Relational Verification Beyond Shape Analysis

with Suresh Jagannathan

PURDUE
UNIVERSITY

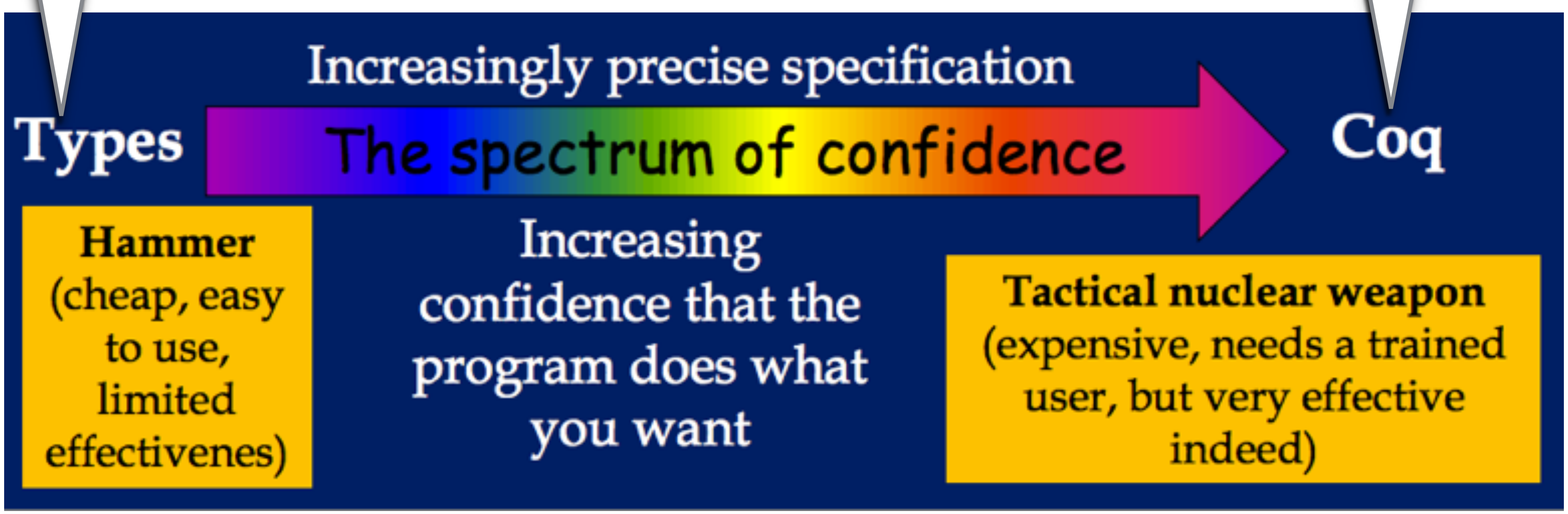
Simon PJ's Spectrum*

Hindley-Milner Types

```
 α list   α tree
f : α tree ⟶ α list
```

Dependent Types + HOL

```
Theorem.
  ∀(α:type).∀(t:α tree).
        len(f t) = size(t)
```



*Simon Peyton Jones, Fun with Type Functions, OPLSS'13

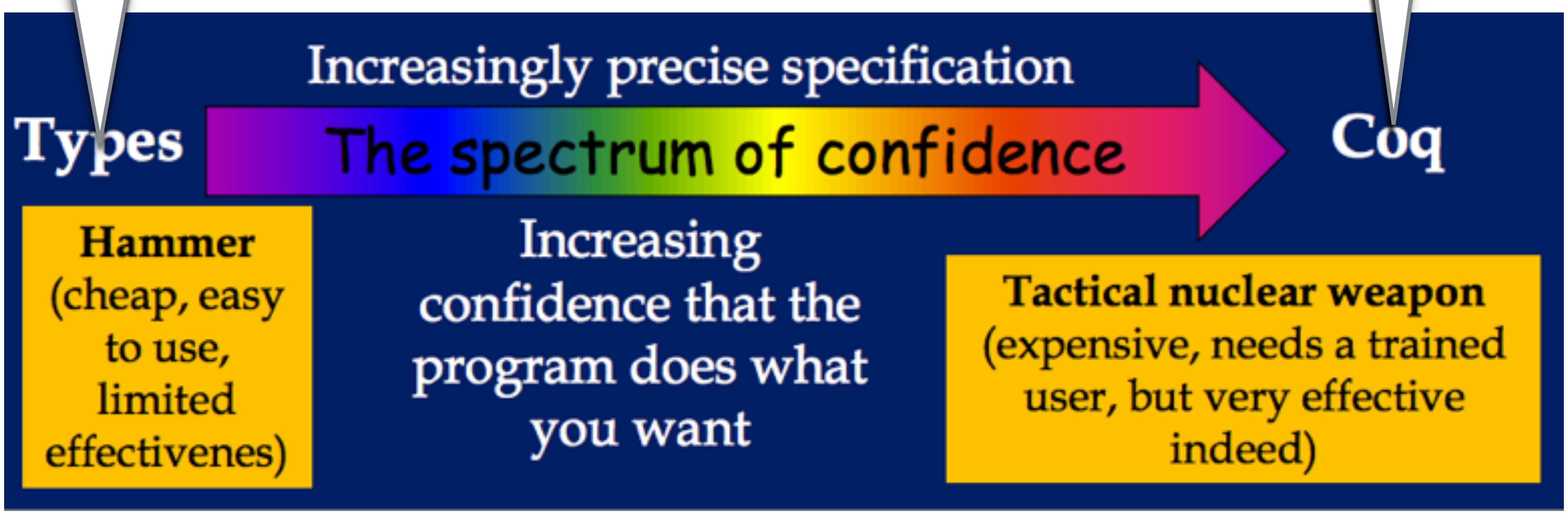
Simon PJ's Spectrum

Hindley-Milner Types

- − Basic Safety
- + Automated type checking
- + Full type inference

Dependent Types + HOL

- + Full Functional Verification
- − User-driven type checking
- − No type inference



Simon PJ's Spectrum

Statically ensuring strong semantic guarantees

??

