Typed Clojure

INDIANA UNIVERSITY

Squash the work!

Inferring Useful Types and Contracts via Dynamic Analysis

Ambrose Bonnaire-Sergeant
Sam Tobin-Hochstadt

The Work…

- You're porting an untyped file to an optional type system
  - So you …

Stare…

```
(defn- all-different?
  "Annoyingly, the built-in distinct? doesn't handle 0 args, so we need
to write our own version that considers the empty-list to be distinct"
  [s]
  (if (seq s)
    (apply distinct? s)
    true))

(defmacro assert-with-message
  "Clojure 1.2 didn't allow asserts with a message, so we roll our own here for backwards compatibility"
  [x message]
  (when *assert*
    `(when-not ~x
       (throw (new AssertionError (str "Assert failed: " ~message "\n" (pr-str '~x)))))))

;; so this code works with both 1.2.x and 1.3.0:
(def ^{:private true} plus (first [+' +]))
(def ^{:private true} mult (first [*' *]))

(defn- index-combinations
  [n cnt]
  (lazy-seq
    (let [c (vec (cons 0 (for [j (range 1 (inc n))] (+ j cnt (- (inc n)))))),
          iter-comb
          (fn iter-comb [c j]
            (if (> j n) nil
```

… then Annotate

```
+(t/ann
+  bounded-distributions
+  [(t/Vec t/Int) t/Int :-> (t/Coll (t/Vec '[t/Int t/Int t/Int]))])
+(t/ann
+  cartesian-product
+  (t/IFn
+    [(t/Vec t/Int)
+     (t/Vec t/Int)
+     (t/Vec t/Int)
+     :->
+     (t/Coll (t/Coll t/Int))]
+    [(t/Vec t/Int) (t/Vec t/Int) :-> (t/Coll (t/Coll t/Int))]))
```

and Stare … (hmm Knuth? …)

```
;; Combinations of multisets
;; The algorithm in Knuth generates in the wrong order, so this is a new algorithm
(defn- multi-comb
  "Handles the case when you want the combinations of a list with duplicate items."
  [l t]
  (let [f (frequencies l),
        v (vec (distinct l)),
        domain (range (count v))
        m (vec (for [i domain] (f (v i))))
        qs (bounded-distributions m t)]
    (for [q qs]
      (apply concat
             (for [[index this-bucket _] q]
               (repeat this-bucket (v index)))))))
```

global annotation…

```
+(t/ann
+  multi-comb
+  [(t/Vec (t/U t/Int Character))
+   t/Int
+   :->
+   (t/Coll (t/Coll (t/U t/Int Character)))])
```

…local annotations…

```
  ;; Combinations of multisets
  ;; The algorithm in Knuth generates in the wrong order, so this is a new algorithm
  (defn- multi-comb
    "Handles the case when you want the combinations of a list with duplicate items."
    [l t]
    (let [f (frequencies l),
          v (vec (distinct l)),
          domain (range (count v))
-         m (vec (for [i domain] (f (v i))))
+         m (vec (for ^{::t/ann t/Int} [^{::t/ann t/Int} i domain] (f (v i))))
          qs (bounded-distributions m t)]
-     (for [q qs]
+     (for ^{::t/ann (t/Coll (t/U t/Int Character))} [^{::t/ann (t/Vec '[t/Int t/Int t/Int])} q qs]
        (apply concat
-              (for [[index this-bucket _] q]
+              (for ^{::t/ann (t/Coll (t/U t/Int Character))} [^{::t/ann '[t/Int t/Int t/Int]} [index this-buck
                 (repeat this-bucket (v index)))))))
```

