Typed Clojure

INDIANA UNIVERSITY

Tool-assisted spec development

Ambrose Bonnaire-Sergeant
Sam Tobin-Hochstadt

Spec is awesome

- Expressive specification language
- Runtime instrumentation
- Generative testing
- Documentation
- Parsing

Show of hands:

Who's avoided writing a spec because…?

- Too inexperienced with spec
- Reverse engineering too costly to justify
- Partially wrote spec, but will "fill it in later"

Show of hands:

Who's updated a spec because…?

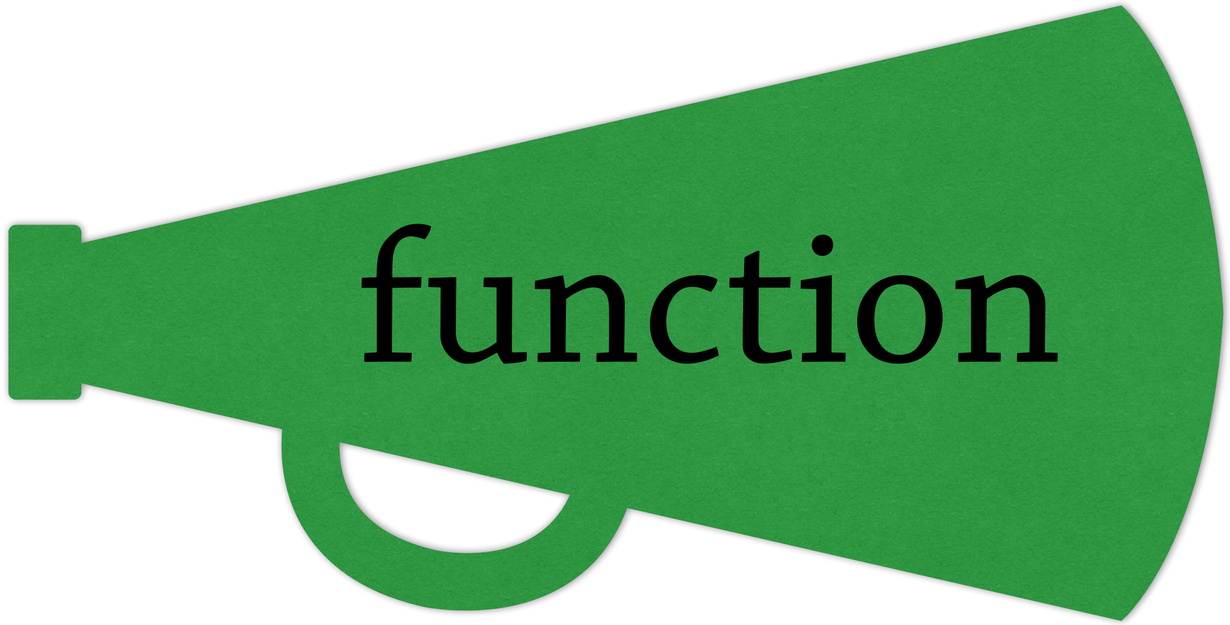
- Typo in the spec
- Function/data out of date with spec

We need help writing our specs

Idea

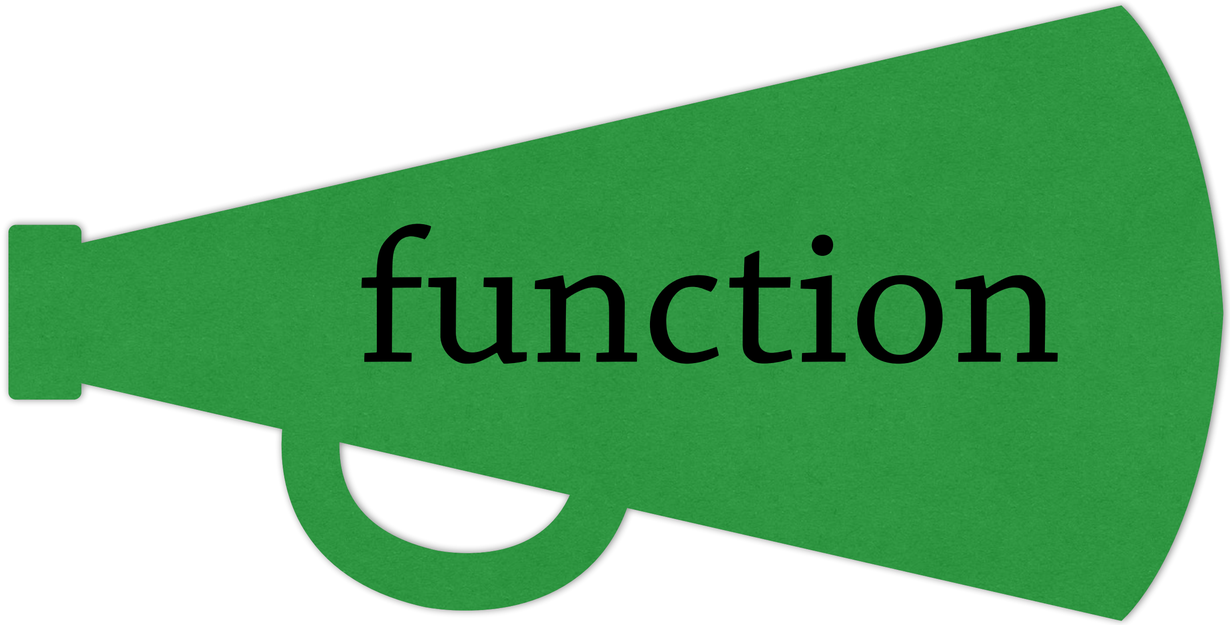
- Spec models how our program currently operates
  - So… **run the program**, and **observe!**
- Use observations to generate specs

Analogy:
Megaphone as a function

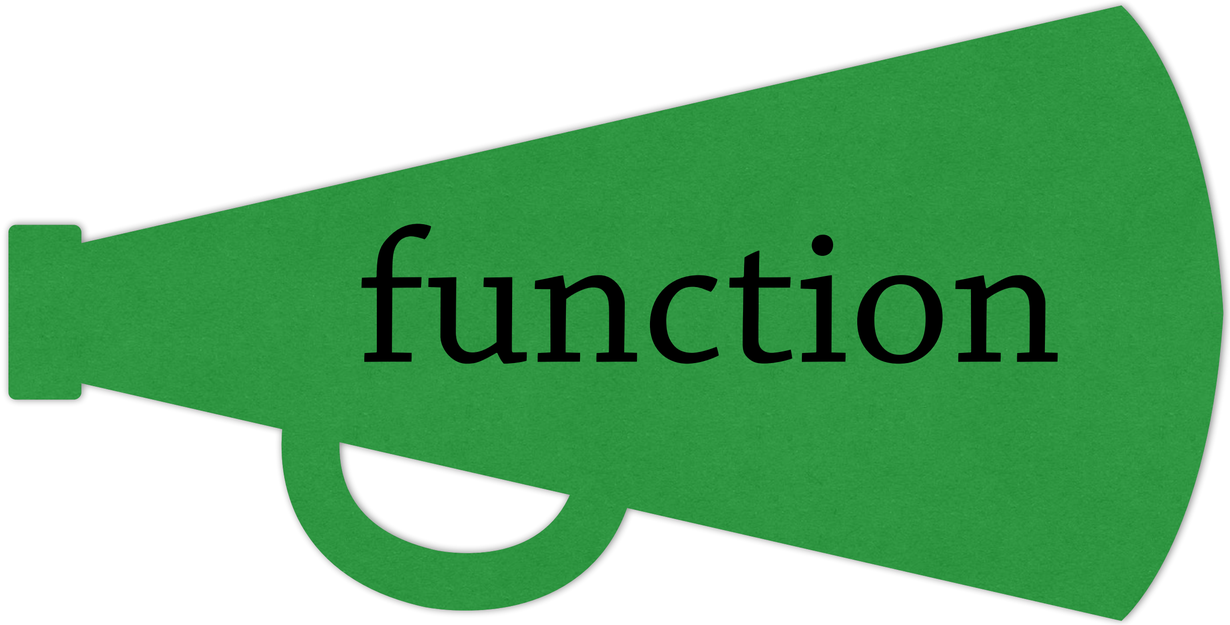


Goal:

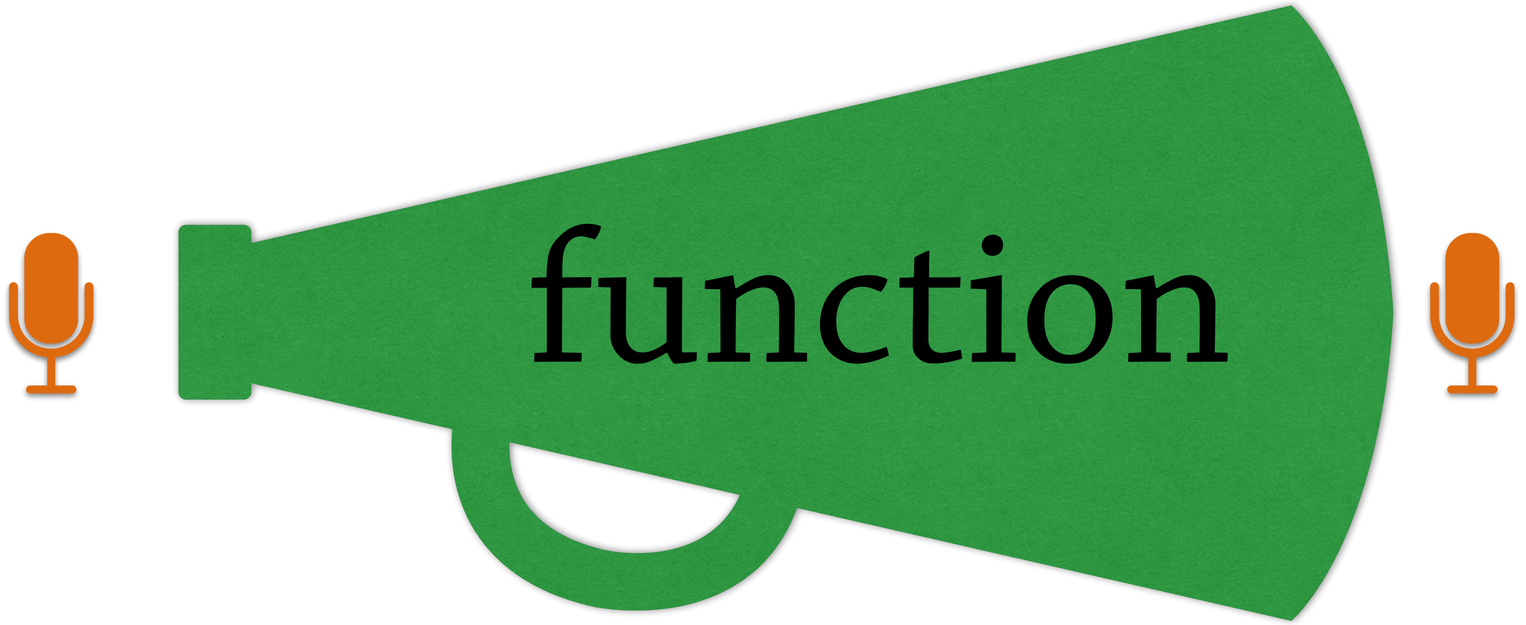
What is the “spec” for a megaphone?