- Effective verification
- Easy to use

Automatic Type Checking

Type Inference

How do we perform automated middleweight verification of functional programs?

SMT-enabled refinement types …

.. is a promising approach

eg: $\underbrace{\text{Liquid}}_{\text{“Logically Qualified” ML Types}}$ Types*

ML Type

`int`

$\downarrow$

$\{\nu\text{:int} \mid \underbrace{\varphi}\}$

Proposition in decidable logic (eg: linear arithmetic)

```
fun max x y = if y>x then y else x
```

$$\texttt{max} : x\text{:int} \longrightarrow y\text{:int} \longrightarrow \{\nu\text{:int} \mid \nu \geq x \wedge \nu \geq y\}$$

VC

$$y > x \vdash [y/\nu]\ (\nu \geq x \wedge \nu \geq y)$$

$$\neg(y > x) \vdash [x/\nu]\ (\nu \geq x \wedge \nu \geq y)$$

$\longrightarrow$ Z3

max typesafe

*P Rondon, M Kawaguchi, and R Jhala, Liquid Types, PLDI’08

But …

… a stereotypical ML/Haskell function is

- Symbolic (polymorphic), and
- Higher-order

`rev` `map` `concat` `foldl` `foldr` `traverse`

`treeBalance` `filter` `exists` `quicksort`

What are their "liquid" refinement types?

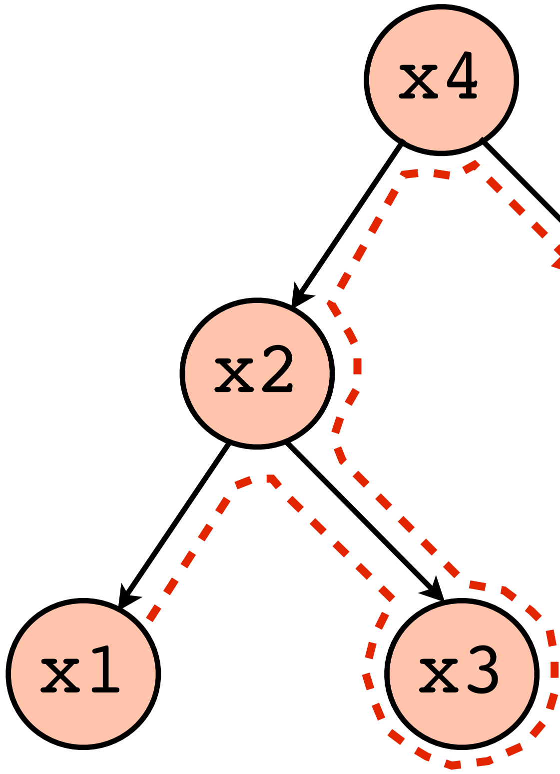
$$\texttt{traverse} : \{\texttt{t}:\alpha\ \texttt{tree}\} \longrightarrow \{\texttt{l}:\alpha\ \texttt{list} \mid \varphi\}$$

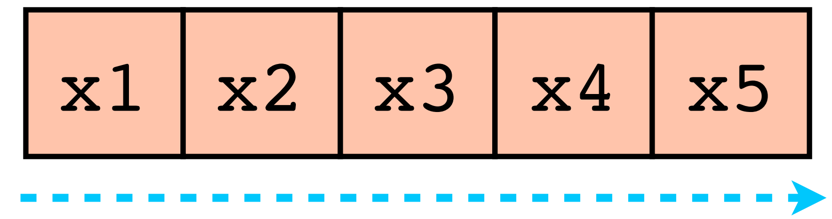
What goes here?

Overview

- CATALYST: A refinement type-based approach to automatically verify shape invariants of functional programs [ICFP'14].
- General relational verification with CATALYST
- Non-trivial Annotation burden as the cost of effective verification.
- Type inference as means to reduce the cost.

traverse





$$\texttt{traverse} : \underbrace{\alpha\ \texttt{tree}}_{\texttt{t}} \longrightarrow \underbrace{\alpha\ \texttt{list}}_{\texttt{l}}$$

$$\texttt{in-order(t)} \equiv \texttt{forward-order(l)}$$

Our Goal

We intend to capture rich relationships between shapes of data structures as type refinements