…stare (… ahhh…Knuth.)

```
(defn- m5  ; M5
  [n m f c u v a b l r s]
  (let [j (loop [j (dec b)]
            (if (not= (v j) 0)
              j
              (recur (dec j))))]
    (cond
      (and r
           (= j a)
           (< (* (dec (v j)) (- r 1))
              (u j))) (m6 n m f c u v a b l r s)
      (and (= j a)
           (= (v j) 1)) (m6 n m f c u v a b l r s)
      :else (let [v (update v j dec)
                  diff-uv (if s (apply + (for [i (range a (inc j))]
                                           (- (u i) (v i)))) nil)
                  v (loop [ks (range (inc j) b)
                           v v]
                      (if (empty? ks)
                        v
                        (let [k (first ks)]
                          (recur (rest ks)
```

annotate … m5 … m6 .. m.. zzzz

```
+(t/ann
+  m5
+  [t/Int
+   t/Int
+   (t/Vec t/Int)
+   (t/Vec t/Int)
+   (t/Vec t/Int)
+   (t/Vec t/Int)
+   t/Int
+   t/Int
+   t/Int
+   (t/U nil t/Int)
+   (t/U nil t/Int)
+   :->
+   (t/Coll (t/Coll (t/Map t/Int t/Int)))])
```

```
+(t/ann
+  m6
+  [t/Int
+   t/Int
+   (t/Vec t/Int)
+   (t/Vec t/Int)
+   (t/Vec t/Int)
+   (t/Vec t/Int)
+   t/Int
+   t/Int
+   t/Int
+   (t/U nil t/Int)
+   (t/U nil t/Int)
+   :->
+   (t/Coll (t/Coll (t/Map t/Int t/Int)))])
```

Help needed!!
Can we automate?

What if your diffs looked like this?

```diff
 (t/ann
   count-combinations-from-frequencies
-  [(t/Map (t/U t/Int Character) t/Int) t/Int :-> t/Int])
+  [(t/Map t/Any t/Int) t/Int :-> t/Int])
 (t/ann
   count-combinations-unmemoized
   [(t/Vec (t/U t/Int Character)) t/Int :-> t/Int])
-(t/ann count-permutations [(t/Coll (t/U t/Int Character)) :-> t/Int])
+(t/ann count-permutations [(t/Coll t/Any) :-> t/Int])
```

…and this?

```
(defn- initial-perm-numbers
  "Takes a sorted frequency map and returns how far into the sequence of
lexicographic permutations you get by varying the first item"
  [freqs]
  (reductions + 0
-             (for ^{::t/ann t/Int} [^{::t/ann '[t/Int t/Int]} [k v] freqs]
+             (for ^{::t/ann t/Int} [^{::t/ann '[t/Any t/Int]} [k v] freqs]
                (count-permutations-from-frequencies (assoc freqs k (dec v))))))
```

… or no diff at all… :)

```
(t/ann
  m5
  [t/Int
   t/Int
   (t/Vec t/Int)
   (t/Vec t/Int)
   (t/Vec t/Int)
   (t/Vec t/Int)
   t/Int
   t/Int
   t/Int
   (t/U nil t/Int)
   (t/U nil t/Int)
   :->
   (t/Coll (t/Coll (t/Map t/Int t/Int)))])
```

```
(t/ann
  m6
  [t/Int
   t/Int
   (t/Vec t/Int)
   (t/Vec t/Int)
   (t/Vec t/Int)
   (t/Vec t/Int)
   t/Int
   t/Int
   t/Int
   (t/U nil t/Int)
   (t/U nil t/Int)
   :->
   (t/Coll (t/Coll (t/Map t/Int t/Int)))])
```

Squash the work!

Background

- Optional/gradual types and contracts are popular verification tools for dynamically typed languages
  - Usually heavily rely on annotations

Problem

- Must keep annotations in sync with code
    - Initial annotation cost, versioning, libraries, iterative changes
- Almost always a manual effort
    - Annotating costs time + error prone

Motivation

- **Reduce time+effort spent annotating**
  - Can we **automate** keeping annotations in sync with code?
- *Benefits*
  - Get more programmers quickly and easily started with verification ✓
  - Help existing users evolve annotations along with code ✓
  - Ultimately encourage more code to be verified ✓

Non-goals

- 100% correct annotations ✗
  - "Useful" annotations are good enough ✓

Our setting

Our setting

- Typed Clojure
  - optional type system for Clojure
- Clojure.spec
  - contract system for Clojure