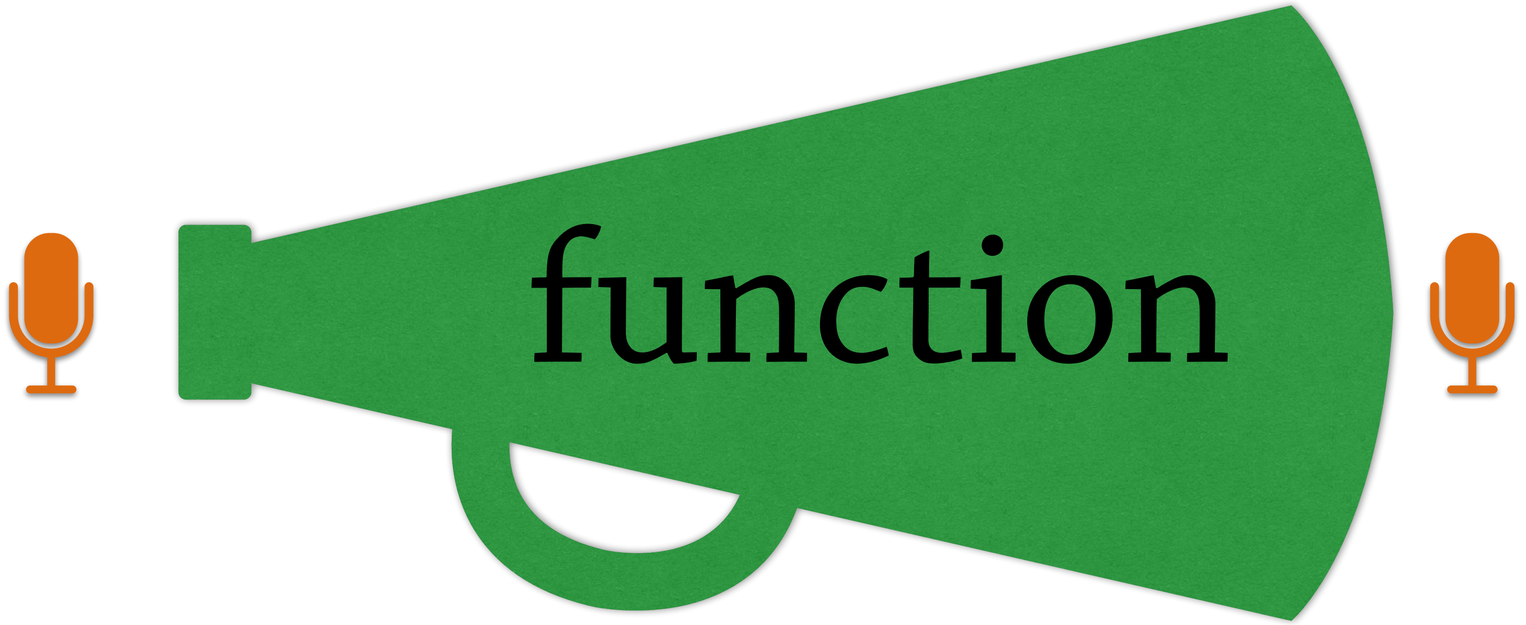
Step 1:
Instrumentation



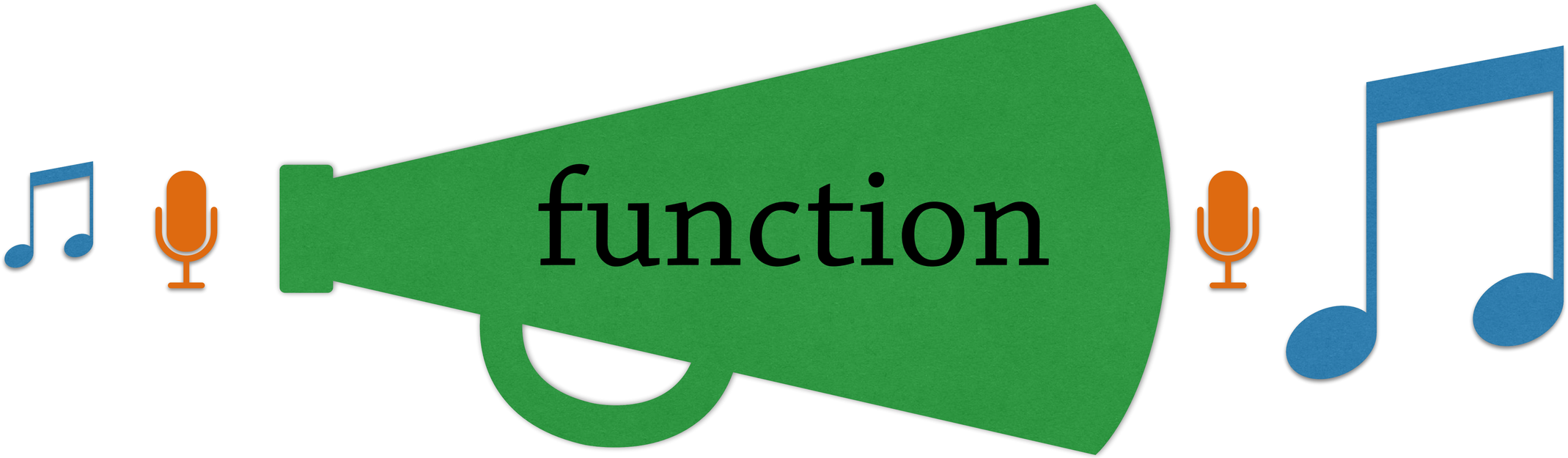
Step 1:
Instrumentation



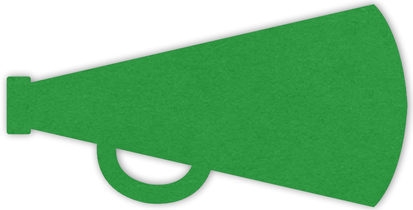
Step 2:
Observe running program



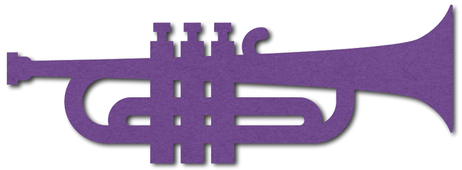
Step 2:
Observe running program



Step 3:
Summarize execution

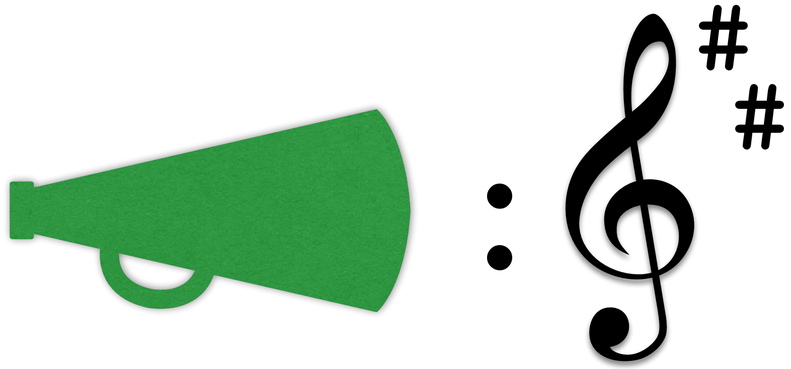


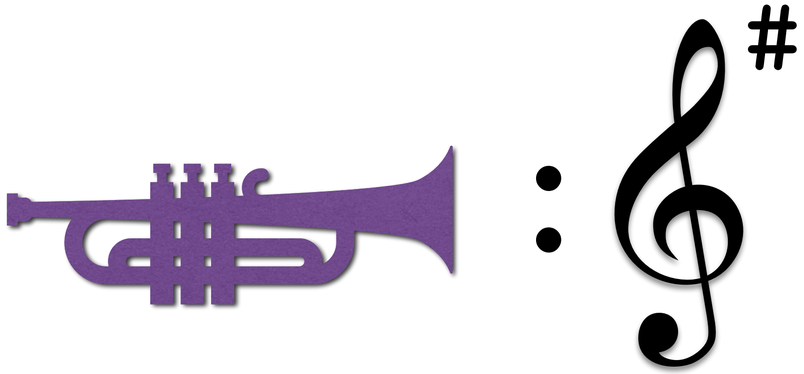
:



:

Step 3:
Summarize execution





The challenge:
Deriving “good” specs from program execution

Values

```
:foo
```

The observed value

Specs

1 `any?` 2 `keyword?` 3 `#{:foo}`

Our choices to spec value

Values

```
:foo
```

Specs

Don't care

```
1 any?
```

```
2 keyword?
```

Homogeneous

```
3 #{:foo}
```

Enumeration

Values

```
{1 2, 3 4}
```

Specs

```
1 map?
2(map-of int? int?)
3(map-of #{1 3} #{2 4})
4(map-of pos? pos?)
```

Instance check

Homogeneous

Enumerate

Constrain

Values

```
{:x 2, :y 3}
```

Specs

```
1 (map-of keyword? int?)
```

Homogeneous

```
2 (def ::x int?)
  (def ::y int?)
  (s/keys :req-un [::x ::y])
```

Entity map

```
{:shape :square
 :width 3}
```

Homogeneous

Entity map

Tagged
Entity map

Values

```
1 (map-of keyword? (or int? keyword?))

2 (def ::shape keyword?)
  (def ::width int?)
  (def ::height int?)
  (s/keys :req-un [::shape ::width])

3 (defmulti shape :shape)
  (defmethod shape [_] :square
    (s/keys :req-un [::shape ::width]))

  (multi-spec shape :shape)
```

Specs

```
{:shape :square
 :width 3}

{:shape :rect
 :width 3
 :height 4}
```

Values

```
1 (map-of keyword? (or int? keyword?))

2 (def ::shape keywor
  (def ::width int?)
  (def ::height int?)
  (s/keys :req-un [::shape ::width]
          :opt-un [::height])

3 (defmulti shape :shape)
  (defmethod shape [_] :square
    (s/keys :req-un [::shape ::width]))
  (defmethod shape [_] :rect
    (s/keys :req-un [::shape ::width
                     ::height]))

  (multi-spec shape :shape)
```

Optional entries

Second tag

Specs

Values

```
{:shape :square, :width 3
 :inner {:shape :rect, :width 1, :height 3,
         :inner {:shape :square, :width 3}}}
```

1 `map?`

2 `(map-of …)`

3 `(keys …)`

4 `(multi-spec …)`

Tagged Entity map

5…

```
(defmulti shape :shape)
(defmethod shape [_
  (s/keys :req-un
          :opt-un
(defmethod shape [_]
  (s/keys :req-un [::shape ::width ::height]
          :opt-un [::inner]))
(s/def ::inner (multi-spec shape :shape))
(multi-spec shape :shape)
```

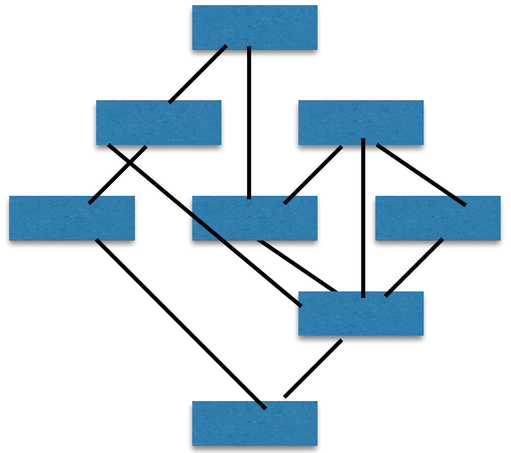
Recursively defined, tagged entity map

Phew!

Many implicit decisions

How do we make sense of all this?

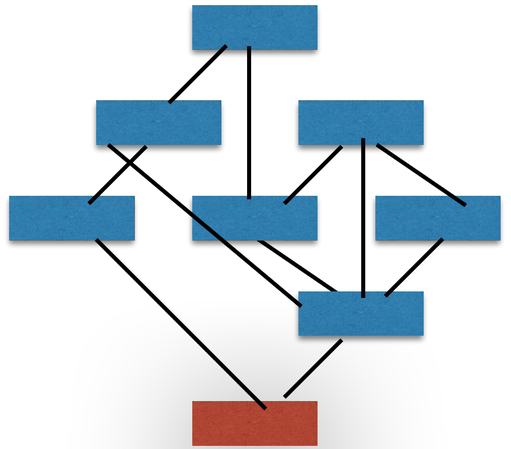
1 View possible specs as a **lattice**



2 Observing different values **moves** along lattice

How do we make sense of all this?

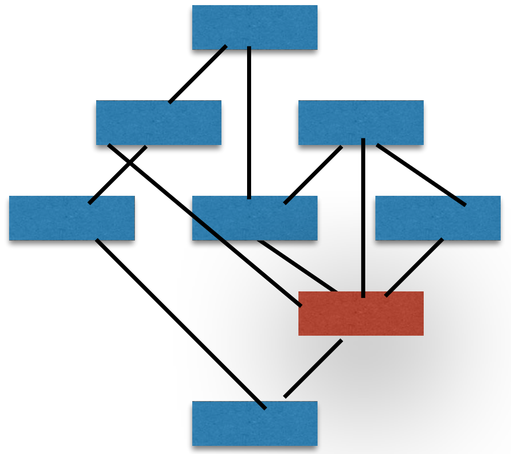
1 View possible specs as a **lattice**



2 Observing different values **moves** along lattice

How do we make sense of all this?

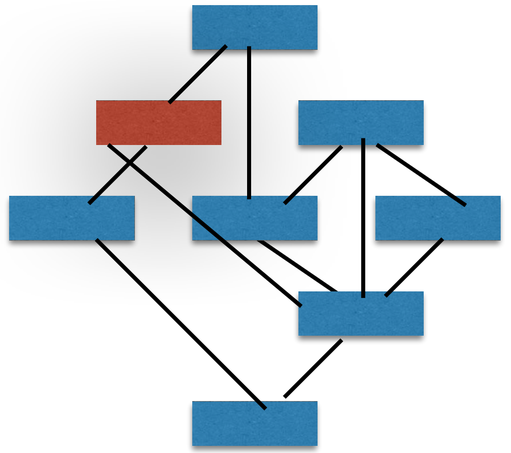
1 View possible specs as a **lattice**



2 Observing different values **moves** along lattice

How do we make sense of all this?

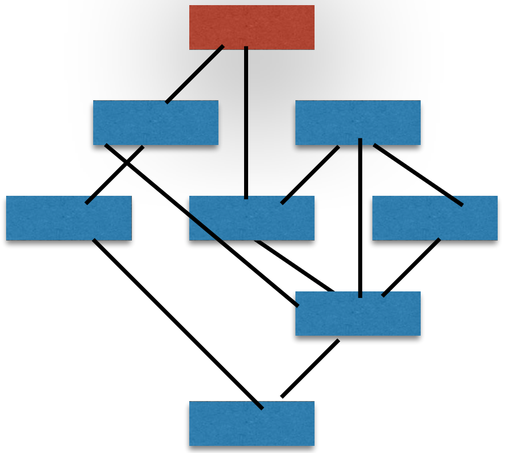
1 View possible specs as a **lattice**



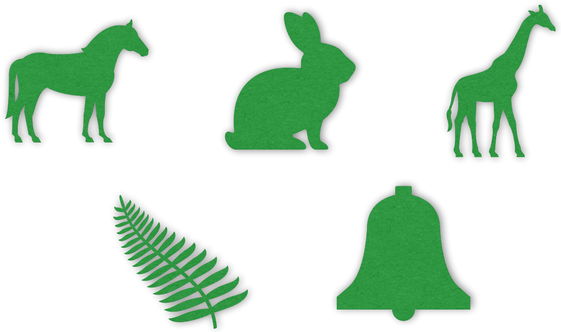
2 Observing different values **moves** along lattice

How do we make sense of all this?

1 View possible specs as a **lattice**



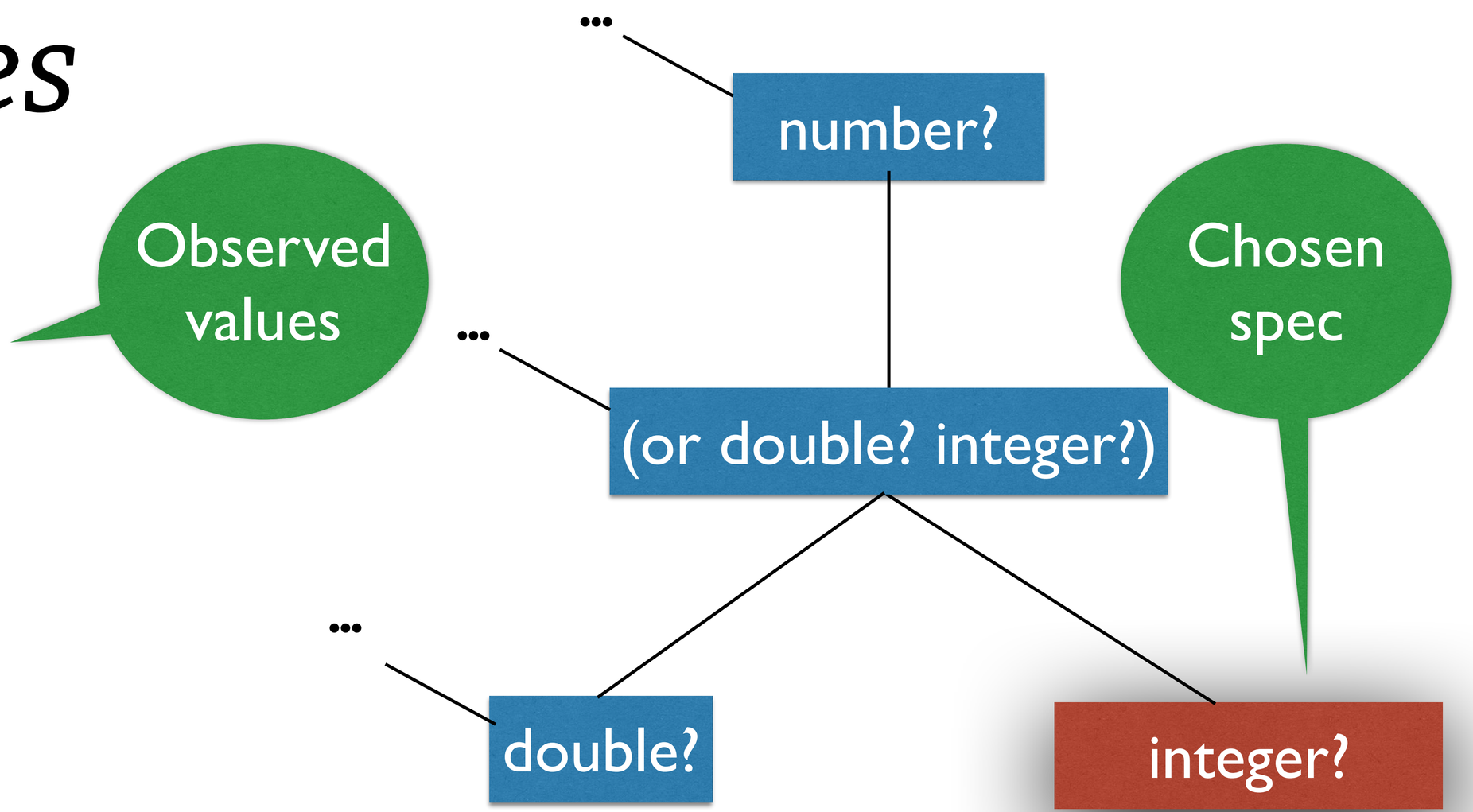
2 Observing different values **moves** along lattice