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

predicate over `Shape(l)` & `Shape(t)`

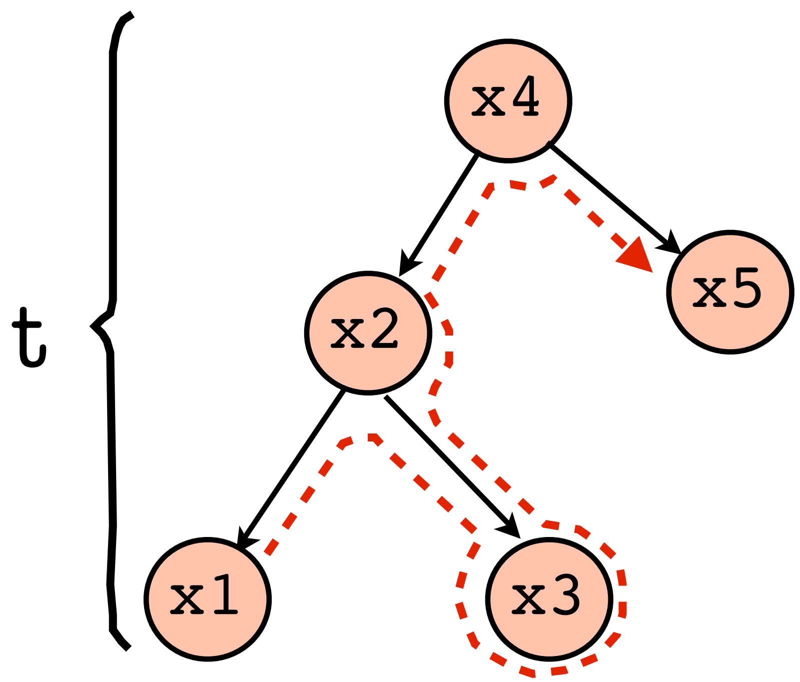
Key Observation

.. is that relations can express fine-grained shapes

```
forward-order        backward-order
pre-order            def-use
post-order           tree-membership
list-membership      in-order
SSA use-def
```

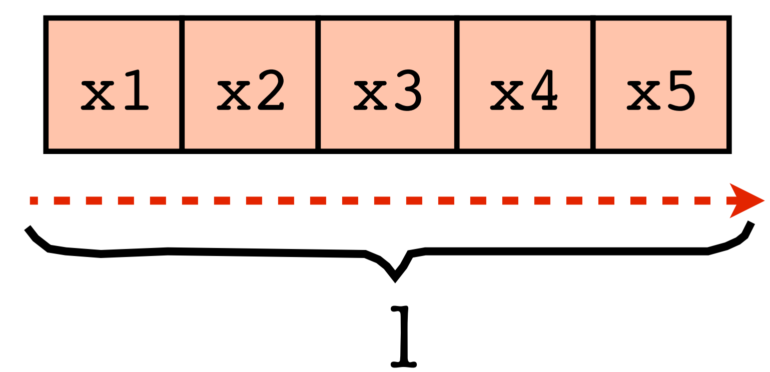
Relations!

For Example ...



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

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

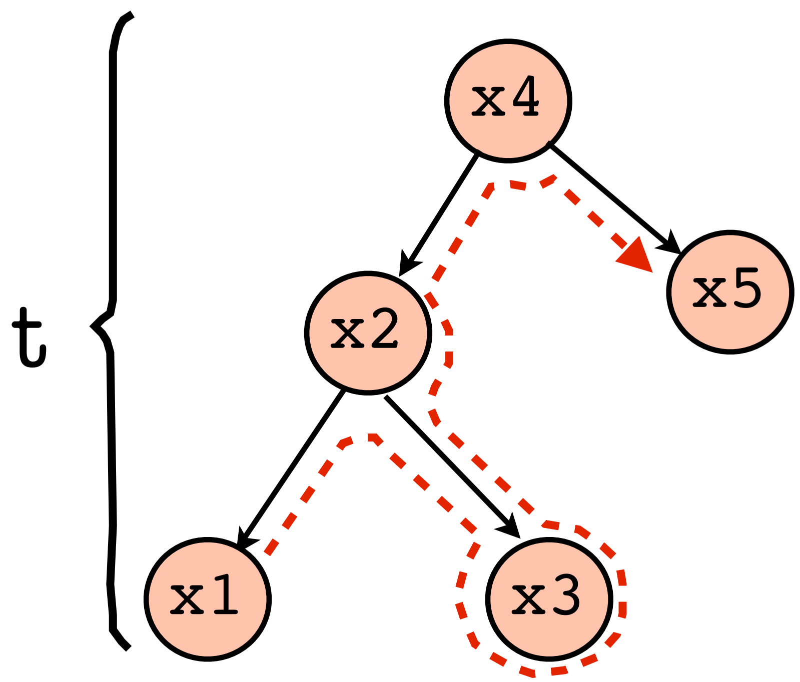


`fwd-order` of `l` is binary relation such that: `fwd-order`$(x_i,x_j) \Leftrightarrow i \leq j$

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

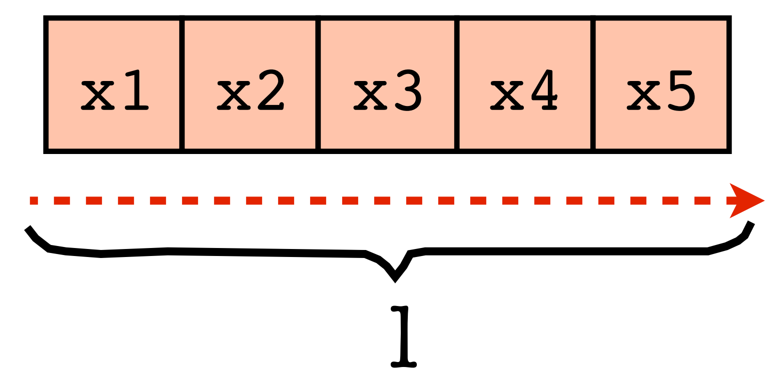
$R_{fo}(l) = R_{io}(t)$

For Example ...



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

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



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

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

traverse : {t:α tree} $\longrightarrow$ {l:α list | $R_{fo}(l) = R_{io}(t)$}

Relational type of traverse

The Language of Relations ...

... with relational operators ($\cup$, $\times$), is capable of expressing fine-grained shapes.