Typed Clojure

Clojure

Typed Clojure

```clojure
(t/ann remove-nth [(t/Coll t/Int) t/Int :-> (t/Vec t/Int)])
(t/ann selections [(t/Vec t/Int) t/Int :-> (t/Coll (t/Coll t/Int))])
```

```clojure
;; Helper function for bounded-distributions
(defn- distribute [m index total distribution already-distributed]
  (loop [^{::t/ann (t/Vec '[t/Int t/Int t/Int])} distribution distribution
         ^{::t/ann t/Int} index index
         ^{::t/ann t/Int} already-distributed already-distributed]
    (if (>= index (count m)) nil
      (let [quantity-to-distribute (- total already-distributed)
            mi (m index)]
        (if (<= quantity-to-distribute mi)
          (conj distribution [index quantity-to-distribute total])
          (recur (conj distribution [index mi (+ already-distributed mi)])
                 (inc index)
                 (+ already-distributed mi)))))))
```

Clojure.spec

```clojure
(s/fdef
  selections
  :args
  (s/cat :items (s/coll-of int?) :n int?)
  :ret
  (s/coll-of (s/coll-of int?)))
```

```clojure
(s/fdef
  remove-nth
  :args
  (s/cat :l (s/coll-of int?) :n int?)
  :ret
  (s/coll-of int?))
```

Dynamic Analysis

Dynamic Analysis

- Observe and collect information on running programs
    - Via unit/generative tests, dummy runs

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}
```

Runtime Instrumentation

```
(track e path)
;=> v
```

Wrap *e* as *v*, where *path* is the original source of *e*.

Top-level typed bindings

```
(def b e)
```

```
(def b (track e ['b]))
```

Summarizing execution

```
(def forty-two 42)
```

```
(def forty-two
  (track 42 ['forty-two]))
```

```
; Inference result:
; ['forty-two] : Long
(def forty-two 42)
```

Track functions (part 1)

```clojure
(defn point [x y]
  {:x x
   :y y})
```

```clojure
; Int Int -> Point
(def point
  (track
    (fn [x y]
      {:x x
       :y y})
    ['point]))
```

Track functions (part 2)

```
; Int Int -> Point
(def point
  (track
    (fn [x y]
      {:x x
       :y y})
    ['point]))
```

```
(def point
  (fn [x y]
    (track
      ((fn [x y]
         {:x x
          :y y})
       (track x ['point {:dom 0}])
       (track y ['point {:dom 1}]))
      ['point :rng])))
```

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}
```

Connecting the dots

```clojure
(def forty-two 42)
```

$$\Gamma = \{\texttt{forty-two} : \texttt{Long}\}$$

```clojure
(def forty-two
  (track 42 ['forty-two]))
```

```clojure
; Inference result:
; ['forty-two] : Long
(def forty-two 42)
```

Connecting the dots

```
(def forty-two 42)
```

$$\Gamma = \{\texttt{forty-two} : \texttt{Long}\}$$

```
(def forty-two
  (track 42 ['forty-two]))
```

How?

```
; Inference result:
; ['forty-two] : Long
(def forty-two 42)
```

Converting Dynamic Inference results to useful annotations

From inference results, to type environments

$$\mathrm{inferAnns} : r \to \Delta$$

Inference results

l	$::= x \mid \mathbf{dom} \mid \mathbf{rng} \mid \mathbf{key}_{\vec{k}}(k)$	Path Elements
π	$::= \overrightarrow{l}$	Paths
r	$::= \{\overrightarrow{\pi : \tau}\}$	Inference results

Type environments

Γ	$::=$	$\{\overrightarrow{x : \tau}\}$	Type environments
A	$::=$	$\{\overrightarrow{a \mapsto \tau}\}$	Type alias environments
Δ	$::=$	(A, Γ)	Combined environments

From inference results, to type environments

r	$::= \{\overrightarrow{\pi : \tau}\}$	Inference results
Δ	$::= (A, \Gamma)$	Combined environments

$$\text{inferAnns} : r \rightarrow \Delta$$

Our approach

$$\mathsf{inferAnns} : r \rightarrow \Delta$$
$$\mathsf{inferAnns} = \mathsf{squashGlobal} \circ \mathsf{squashLocal} \circ \mathsf{gen}\Gamma$$