Build lattice using heuristics about Clojure idioms

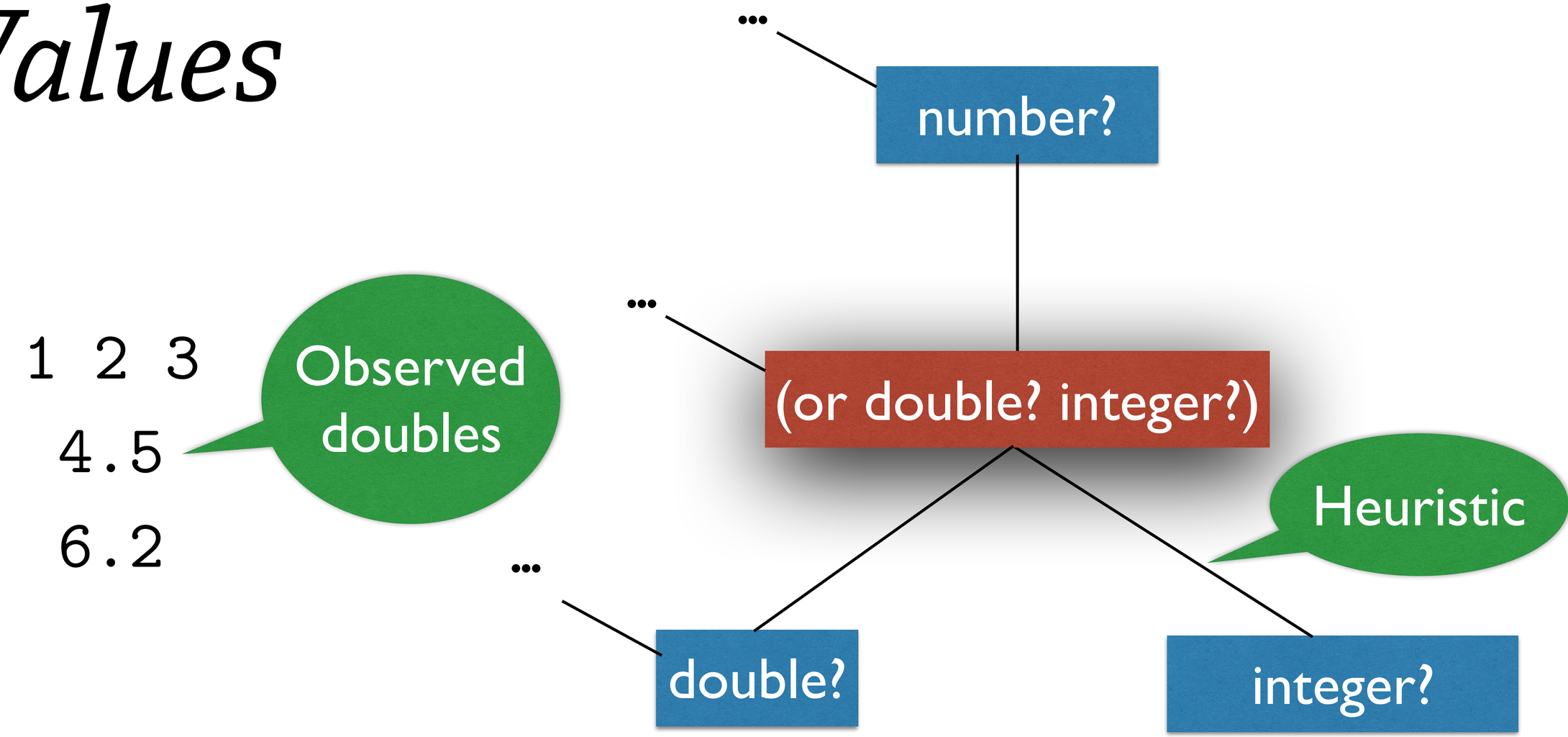
Observed Values

Chosen Spec



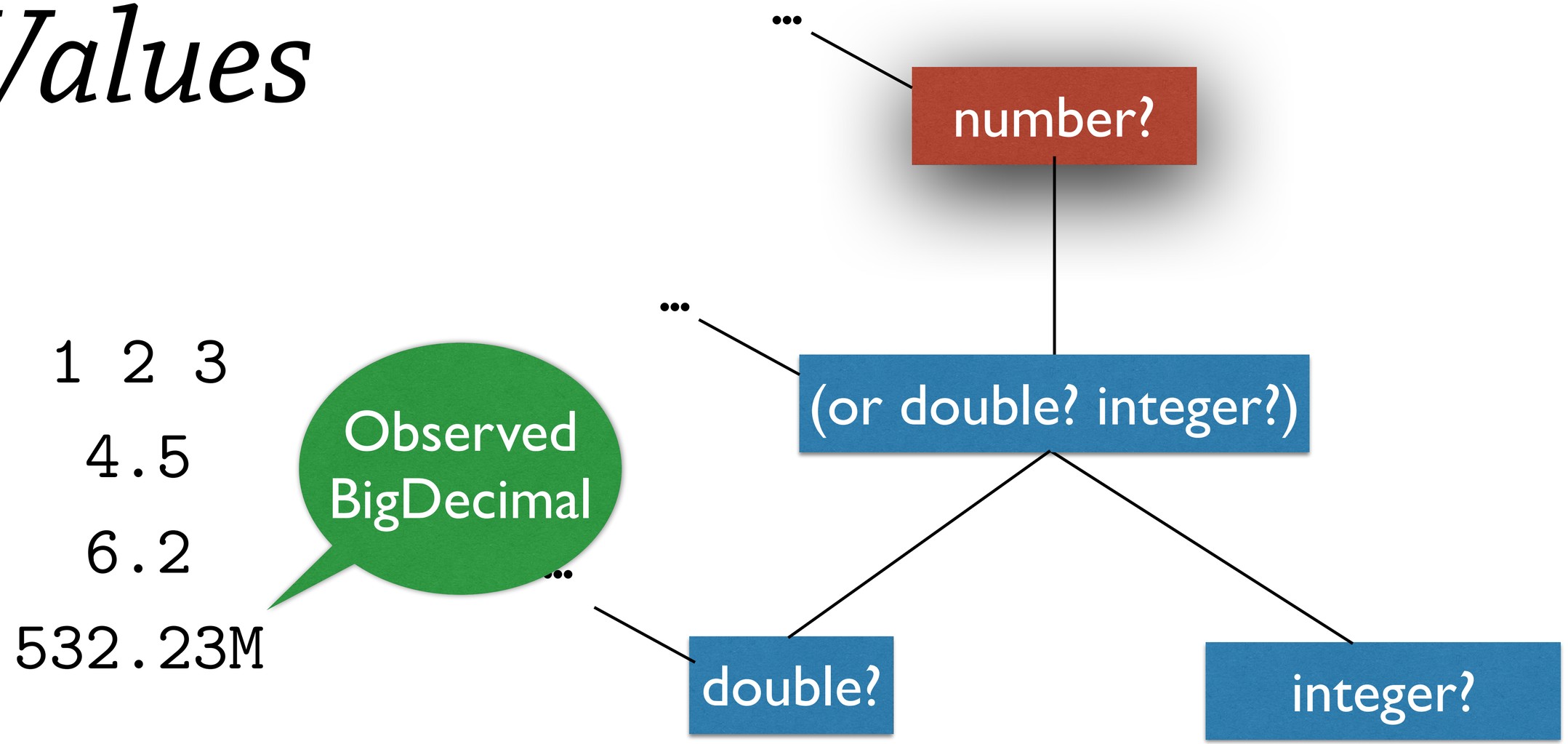
Observed Values

Chosen Spec



Observed Values

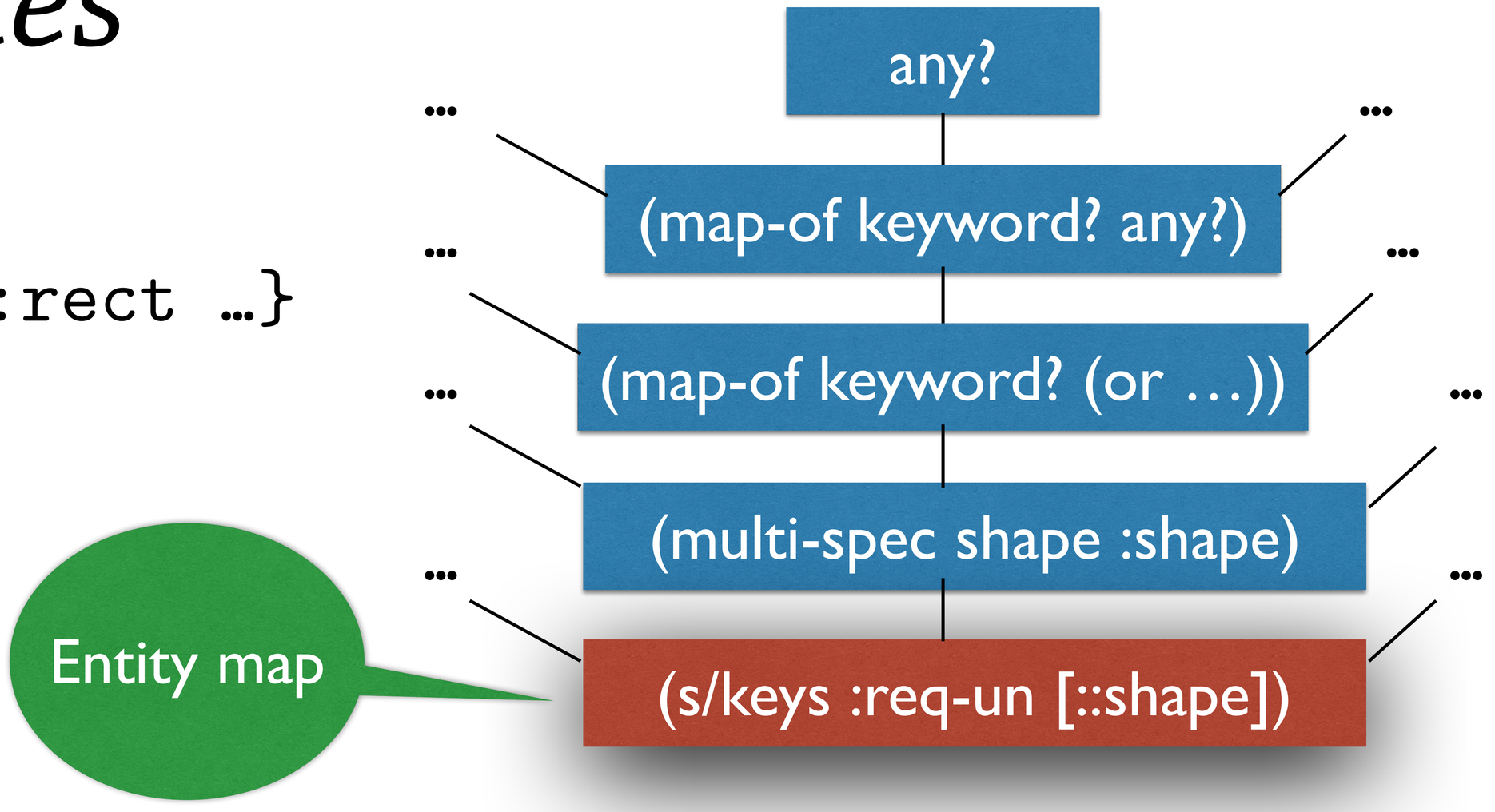
Chosen Spec



Observed Values

Chosen Spec

```
{:shape :rect …}
```

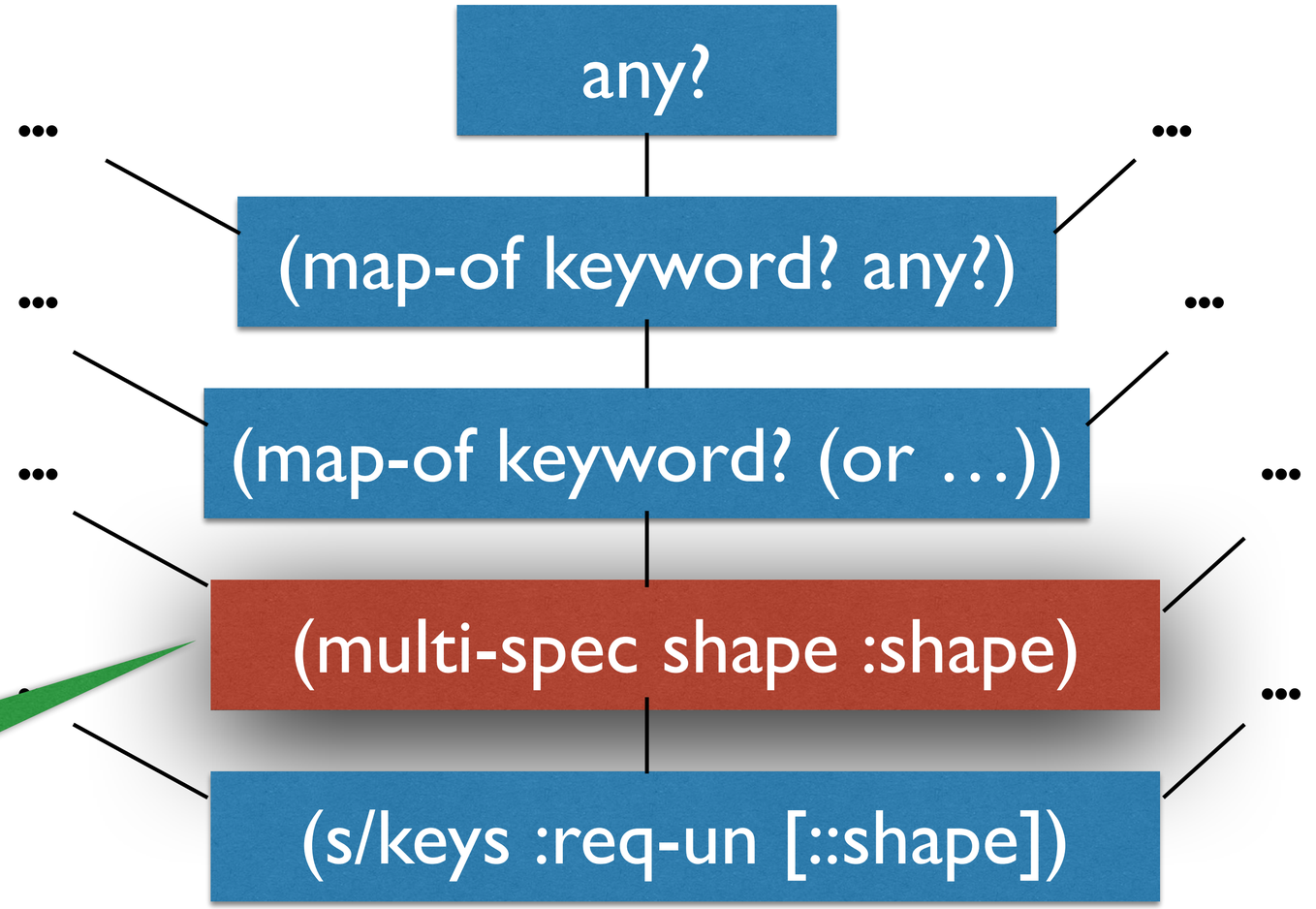


Observed Values

Chosen Spec

```
{:shape :rect …}

{:shape :circle …}
```



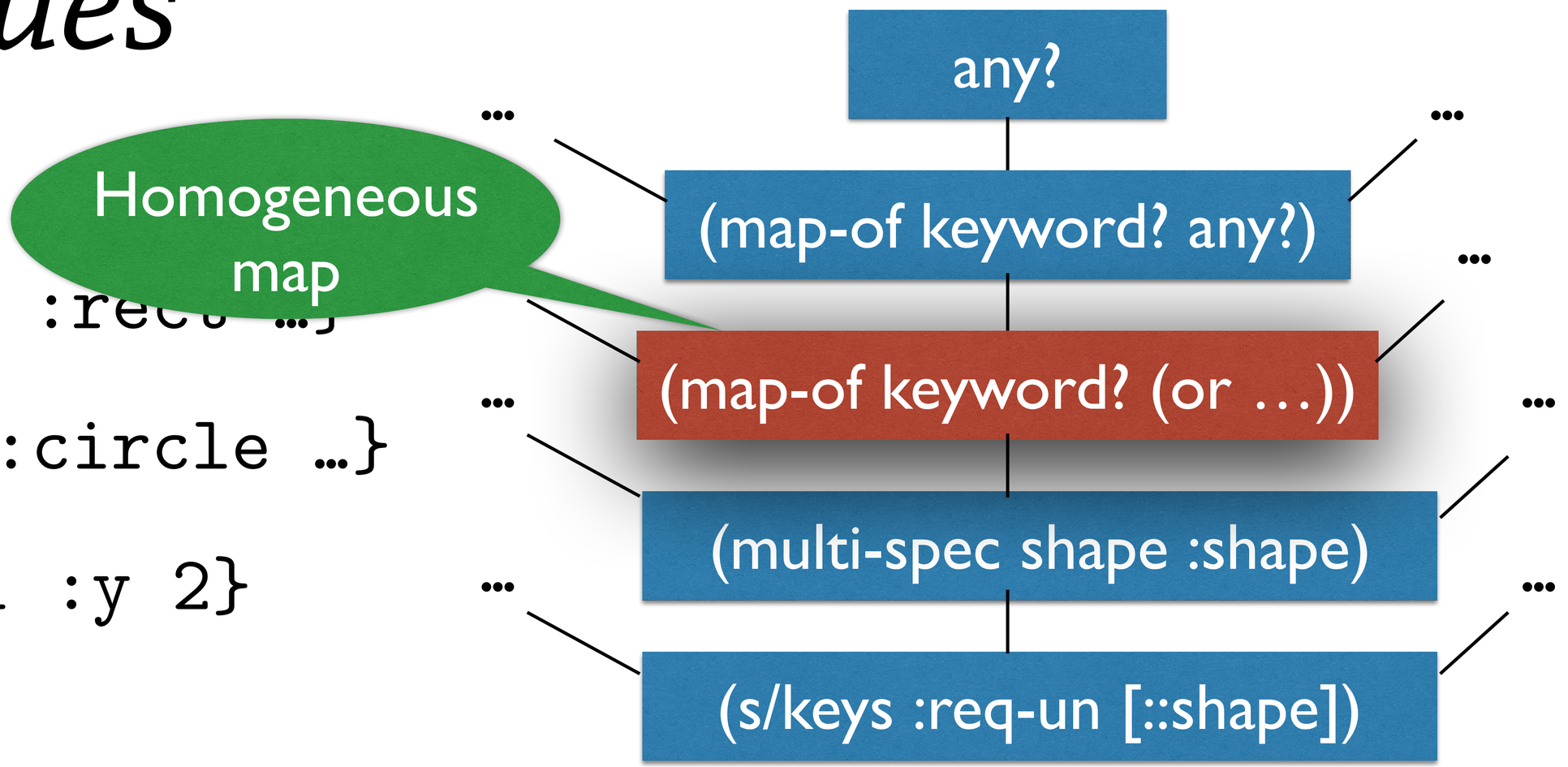
Observed Values

Chosen Spec

```
{:shape :rect …}

{:shape :circle …}

    {:x 1 :y 2}
```



Observed Values

Chosen Spec

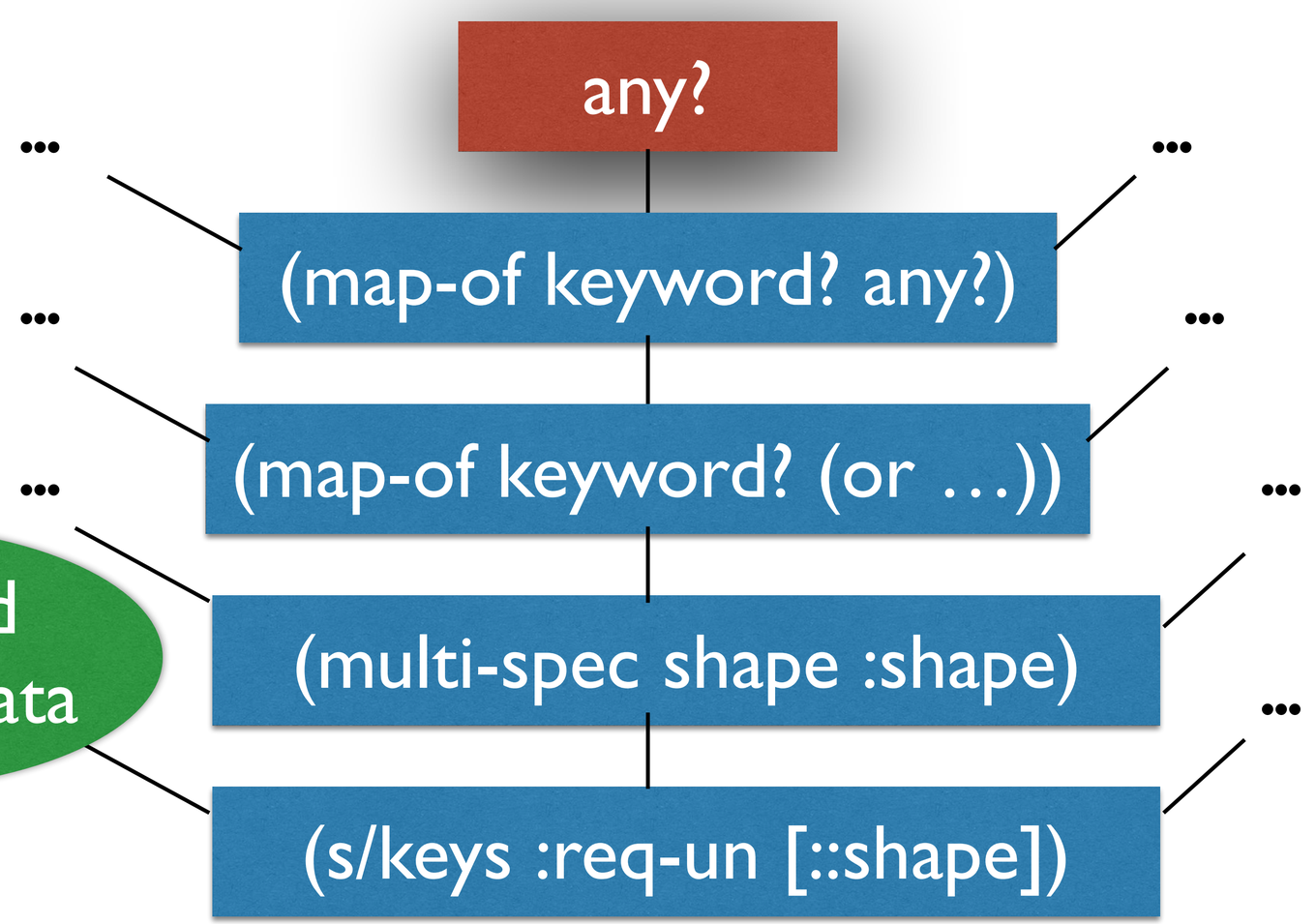
```
{:shape :rect …}

{:shape :circle …}

{:x 1 :y

42   'sym   :kw
```

Observed non-map data



Demo

Real-world Examples

The good…

Names

```
28  +(s/fdef expt-int :args (s/cat :base int? :pow int?) :ret int?)
29  +(s/fdef
30  +  init
31  +  :args
32  +  (s/cat :n int? :s (s/or :nil? nil? :int? int?))
33  +  :ret
34  +  (s/coll-of int?))
35  +(s/fdef
36  +  count-permutations-from-frequencies
37  +  :args
38  +  (s/cat :freqs (s/map-of (s/or :char? char? :int? int?) int?))
39  +  :ret
40  +  int?)
```

Reuse argument names