Predicates ($=$, $\subset$) over relations let us relate shapes

For Eg:

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

Relations can be constructed from a small set of primitives (R_{id}, R_{null}, R_{eq},..)

However ...

Parametricity

For higher-order & polymorphic functions, we need “parametric” constructs:

- Relationally Parametric Types
- Parametric Relations

Parametricity

For higher-order & polymorphic functions, we need "parametric" constructs:

- Relationally Parametric Types
- Parametric Relations

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

β list $\quad$ β tree

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

`id` is agnostic of shape of its argument

Shape of the argument is also the shape of its result

Relational Parameters

$$\texttt{id} : \overset{\beta\ \texttt{list}\quad \beta\ \texttt{tree}}{\alpha} \longrightarrow \alpha$$

`id` is agnostic of $\underbrace{\text{shape of its argument}}_{\text{Denote with an abstract relation}}$

Shape of the argument is also the shape of its result

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

Relationally parametric type of `id`

Parametricity

For higher-order & polymorphic functions, we need “parametric” constructs:

- Relationally Parametric Types
- Parametric Relations

Parametric Relations …

… let us parameterize relations over relations

For Eg:

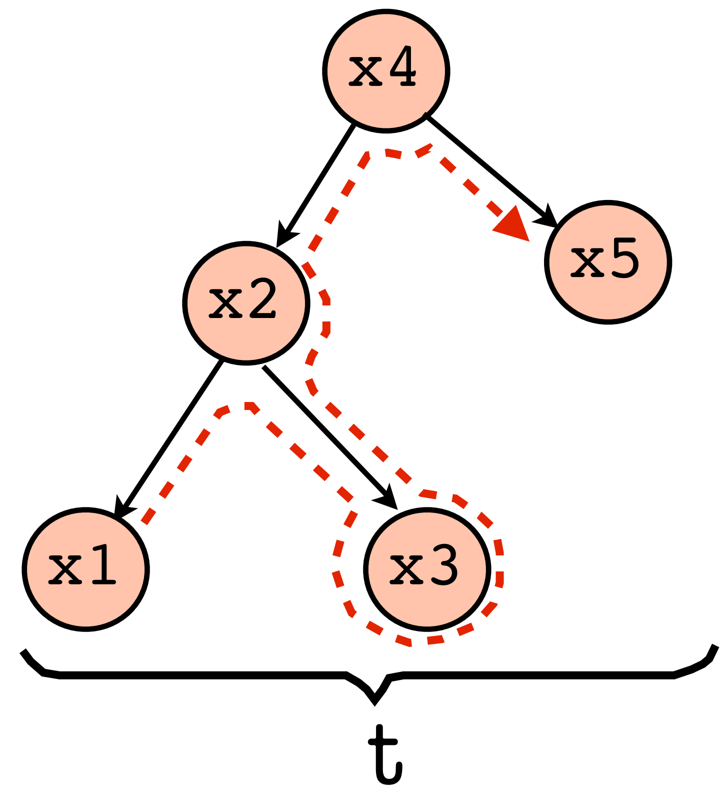
Relates elements of t

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

Generalize

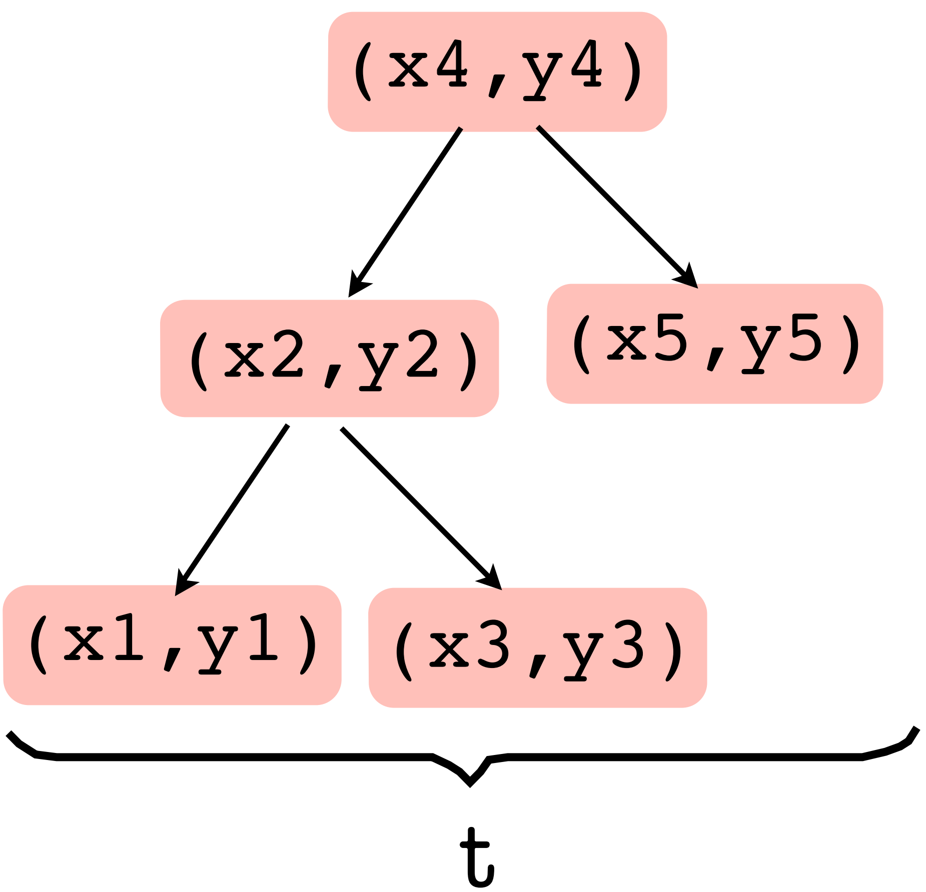
$$R_{io}[\rho](t) = \{(\rho(x_i), \rho(x_j)) \mid i \leq j\}$$

Relates substructures of elements of t as defined by ρ



Very useful to express fine-grained shapes!