Our approach

$$\text{inferAnns} : r \to \Delta$$
$$\text{inferAnns} = \text{squashGlobal} \circ \text{squashLocal} \circ \text{gen}\Gamma$$

$$\begin{aligned} \text{gen}\Gamma &: r \to \Gamma \\ \text{squashLocal} &: \Gamma \to \Delta \\ \text{squashGlobal} &: \Delta \to \Delta \end{aligned}$$

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

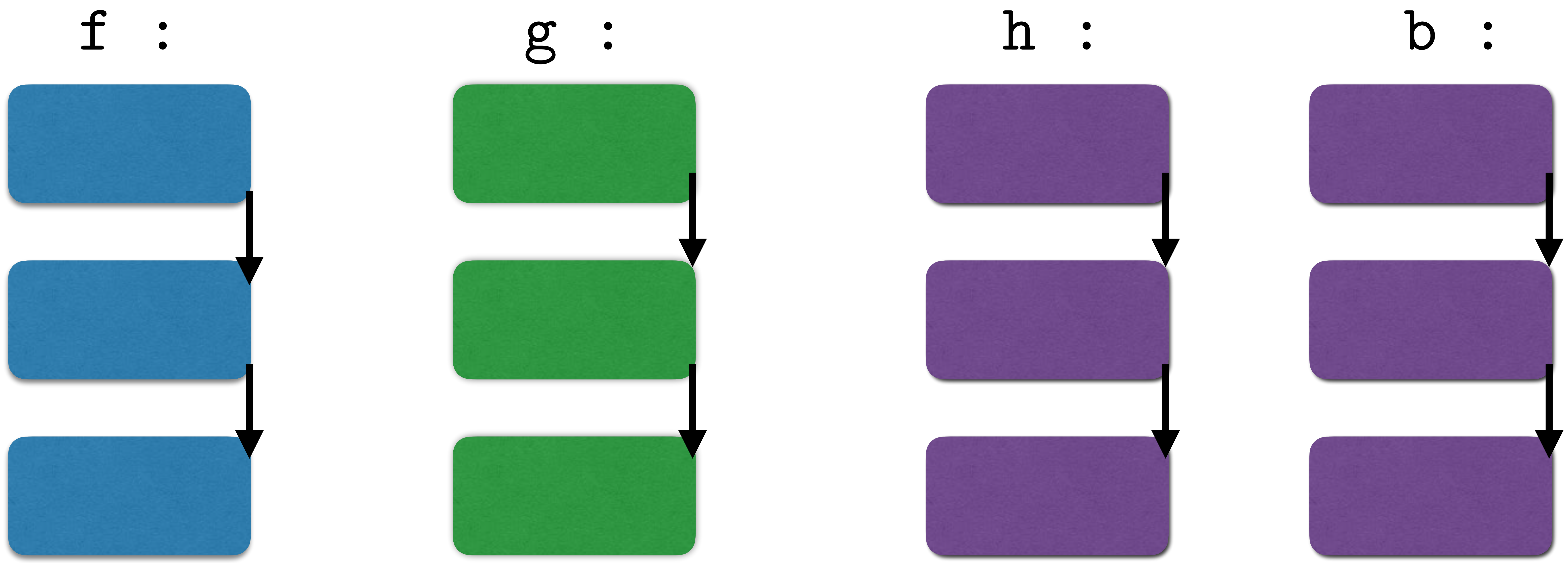
1) Generate naive type environment from dynamic inference results

```
; ['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}]
```