Good spec tags

Simple collection manipulation

```
221  +(s/fdef
222  +  lex-permutations
223  +  :args
224  +  (s/cat :c (s/coll-of int?))
225  +  :ret
226  +  (s/coll-of (s/coll-of int? :into vector?)))
```

Succinct

Multiple arities

```
140  +(s/fdef
141  +  date-time
142  +  :args
143  +  (s/alt
144  +    :1-arg
145  +    (s/cat :year int?)
146  +    :2-args
147  +    (s/cat :year int? :month int?)
148  +    :3-args
149  +    (s/cat :year int? :month int? :day int?)
150  +    :4-args
151  +    (s/cat :year int? :month int? :day int? :hour int?)
152  +    :5-args
153  +    (s/cat :year int? :month int? :day int? :hour int? :minute int?)
154  +    :6-args
```

Multi-specs

```
+(defmulti op-multi-spec4772367 :op)
+(defmethod
+  op-multi-spec4772367
+  :map
+  [_]
+  (s/keys
+    :req-un
+    [:clojure.core.typed.unqualified-keys/children
+     :clojure.core.typed.unqualified-keys/env
+     :clojure.core.typed.unqualified-keys/form
+     :clojure.core.typed.unqualified-keys/keys
+     :clojure.core.typed.unqualified-keys/op
+     :clojure.core.typed.unqualified-keys/tag
+     :clojure.core.typed.unqualified-keys/vals]))
```

{:op :map ..}

```
50 +(defmethod
