)\}\]
(ρ_α, ρ_β) treeMap : {t_1: α tree} 
→ ({x: α} → {y: β | ρ_β(y) = ρ_α(x)}) 
→ {t_2: β tree | RiO[ρ_β](t_2) = RiO[ρ_α](t_1)}
For Example ...

treeMap \((R_{\text{fst}}, R_{\text{id}})\): \(\{t_1: \alpha \text{ tree}\}\)
\[
\rightarrow (\{x: \alpha\} \rightarrow \{y: \beta \mid R_{\text{id}}(y) = R_{\text{fst}}(x)\})
\rightarrow \{t_2: \beta \text{ tree} \mid R_{\text{io}}[R_{\text{id}}](t_2) = R_{\text{io}}[R_{\text{fst}}](t_1)\}
\]

Let \(R_{\text{id}}(x) = \{x\}\) be Identity relation

\(t_1\)

\(t_2\)
For Example ...

$$\text{treeMap} \ (R_{\text{fst}}, R_{\text{id}}): \ \{t_1: \alpha \ \text{tree}\}$$
$$\rightarrow (\{x: \alpha\} \rightarrow \{y: \beta \mid R_{\text{id}}(y) = R_{\text{fst}}(x)\})$$
$$\rightarrow \{t_2: \beta \ \text{tree} \mid R_{\text{io}}[R_{\text{id}}](t_2) = R_{\text{io}}[R_{\text{fst}}](t_1)\}$$

Let $$R_{\text{id}}(x) = \{x\}$$ be Identity relation

in-order among elements of $$t_2 = \text{in-order among first components of pairs in } t_1$$
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

\[
\begin{align*}
\Gamma \vdash \{ \nu : T \mid \phi_1 \} & \quad \Gamma \vdash \{ \nu : T \mid \phi_2 \} \\
[\Gamma_R] \models [\Gamma, \nu : T] \Rightarrow [\phi_1] \Rightarrow [\phi_2] \\
\Gamma \vdash \{ \nu : T \mid \phi_1 \} \ll \{ \nu : T \mid \phi_2 \}
\end{align*}
\]

i.e., if \( \phi \) 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)

⇒

Type checking is decidable
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**

<table>
<thead>
<tr>
<th>Uninterpreted sorts</th>
<th>( T_0, T_1, \ldots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted variables</td>
<td>( x : T_0, y : T_1, \ldots )</td>
</tr>
<tr>
<td>Sorted uninterpreted boolean functions (relations)</td>
<td>( R : T_0 \rightarrow \text{bool} \ldots )</td>
</tr>
<tr>
<td>Prenex quantification over sorted variables</td>
<td>( \forall (k : T_0). R(x, k) \iff x = k, )</td>
</tr>
<tr>
<td></td>
<td>( \exists (j : T_0). f(y) = j )</td>
</tr>
</tbody>
</table>
... is translation of artifacts of type refinement language into the EPR fragment of MSFOL.
Encoding ...

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

\[
\text{int, } \alpha, \alpha \text{ list} \rightarrow \text{translate} \rightarrow T_0, T_1, T_2, \ldots
\]
Encoding ...

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

\[
\begin{align*}
\text{int, } \alpha, \alpha \text{ list} & \quad \rightarrow \quad T_0, T_1, T_2, \ldots \\
x: \alpha, \ l: \alpha \text{ list} & \quad \rightarrow \quad x:T_1, \ l:T_2, \ldots 
\end{align*}
\]
... 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}

\begin{align*}
\text{translate} \quad & T_0, T_1, T_2, \ldots \\
& x:T_1, l:T_2, \ldots \\
& R_{fo}:T_2*T_1*T_1 \rightarrow \text{bool}, \\
& R_{lm}:T_2*T_1 \rightarrow \text{bool}
\end{align*}
... is translation of artifacts of type refinement language into the EPR fragment of MSFOL.

\[
\begin{align*}
\text{int, } \alpha, \alpha \text{ list} \\
x: \alpha, \ l: \alpha \text{ list} \\
R_{fo}, \\
R_{lm}
\end{align*}
\]

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

\[
\begin{align*}
T_0, T_1, T_2, \ldots \\
x: T_1, l: T_2, \ldots \\
R_{fo}: T_2 \times T_1 \times T_1 \rightarrow \text{bool}, \\
R_{lm}: T_2 \times T_1 \rightarrow \text{bool} \\
\forall (k, j: T_1). R_{fo}(l, k, j) \iff \\
(k=x) \land R_{lm}(xs, j)
\end{align*}
\]
Encoding ...