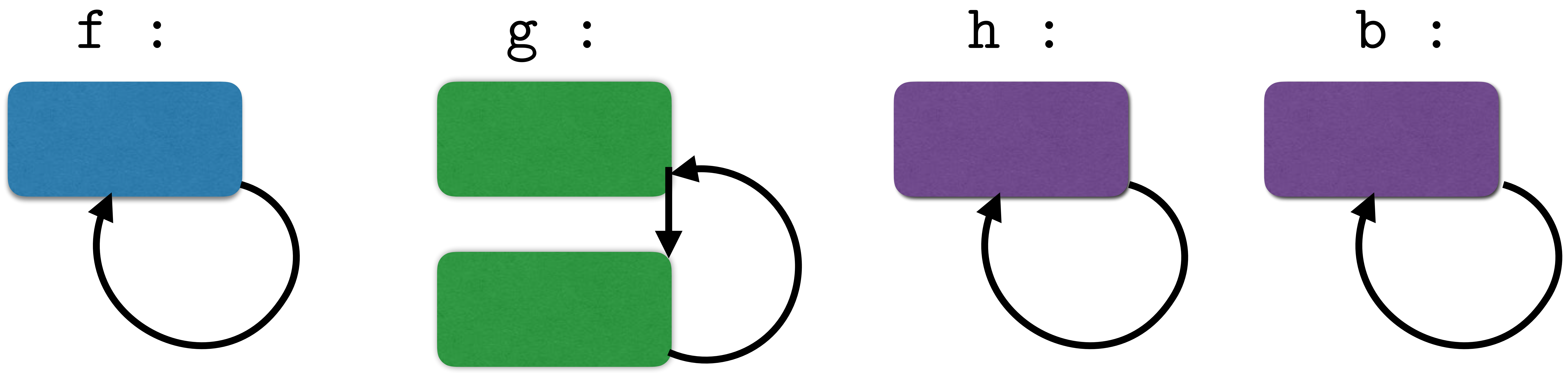
Step 2: squashLocal : $\Gamma \rightarrow \Delta$

2) Create local recursive types ("vertically")



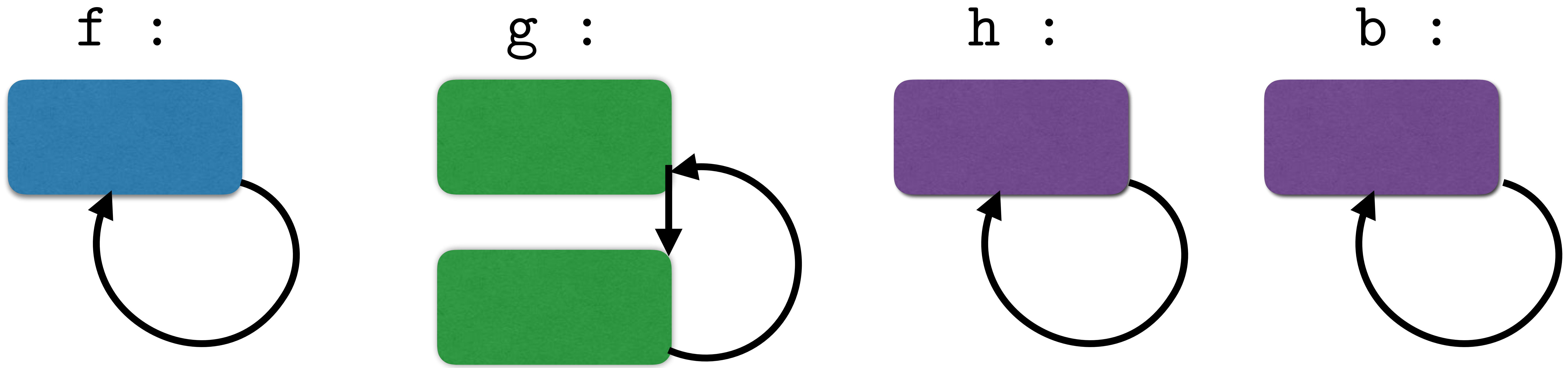
Step 2: squashLocal : $\Gamma \rightarrow \Delta$

2) Create local recursive types ("vertically")