For Example ...



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

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

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

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

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

`in-order` among first-components of pairs in t

In Summary ...

ML Type $\mathtt{A} \rightarrow \mathtt{B}$

(extended to)

Type refinements

$$(\rho)\ \mathtt{x}:\{\nu:\mathtt{A} \mid \varphi_{\mathtt{A}}\} \rightarrow \{\nu:\mathtt{B} \mid \varphi_{\mathtt{B}}\}$$

Relational parameters
on types

(are)

$$R_{\mathtt{B}}(\nu) = R_{\mathtt{A}}(\mathtt{x})$$

Simple inductive relations

(or)

$$R_{\mathtt{B}}[\rho](\nu) = R_{\mathtt{A}}[\rho](\mathtt{x})$$

Parametric inductive relations

In Summary …

$$\text{ML Type} \quad \mathtt{A} \rightarrow \mathtt{B}$$

(extended to)

$$(\rho)\ \mathtt{x}:\{\nu:\mathtt{A} \mid \varphi_{\mathtt{A}}\} \rightarrow \{\nu:\mathtt{B} \mid \underbrace{\varphi_{\mathtt{B}}}\}$$

(are)

$$R_{\mathtt{B}}(\nu) = R_{\mathtt{A}}(\mathtt{x})$$

(or)

$$R_{\mathtt{B}}[\rho](\nu) = R_{\mathtt{A}}[\rho](\mathtt{x})$$

Type checking is still decidable …

.. and practical!

We have automatically verified …

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		
⋮	⋮		

Observe …

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		
⋮	⋮		

They have nothing to do with shape/size of list/graph

Relational verification can go beyond verifying shape invariants

For example …

$$\texttt{quicksort} : \overbrace{\alpha\ \texttt{list}}^{l_1} \longrightarrow \overbrace{(\alpha \longrightarrow \alpha \longrightarrow \mathrm{bool})}^{f} \longrightarrow \overbrace{\alpha\ \texttt{list}}^{l_2}$$

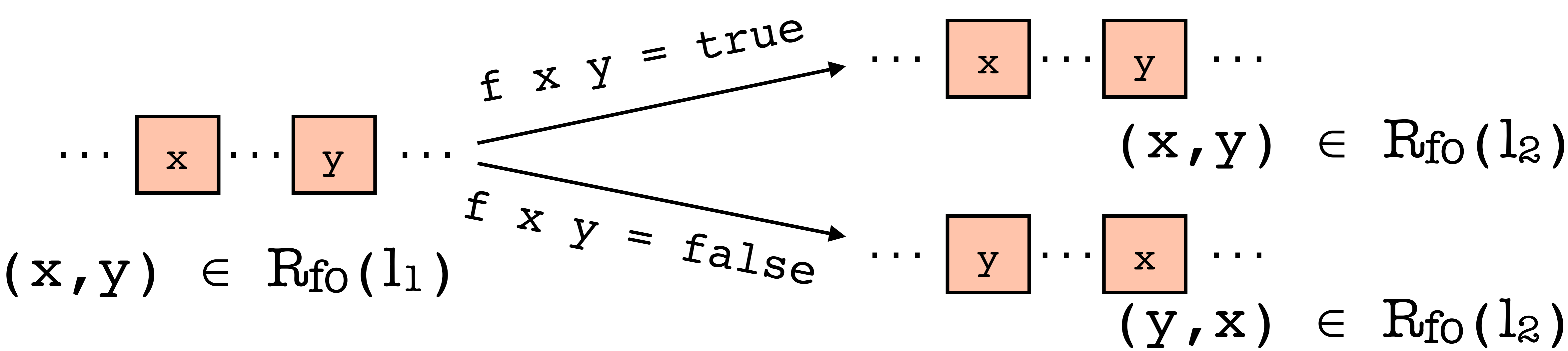
Can we write a spec for quicksort that should be valid regardless of how it is implemented?

Think in terms of order

$$\texttt{quicksort} : \overbrace{\alpha\ \texttt{list}}^{l_1} \longrightarrow \overbrace{(\alpha \longrightarrow \alpha \longrightarrow \text{bool})}^{f} \longrightarrow \overbrace{\alpha\ \texttt{list}}^{l_2}$$

a relation!

Result list is the input list sorted *as per the order* determined by the comparator function (f).