51 +  op-multi-spec4772367
52 +  :vector
53 +  [_]
54 +  (s/keys
55 +    :req-un
56 +    [:clojure.core.typed.unqualified-keys/children
```

{:op :vector ..}

```
62 +(defmethod
63 +  op-multi-spec4772367
64 +  :var
65 +  [_]
66 +  (s/keys
67 +    :req-un
68 +    [:clojure.core.typed.unqualified-keys/env
```

{:op :var ..}

Useful aliases

```
+(s/fdef
+  munge
+  :args
+  (s/alt
+    :1-arg
+    (s/cat
+      :s
+      (s/or
+        :ColumnLineNameMap
+        ::ColumnLineNameMap
+        :NameNsArglistsMap
+        ::NameNsArglistsMap))
+    :2-args
```

Usage

```
+(s/fdef
+  shadow-depth
+  :args
+  (s/cat
+    :s
+    (s/or
+      :ColumnLineNameMap
+      ::ColumnLineNameMap
+      :NameNsArglistsMap
+      ::NameNsArglistsMap))
+  :ret
+  int?)
```

Usage

```
312 +(s/def
313 +  ::NameNsArglistsMap
314 +  (s/keys
315 +    :req-un
316 +    [:clojure.core.typed.unqualified-keys/name
317 +     :clojure.core.typed.unqualified-keys/ns]
318 +    :opt-un
319 +    [:clojure.core.typed.unqualified-keys/arglists
320 +     :clojure.core.typed.unqualified-keys/arglists-meta
321 +     :clojure.core.typed.unqualified-keys/column
322 +     :clojure.core.typed.unqualified-keys/doc
```