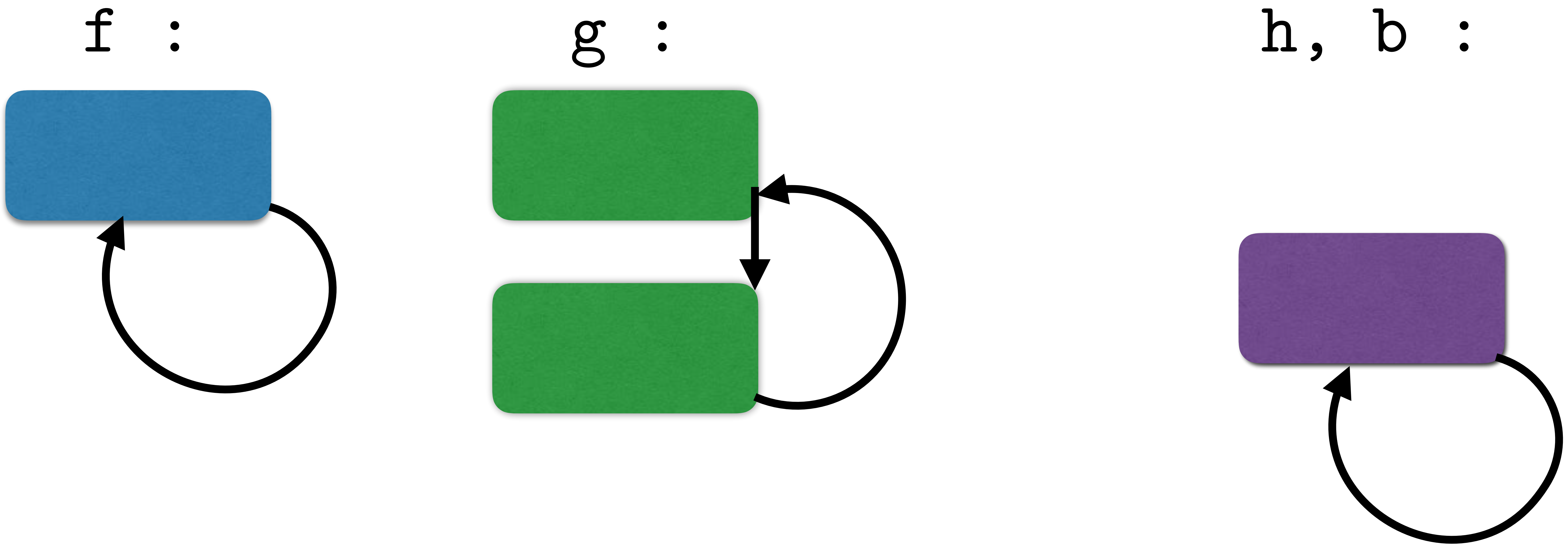
Step 3: squashGlobal : $\Delta \rightarrow \Delta$

3) Merge possibly-recursive types globally ("horizontally")



Step 3: squashGlobal : Δ → Δ

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



Experiments

Experiment 1: Annotation quality

- Compactness
- Accuracy
- Organization

Naming

- Reusing names from program sources is effective

```
+(s/fdef expt-int :args (s/cat :base int? :pow int?) :ret int?)
+(s/fdef
+  init
+  :args
+  (s/cat :n int? :s (s/or :nil? nil? :int? int?))
+  :ret
+  (s/coll-of int?))
+(s/fdef
+  count-permutations-from-frequencies
+  :args
+  (s/cat :freqs (s/map-of (s/or :char? char? :int? int?) int?))
+  :ret
+  int?)
```

Crude naming is still informative

```
+(t/defalias AsFileAsUrlMap '{:as-file t/Any, :as-url t/Any})
+(t/defalias
+  DocImplsMethodBuildersMap
+  '{:doc t/Str,
+     :impls (t/Map (t/U nil Class) AsFileAsUrlMap),
+     :method-builders (t/Map clojure.lang.Var AnyFunction),
+     :method-map AsFileAsUrlMap,
+     :on t/Sym,
+     :on-interface Class,
+     :sigs AsFileAsUrlMap,
+     :var clojure.lang.Var})
```

Effectively annotate recursive data

```
+(defalias
+  T
+  (U
+    '{:T ':false}
+    '{:T ':fun, :params '[NameTypeMap], :return T}
+    '{:T ':intersection, :types (Set T)}
+    '{:T ':num}))
```

```
+(defalias
+  P
+  (U
+    '{:P ':=, :exps (Set E)}
+    '{:P ':and, :ps (Set P)}
+    '{:P ':is, :exp E, :type T}
+    '{:P ':not, :p P}
+    '{:P ':or, :ps (Set P)}))
```

```
+(defalias
+  E
+  (U
+    '{:E ':app, :args (Vec E), :fun E}
+    '{:E ':false}
+    '{:E ':if, :else E, :test E, :then E}
+    '{:E ':lambda, :arg Sym, :arg-type T, :body E}
+    '{:E ':var, :name Sym}))
```