Thinking Relationally

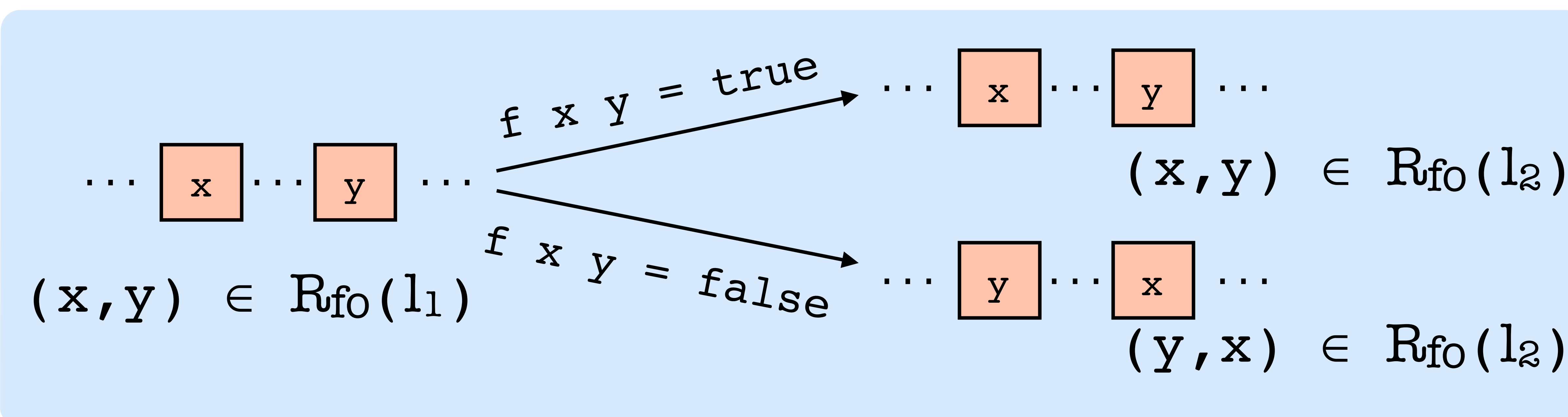
Let $\mathtt{R}$ be the relation denoting the order induced by the comparator function

$$(\mathtt{R})\ \mathtt{quicksort} : \alpha\ \mathtt{list} \longrightarrow \boxed{(\{x{:}\alpha\} \longrightarrow \{y : \alpha\} \longrightarrow \{v : \mathrm{bool} \mid v{=}\mathtt{true} \Leftrightarrow \mathtt{R}(x,y)\})} \longrightarrow \alpha\ \mathtt{list}$$

type of the comparator function

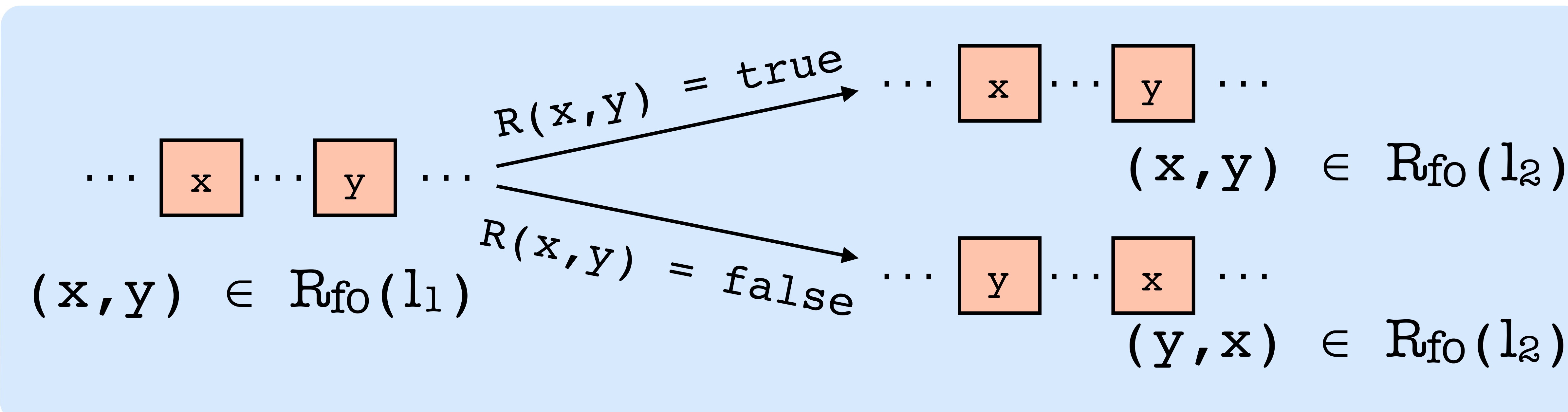
Thinking Relationally

$$(R)\ \texttt{quicksort} : \alpha\ \texttt{list} \longrightarrow (\{x{:}\alpha\} \longrightarrow \{y{:}\alpha\} \longrightarrow \{v : \texttt{bool} \mid v{=}\texttt{true} \Leftrightarrow R(x,y)\}) \longrightarrow \alpha\ \texttt{list}$$



Thinking Relationally

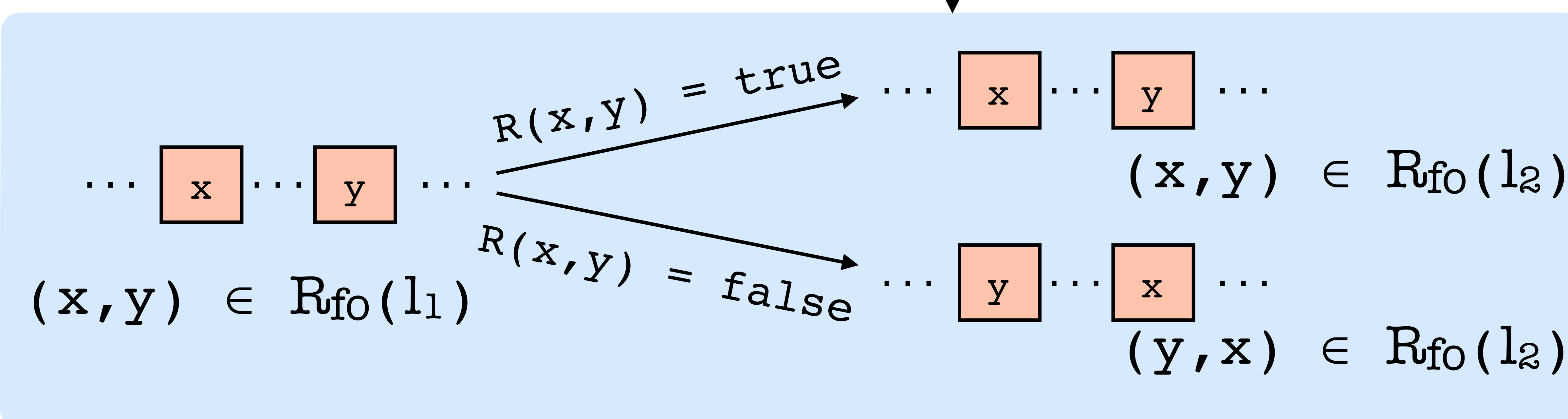
$$(\mathcal{R})\ \texttt{quicksort} : \alpha\ \texttt{list} \longrightarrow (\{x{:}\alpha\} \longrightarrow \{y{:}\alpha\} \longrightarrow \{v : \mathrm{bool} \mid v{=}\mathrm{true} \Leftrightarrow \mathcal{R}(x,y)\}) \longrightarrow \alpha\ \texttt{list}$$