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

\[
\begin{align*}
\text{int}, \alpha, \alpha \text{ list} & \quad \text{T}_0, \text{T}_1, \text{T}_2, \ldots \\
x: \alpha, \ l: \alpha \text{ list} & \quad x: \text{T}_1, \ l: \text{T}_2, \ldots \\
R_{\text{fo}}, & \\
R_{\text{lm}} & \\
R_{\text{fo}}(l) = \{x\} \times R_{\text{lm}}(xs) & \\
\end{align*}
\]

\[
\begin{align*}
\forall (k, j: \text{T}_1). R_{\text{fo}}(l, k, j) & \iff \\
(k = x) & \land R_{\text{lm}}(xs, j)
\end{align*}
\]

but ...
Encoding ...

... parametric relations is not straightforward

Parametric Relations

\( R_{10}[R_{\text{fst}}], \ R_{fo}[R_{\text{id}}] \)

(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:

\[(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4), (x_5, y_5)\]
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:

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

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

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

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

We have already seen:

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:

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

\[ R_{\text{io}}[R_{\text{fst}}](t) = \{ (x_i, x_j) \mid i \leq j \} \]
A fully instantiated parametric relation can be defined in terms of its component non-parametric relations.

For eg:

We have already seen:

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

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

\[ R_{io}[R_{fst}](t) = \{(R_{fst}(a), R_{fst}(b)) \mid (a, b) \in R_{io}(t)\} \]
A fully instantiated parametric relation can be defined in terms of its component non-parametric relations.

For eg:

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

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

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

We have already seen:

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

\[ R_{i0}[R_{\text{fst}}], \]
\[ R_{fo}[R_{\text{id}}] \]

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 (e.g., 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.
Implemented as extended type checking pass in MLton Standard ML compiler
Implemented as extended type checking pass in MLton Standard ML compiler
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 → MLton Frontend → Core ML + spec

VC Gen

Relational VC
Implemented as extended type checking pass in MLton Standard ML compiler
Implemented as extended type checking pass in MLton Standard ML compiler

- SML Program + spec
  - MLton Frontend
  - Core ML + spec
    - VC Gen
      - VC Encode
      - Relational VC
    - Z3
  - VC Gen
- CATALYST
Validation

<table>
<thead>
<tr>
<th>Lists</th>
<th>Okasaki trees</th>
<th>Functional Graphs</th>
<th>MLton functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>rev</td>
<td>inOrder</td>
<td>folds</td>
<td>alpha-rename</td>
</tr>
<tr>
<td>concat</td>
<td>preOrder</td>
<td>traversals</td>
<td>substitutions</td>
</tr>
<tr>
<td>map</td>
<td>postOrder</td>
<td>maps</td>
<td>SSA</td>
</tr>
<tr>
<td>foldl</td>
<td>treefoldl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foldr</td>
<td>treefoldr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exists</td>
<td>balance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>filter</td>
<td>rotate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

.
### Validation

<table>
<thead>
<tr>
<th>Lists</th>
<th>Okasaki trees</th>
<th>Functional Graphs</th>
<th>MLton functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>rev</td>
<td>inOrder</td>
<td>folds</td>
<td>alpha-rename</td>
</tr>
<tr>
<td>concat</td>
<td>preOrder</td>
<td>traversals</td>
<td>substitutions</td>
</tr>
<tr>
<td>map</td>
<td>postOrder</td>
<td>maps</td>
<td>SSA</td>
</tr>
<tr>
<td>foldl</td>
<td>treefoldl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foldr</td>
<td>treefoldr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exists</td>
<td>balance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>filter</td>
<td>rotate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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