Effectively annotate recursive data

```
+(defalias
+  T
+  (U
+    '{:T ':false}
+    '{:T ':fun, :params '[NameTypeMap], :return T}
+    '{:T ':intersection, :types (Set T)}
+    '{:T ':num}))
```

```
+(defalias
+  P
+  (U
+    '{:P ':=, :exps (Set E)}
+    '{:P ':and, :ps (Set P)}
+    '{:P ':is, :exp E, :type T}
+    '{:P ':not, :p P}
+    '{:P ':or, :ps (Set P)}))
```

```
+(defalias
+  E
+  (U
+    '{:E ':app, :args (Vec E), :fun E}
+    '{:E ':false}
+    '{:E ':if, :else E, :test E, :then E}
+    '{:E ':lambda, :arg Sym, :arg-type T, :body E}
+    '{:E ':var, :name Sym}))
```

Experiment 2: Runnable contracts

- Do the contracts pass the unit tests?
  - Yes.
  - A nice consistency/sanity check for the approach

Experiment 3: Manual delta

- Generate types
  - What kind of manual changes needed to type check?

```
 (defn- initial-perm-numbers
   "Takes a sorted frequency map and returns how far into the sequence of
 lexicographic permutations you get by varying the first item"
   [freqs]
   (reductions + 0
-              (for ^{::t/ann t/Int} [^{::t/ann '[t/Int t/Int]} [k v] freqs]
+              (for ^{::t/ann t/Int} [^{::t/ann '[t/Any t/Int]} [k v] freqs]
                 (count-permutations-from-frequencies (assoc freqs k (dec v))))))
```

Case study: Type checking `raynes/fs`

- 76 generated top-level annotations
  - 59 annotations out of the box!
  - 17 needed changes (22%)

```
(t/ann exists? [(t/U t/Str File) :-> Boolean])
```

```
-(t/ann copy-dir [File File :-> File])
-(t/ann copy-dir-into [File File :-> nil])
+(t/ann copy-dir [File File :-> (t/U nil File)])
+(t/ann copy-dir-into [File File :-> (t/U nil File)])
```

Case study: Type checking `raynes/fs`

- 50 casts manually added
  - Where to draw the typed/untyped boundary?

```
  (defn tmpdir
    "The temporary file directory looked up via the `java.io.tmpdir`
     system property. Does not create a temporary directory."
    []
+   {:post [(string? %)]}
    (System/getProperty "java.io.tmpdir"))
```

Over-specificity

- Can be overly specific for generic functions
  - No support for polymorphism

```
 (t/ann
   write-object
-  [(t/Map (t/U nil t/Str t/Int) t/Int) PrintWriter :-> nil])
+  [(t/Map t/Any t/Any) PrintWriter :-> nil])
```

Local annotations are useful

- We generate local annotations, sometimes very useful and saves a lot of work

Library	Lines of types	Local annotations	Manual Line +/- Diff
startrek-clojure	133	3	+70 -41
math.combinatorics	395	147	+124 -120
fs	157	1	+119 -86
data.json	168	9	+94 -125
mini.occ	49	1	+46 -26