Thinking Relationally

$$(R)\ \ \texttt{quicksort} : \{l_1 : \alpha\ \texttt{list}\} \longrightarrow (\{x : \alpha\} \longrightarrow \{y : \alpha\} \longrightarrow \{v : \text{bool} \mid v = \text{true} \Leftrightarrow R(x,y)\}) \longrightarrow \{l_2 : \alpha\ \texttt{list} \mid \varphi\}$$

(has to express)



So, the relational type is …

$$(R)\ \texttt{quicksort} : \{l_1 : \alpha\ \texttt{list}\} \longrightarrow (\{x:\alpha\} \longrightarrow \{y:\alpha\}$$
$$\longrightarrow \{v : \mathrm{bool} \mid v = \mathrm{true} \Leftrightarrow R(x,y)\})$$
$$\longrightarrow \{l_2 : \alpha\ \texttt{list} \mid \varphi\}$$

$$\Updownarrow$$

$$\forall (x,y) \in R_{fo}(l_1).\ R(x,y) \Rightarrow (x,y) \in R_{fo}(l_2)\ \wedge$$
$$\neg R(x,y) \Rightarrow (y,x) \in R_{fo}(l_2)$$

Full-functional correctness of quicksort can be verified automatically!

In reality …

.. this is what the actual type looks like:

```
(R) quicksort : {l : 'a list} -> ({x:'a} -> {y:'a}
              -> {z:bool | z = true <=> y ∈ R(x)})
              -> {v : 'a list | Rmem(v) = Rmem(l)
                     /\ Robs(v) = bind(Robs(l), λ(x,y).
                                  ({x} X ({y} - ({y} - R(x))))
                                   U  (({y} - R(x)) X {x}))
```

Fold Left

Here is a (relatively simple) relational type of fold_left:

```
('R1,'R2,'R3) fold_left : l -> b -> ({x} -> {acc}
    -> {z | 'R2(z) = 'R1(x) U 'R2(acc) /\
            'R3(z) = ('R1(x) X 'R2(acc)) U 'R3(acc)})
    -> {v | 'R2(v) = (Rmem 'R1)(l) U 'R2(b) /\
            'R3(v) = (Roas 'R1)(l) U 'R3(b) U
              ('R2(b) X (Rmem 'R1)(l))}
```

Types for First-Order functions …

…are also non-trivial

reverse:

```
(R) rev : {l : 'a list} -> {v : 'a list | Rmem[R](v) = Rmem[R](l) /\
                                          Robs[R](v) = Roas[R](l)}
```

concat:

```
(R) concat : {l1} -> {l2} -> {v | Rmem[R](v) = Rmem[R](l1) U Rmem[R](l2) /\
                                  Robs[R](v) = Robs[R](l1) U Robs[R](l2) U
                                               (Rmem[R](l1) X Rmem[R](l2))}
```

Pain points

- *Crafting* an appropriate relational type requires considerable effort.
- Annotation burden is non-trivial.

How do we relax this burden?

 Infer types

Type Inference*

- Conventional approach (constraint generation + solving) does not work.
  - Constraints are of form $\varphi_0 \Rightarrow \varphi_1$, where φ_0 and φ_1 are formulas in relational logic.
  - Unification is too severe. (eg: $\mathtt{x} \in \{\mathtt{1,2}\} \neq \mathtt{x} \in \{\mathtt{1}\}$ yet $\mathtt{x} \in \{\mathtt{1,2}\} \Rightarrow \mathtt{x} \in \{\mathtt{1}\}$).
  - Trial and error may not terminate in practice: too many (if not infinite) candidates for φ_0 and φ_1.

*Note: CATALYST already has local (within-function) type inference

Type Inference - Unconventional Approaches

- Using simple Under-Specifications to derive rich relational specifications.
    - Eg: $\forall(\mathrm{l}{:}\text{'a list}).\ \mathrm{rev}\ (\mathrm{rev}\ \ \mathrm{l}) = \mathrm{l})$

Easier to write than
$\mathrm{R}_{\mathrm{fo}}[\mathrm{R}](\mathrm{l}) = \mathrm{R}_{\mathrm{bo}}[\mathrm{R}](\mathrm{rev}\ \mathrm{l})$

- CEGAR and CEGIS: learning from errors for better trial and error.