Extract common map into alias

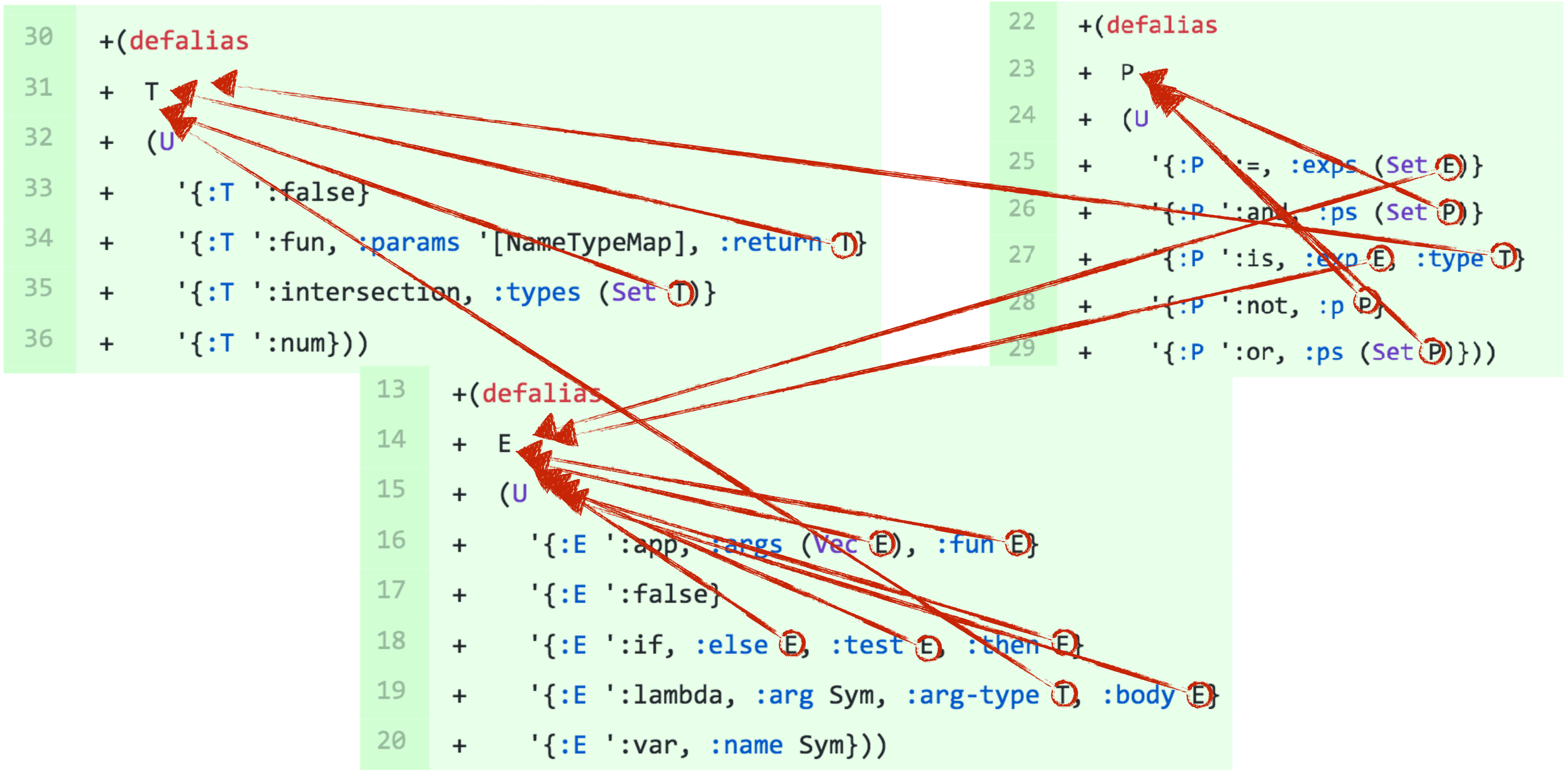
Mutually recursive maps

```
30  +(defalias
31  +  T
32  +  (U
33  +    '{:T ':false}
34  +    '{:T ':fun, :params '[NameTypeMap], :return T}
35  +    '{:T ':intersection, :types (Set T)}
36  +    '{:T ':num}))
```

```
22  +(defalias
23  +  P
24  +  (U
25  +    '{:P ':=, :exps (Set E)}
26  +    '{:P ':and, :ps (Set P)}
27  +    '{:P ':is, :exp E, :type T}
28  +    '{:P ':not, :p P}
29  +    '{:P ':or, :ps (Set P)}))
```

```
13  +(defalias
14  +  E
15  +  (U
16  +    '{:E ':app, :args (Vec E), :fun E}
17  +    '{:E ':false}
18  +    '{:E ':if, :else E, :test E, :then E}
19  +    '{:E ':lambda, :arg Sym, :arg-type T, :body E}
20  +    '{:E ':var, :name Sym}))
```

Mutually recursive maps



The bad…

```
201 +(s/fdef
202 +  lex-partitions-H
203 +  :args
204 +  (s/alt
205 +    :1-arg
206 +    (s/cat :N int?)
207 +    :5-args
208 +    (s/cat
209 +      :N
210 +      int?
211 +      :rest-arg-0
212 +      #{:min}
213 +      :rest-arg-1
214 +      int?
215 +      :rest-arg-2
216 +      #{:max}
217 +      :rest-arg-3
218 +      int?))
```

:min kw arg

:max kw arg

Lost kw arguments

Over-specificity

```
295  +(s/fdef
296  +  multi-perm
297  +  :args
298  +  (s/cat :l (s/tuple int? int? int?))
299  +  :ret
300  +  (s/coll-of (s/coll-of int?)))
```

Only unit tested 3-tuple!

The ugly…

No rest arg inference

```
+(s/fdef
+  element
+  :args
+  (s/alt
+    :13-args
+    (s/cat
+      :element-0
+      keyword?
+      :element-1
+      ::HIJMap
+      :element-2
+      string?
+      :element-3
+      (partial instance? clojure.data.xml.node.Element)
+      :element-4
+      string?
```

13 args

```
+      :element-9
+      (partial instance? clojure.data.xml.node.Element)
+      :element-10
+      string?
+      :element-11
+      (partial instance? clo                    t)
+      :element-12
+      string?)
+    :6-args
+    (s/cat
+      :element-0
+      keyword?
+      :element-1
+      nil?
+      :element-2
+      (partial instance? clojure.data.xml.node.Element)
+      :element-3
+      string?
+      :element-4
+      (partial instance? clojure.data.xml.node.Element)
+      :element-5
+      string?)
```

6 args

The future

Polymorphic types

```
19 (t/ann identity [t/Int :-> t/Int])
20 (comment (t/ann identity (t/All [x] [x :-> x])))
21 (comment (t/ann identity (t/All [x] [x :-> x])))
```

Polymorphic types

Fragments of polymorphic types

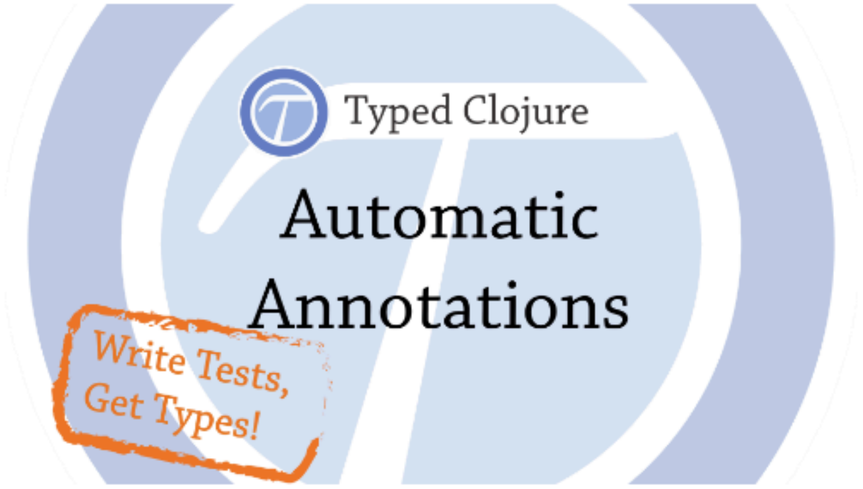
```
23 (t/ann
24   mymap
25   [[(t/U t/Int t/Sym) :-> (t/U t/Str t/Int)]
26    (t/Vec (t/U t/Int t/Sym))
27    :->
28    (t/Coll (t/U t/Str t/Int))])
29 (comment (t/ann mymap (t/All [x] [[x :-> ?] (t/Vec x) :-> ?])))
30 (comment (t/ann mymap (t/All [x] [[x :-> ?] '[? ? x] :-> ?])))
31 (comment (t/ann mymap (t/All [x] [[? :-> x] ? :-> (t/Coll x)])))
32 (comment (t/ann mymap (t/All [x] [[x :-> ?] '[x ? ?] :-> ?])))
33 (comment (t/ann mymap (t/All [x] [[x :-> ?] '[? x ?] :-> ?])))
34 (comment (t/ann mymap (t/All [x] [[? :-> x] ? :-> (t/Coll x)])))
35 (comment (t/ann mymap (t/All [x] [[x :-> ?] '[x] :-> ?])))
```

Questions,
Demos, and
magic tricks?

Thanks

ambrosebs.com/auto-ann.html

Automatic Annotations for Typed Clojure + clojure.spec



This page summarises my work on automatic annotation generation.