Fig. 9. Generated types

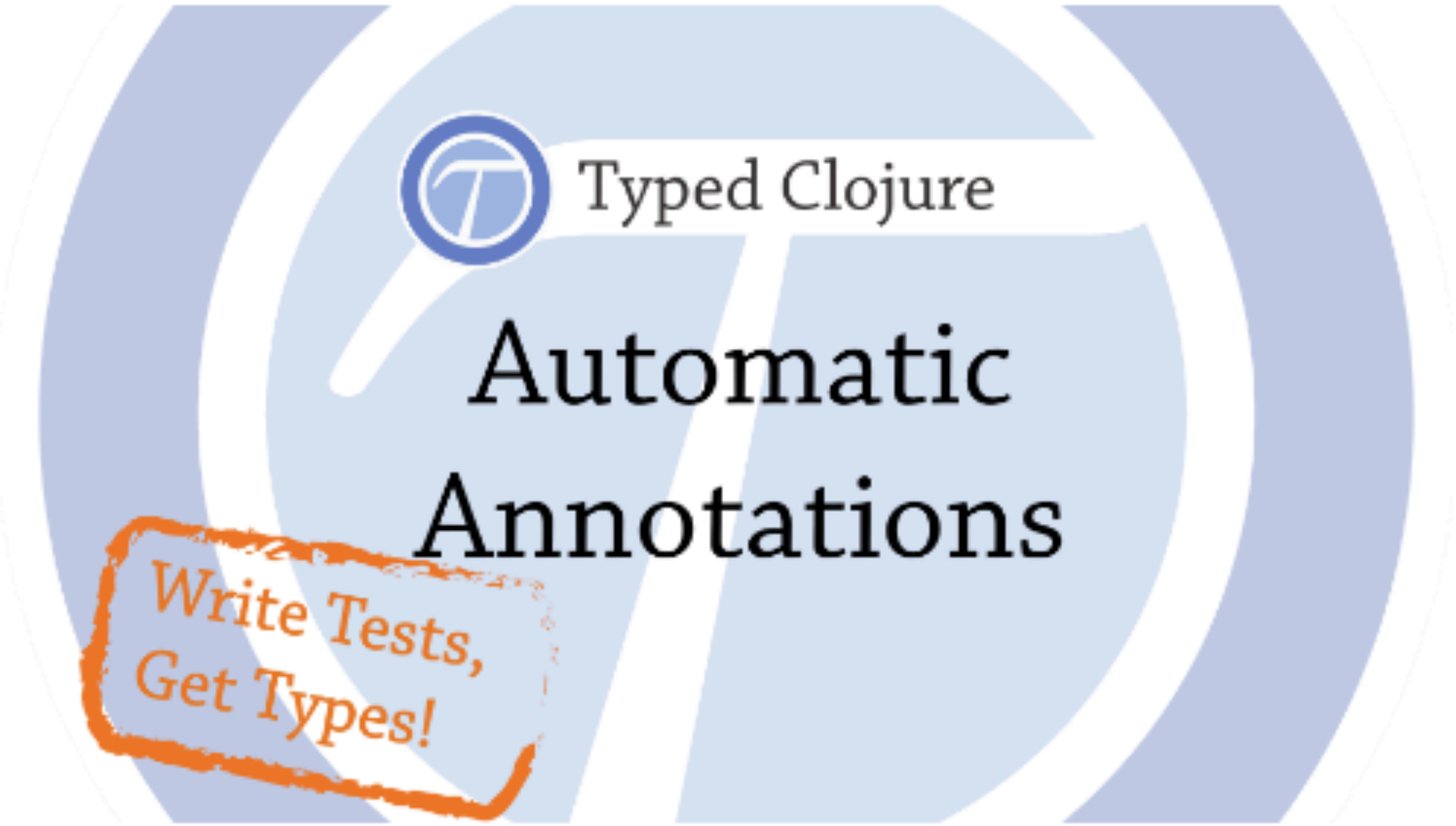
Case study: Type checking `math.combinatorics`

- 147 generated local annotations (counting 1 per fn arg/rng position)
  - 1 manually changed annotation, 8 local annotations skipped checking
  - 139+ useful annotations out of the box (93%)

```diff
-  (loop [freqs (into (sorted-map) (frequencies l)),
-         indices (factorial-numbers-with-duplicates n freqs)
-         perm []]
+  (loop [^{::t/ann (t/Map t/Int t/Int)} freqs (into (sorted-map) (frequencies l)),
+         ^{::t/ann (t/Coll t/Int)} indices (factorial-numbers-with-duplicates n freqs)
+         ^{::t/ann (t/Vec t/Int)} perm []]
```

ambrosebs.com

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
fs	Generated core.typed	Manually type checked diff	clojure.spec
data.json	Generated core.typed	Manually type checked diff	clojure.spec

Future work

- Incorporate+modify existing annotations
- More granular options for runtime tracking
  - Currently per-namespace only

Squash the work!

Thanks!

ambrosebs.com

Ambrose Bonnaire-Sergeant