- Using tests to "Learn Dependent Types" [Zhu, Jagannathan & Nori, MVD'14]

Conclusion

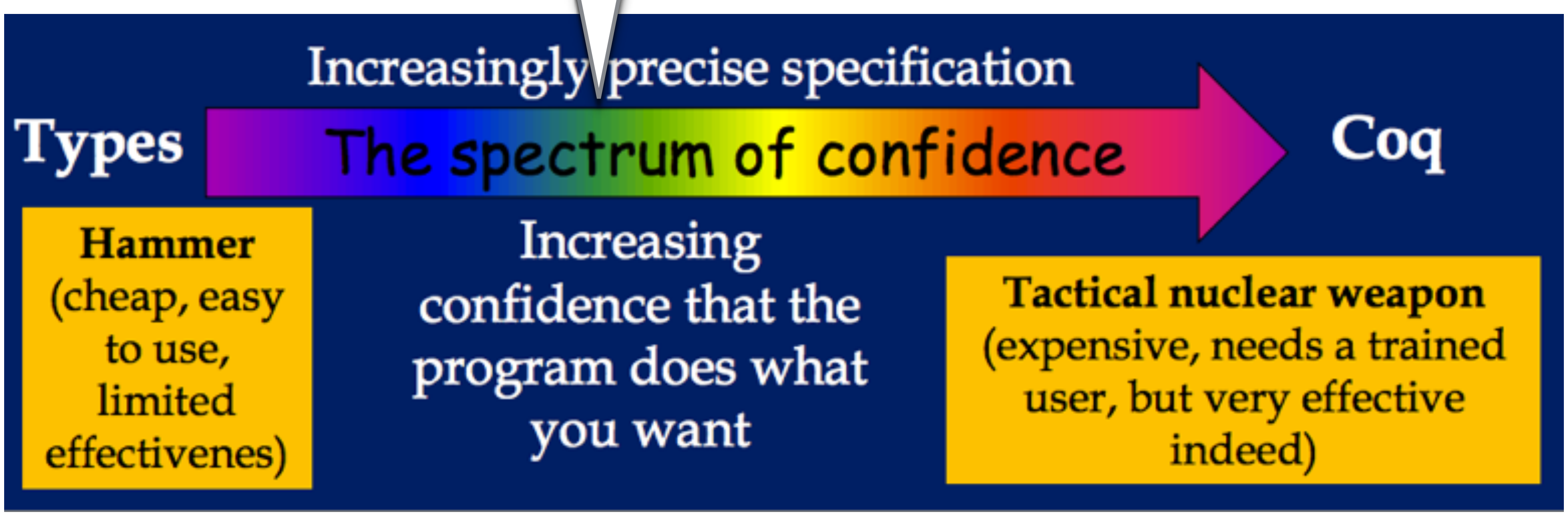
Statically ensuring strong semantic guarantees

??

- Effective verification
- Easy to use

Automatic Type Checking

Type Inference



Conclusion

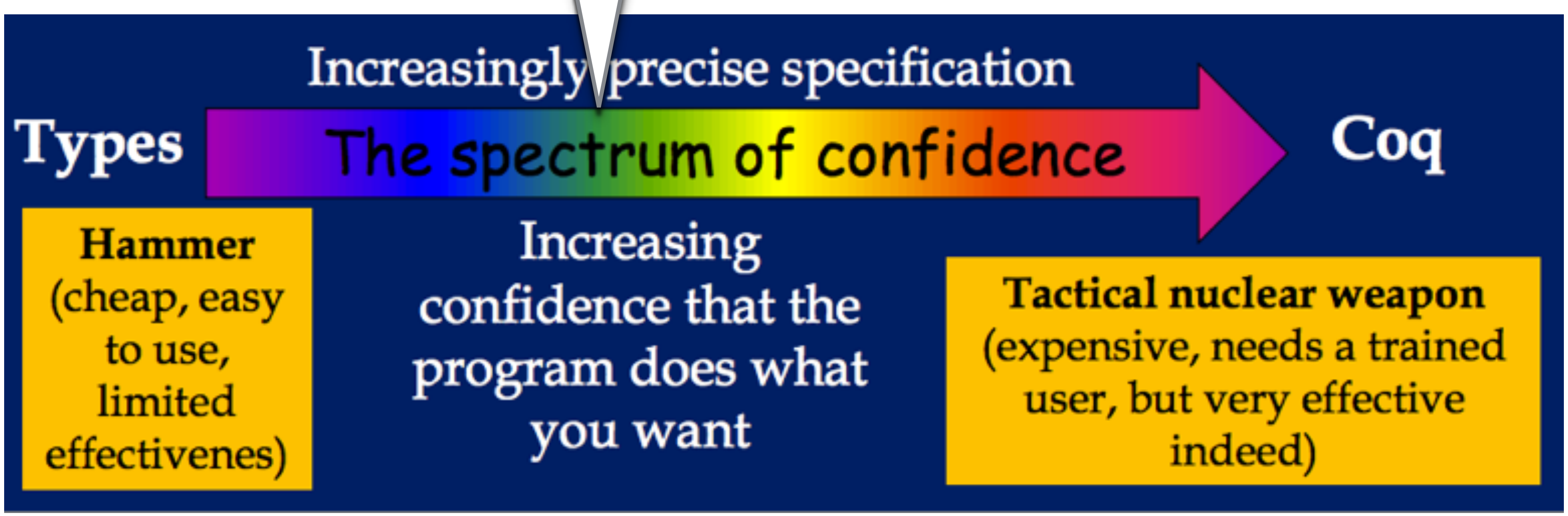
shape invariants and beyond!

CATALYST

- Effective verification
- Easy to use

✓ Automatic Type Checking

Type Inference



Thank you

http://tycon.github.io/catalyst/