Library annotations

Here I will list a bunch of libraries we have generated annotations for. They don't type check, but the idea is they're very close--- and with good alias names! Last updated: 3rd April 2017

startrek-clojure	Generated	core.typed	Manually type checked	diff clojure.spec
math.combinatorics	Generated	core.typed	Manually type checked	diff clojure.spec

Inference results via side effects

```
(point 1 2)
; ['point {:dom 0}] : Long
; ['point {:dom 1}] : Long
; ['point :rng (key :x)] : Long
; ['point :rng (key :y)] : Long

{:x 1
 :y 2}
```

Step 1: $\mathrm{gen}\Gamma : r \rightarrow \Gamma$

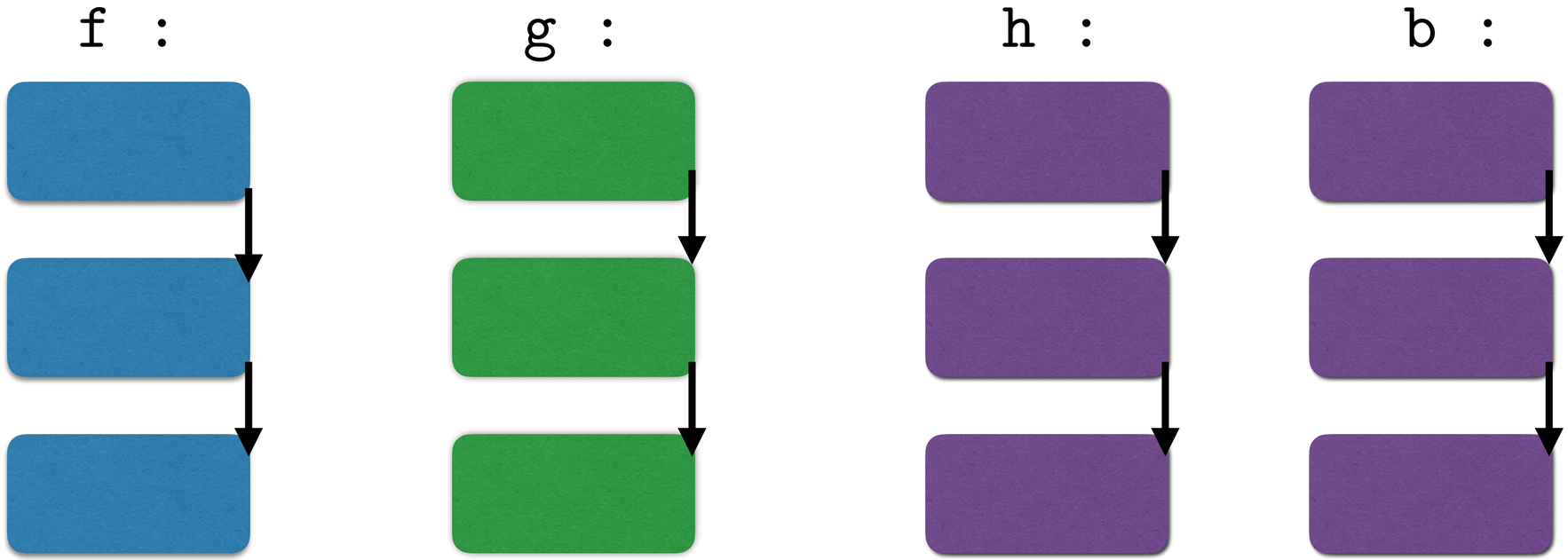
1) Generate naive type environment from dynamic inference results

```
; ['point {:dom 0}] : Long
; ['point {:dom 1}] : Long
; ['point :rng (key :x)] : Long
; ['point :rng (key :y)] : Long
```

```
point : [Long Long -> '{:x Long :y Long}]
```

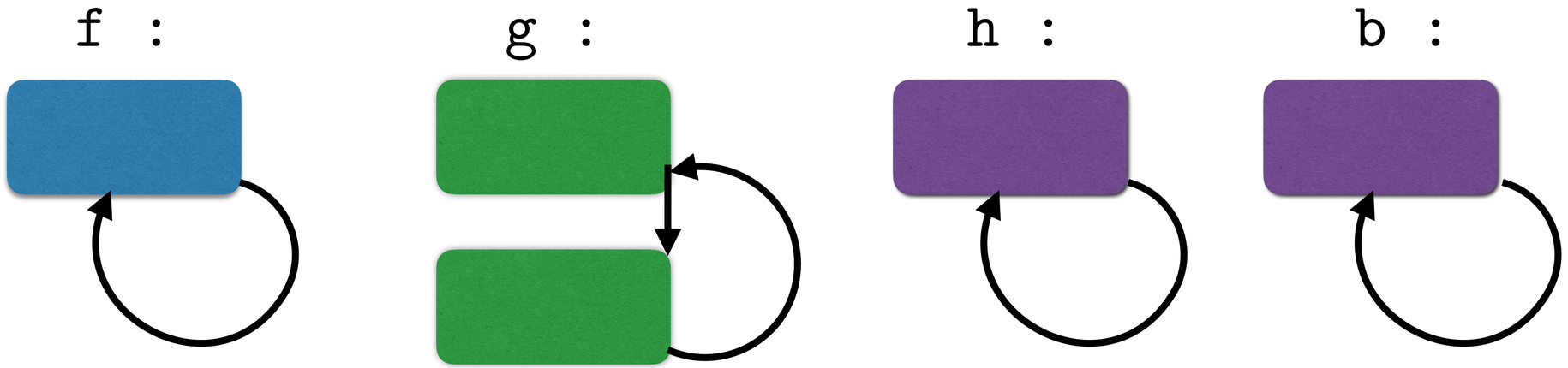
Step 2: squashLocal : $\Gamma \to \Delta$

2) Create local recursive types (“vertically”)



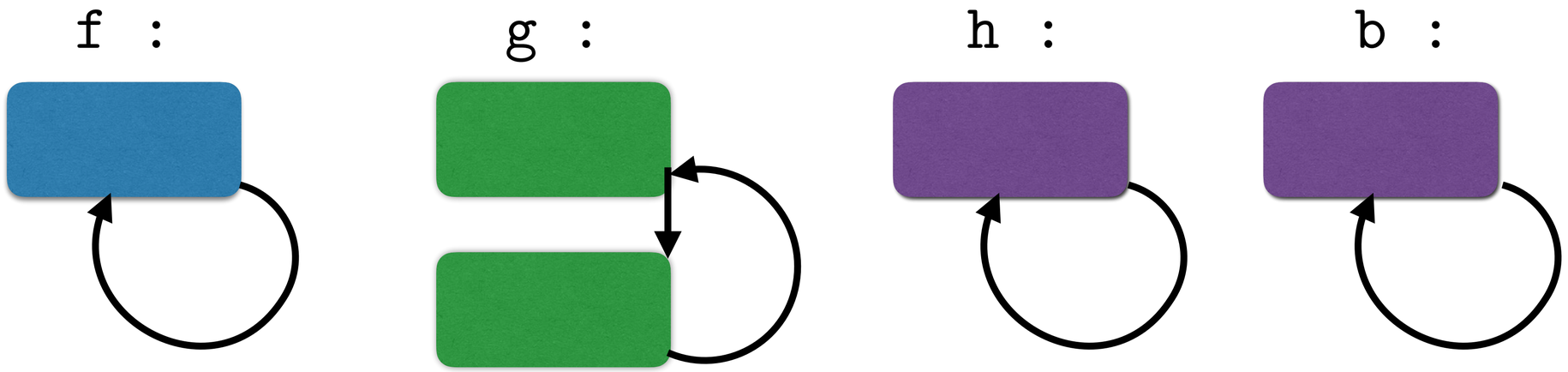
Step 2: squashLocal : $\Gamma \to \Delta$

2) Create local recursive types (“vertically”)



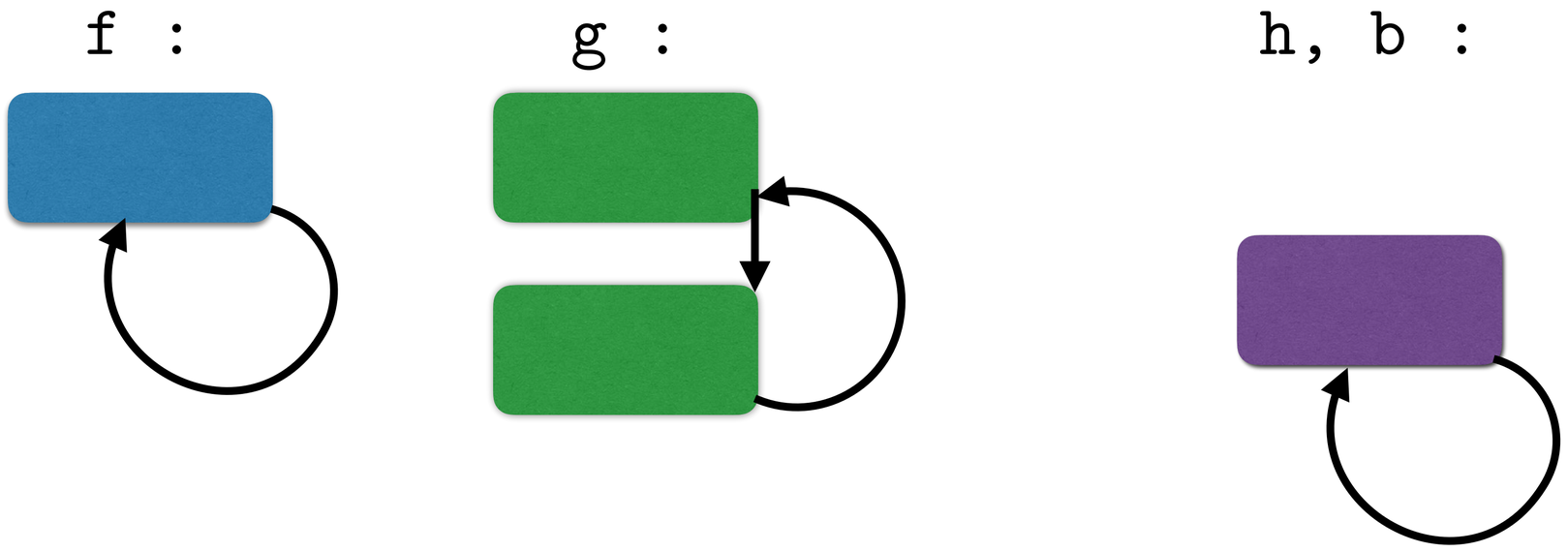
Step 3: squashGlobal : $\Delta \rightarrow \Delta$

3) Merge possibly-recursive types globally (“horizontally”)



Step 3: squashGlobal : $\Delta \rightarrow \Delta$

3) Merge possibly-recursive types globally (“horizontally”)



Instrumentation

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

Instrumentation

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

Paths

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

[emit (arg)]

[emit (ret)]

[emit (ret)]