Typed Clojure

INDIANA UNIVERSITY

Inferring Structural Types from Tests

Ambrose Bonnaire-Sergeant
Sam Tobin-Hochstadt

Typed Clojure

Clojure

- Immutable data structures 
- Hosted on JVM — Java
- Lisp-style Macros — ‘()
- Dynamic typing — *

Clojure

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

(point 1 2)
;=> {:x 1 :y 2}
```

Typed Clojure

Optional type system for Clojure

Supports existing idioms:

- Local flow typing
- Heterogeneous maps
- Java Interop
- Multimethods

[Bonnaire-Sergeant et al. ESOP 2016]

Typed Clojure

Annotations

```
(defalias Point
  '{:x Int
    :y Int})

(ann point [Int Int -> Point])
(defn point [x y]
  {:x x
   :y y})

(point 1 2)
```

Inferring annotations from Tests

Why?

- Gradual typing

Untyped ⟶ Typed

```
  (define anchor-bitmap (delay/sync (make-object bitmap% anchor-bitmap-path)))
  (define (get-anchor-bitmap) (force anchor-bitmap))
-
  (define lock-bitmap (delay/sync (make-object bitmap% lock-bitmap-path)))
  (define (get-lock-bitmap) (force lock-bitmap))
-
  (define unlock-bitmap (delay/sync (make-object bitmap% unlock-bitmap-path)))
  (define (get-unlock-bitmap) (force unlock-bitmap))
-
  (define autowrap-bitmap (delay/sync (make-object bitmap% return-bitmap-path)))
  (define (get-autowrap-bitmap) (force autowrap-bitmap))
-
  (define paren-highlight-bitmap (delay/sync (make-object bitmap% paren-bitmap-path)))
  (define (get-paren-highlight-bitmap) (force paren-highlight-bitmap))
-
```

```
+ #;
+ (: anchor-bitmap (Instanceof BitMap%))
  (define anchor-bitmap (delay/sync (make-object bitmap% anchor-bitmap-path)))
  (define (get-anchor-bitmap) (force anchor-bitmap))
+
+ #;
+ (: lock-bitmap (Instanceof BitMap%))
  (define lock-bitmap (delay/sync (make-object bitmap% lock-bitmap-path)))
+ #;
+ (: get-lock-bitmap (-> (Instanceof BitMap%)))
  (define (get-lock-bitmap) (force lock-bitmap))
+
+ #;
+ (: unlock-bitmap (Instanceof BitMap%))
  (define unlock-bitmap (delay/sync (make-object bitmap% unlock-bitmap-path)))
+ #;
+ (: get-unlock-bitmap (-> (Instanceof BitMap%)))
  (define (get-unlock-bitmap) (force unlock-bitmap))
+
+ #;
+ (: autowrap-bitmap (Instanceof BitMap%))
  (define autowrap-bitmap (delay/sync (make-object bitmap% return-bitmap-path)))
+ #;
+ (: get-autowrap-bitmap (-> (Instanceof BitMap%)))
  (define (get-autowrap-bitmap) (force autowrap-bitmap))
+
+ #;
+ (: paren-highlight-bitmap (Instanceof BitMap%))
  (define paren-highlight-bitmap (delay/sync (make-object bitmap% paren-bitmap-path)))
+ #;
+ (: get-paren-highlight-bitmap (-> (Instanceof BitMap%)))
  (define (get-paren-highlight-bitmap) (force paren-highlight-bitmap))
+
```

Untyped → Typed

```
(define anchor-bitmap (delay/sync (make-object bitmap% anchor-bitmap-path)))
(define (get-anchor-bitmap) (force anchor-bitmap))
-

(define lock-bitmap (delay/sync (make-object bitmap% lock-bitmap-path)))

(define (get-lock-bitmap) (force lock-bitmap))
-

(define unlock-bitmap (delay/sync (make-object bitmap% unlock-bitmap-path)))

(define (get-unlock-bitmap) (force unlock-bitmap))
-

(define autowrap-bitmap (delay/sync (make-object bitmap% return-bitmap-path)))

(define (get-autowrap-bitmap) (force autowrap-bitmap))
-

(define paren-highlight-bitmap (delay/sync (make-object bitmap% paren-bitmap-path)))

(define (get-paren-highlight-bitmap) (force paren-highlight-bitmap))
-
```

Annotations

```
+ #;
+ (: anchor-bitmap (Instanceof BitMap%))
  (define anchor-bitmap (delay/sync (make-object bitmap% anchor-bitmap-path)))
  (define (get-anchor-bitmap) (force anchor-bitmap))
+
+ #;
+ (: lock-bitmap (Instanceof BitMap%))
  (define lock-bitmap (delay/sync (make-object bitmap% lock-bitmap-path)))
+ #;
+ (: get-lock-bitmap (-> (Instanceof BitMap%)))
  (define (get-lock-bitmap) (force lock-bitmap))
+
+ #;
+ (: unlock-bitmap (Instanceof BitMap%))
  (define unlock-bitmap (delay/sync (make-object bitmap% unlock-bitmap-path)))
+ #;
+ (: get-unlock-bitmap (-> (Instanceof BitMap%)))
  (define (get-unlock-bitmap) (force unlock-bitmap))
+
+ #;
+ (: autowrap-bitmap (Instanceof BitMap%))
  (define autowrap-bitmap (delay/sync (make-object bitmap% return-bitmap-path)))
+ #;
+ (: get-autowrap-bitmap (-> (Instanceof BitMap%)))
  (define (get-autowrap-bitmap) (force autowrap-bitmap))
+
+ #;
+ (: paren-highlight-bitmap (Instanceof BitMap%))
  (define paren-highlight-bitmap (delay/sync (make-object bitmap% paren-bitmap-path)))
+ #;
+ (: get-paren-highlight-bitmap (-> (Instanceof BitMap%)))
  (define (get-paren-highlight-bitmap) (force paren-highlight-bitmap))
+
```

Why?

- Gradual typing
- Unfinished program state

"Based on the current tests, what are input/outputs?"

?

```
(defn g [m]
  (merge m {:x 1}))
```

"Based on the current tests, what are input/outputs?"

Documentation

```
(ann g ['{:y Int} -> '{:x Int :y Int}])
(defn g [m]
  (merge m {:x 1}))
```

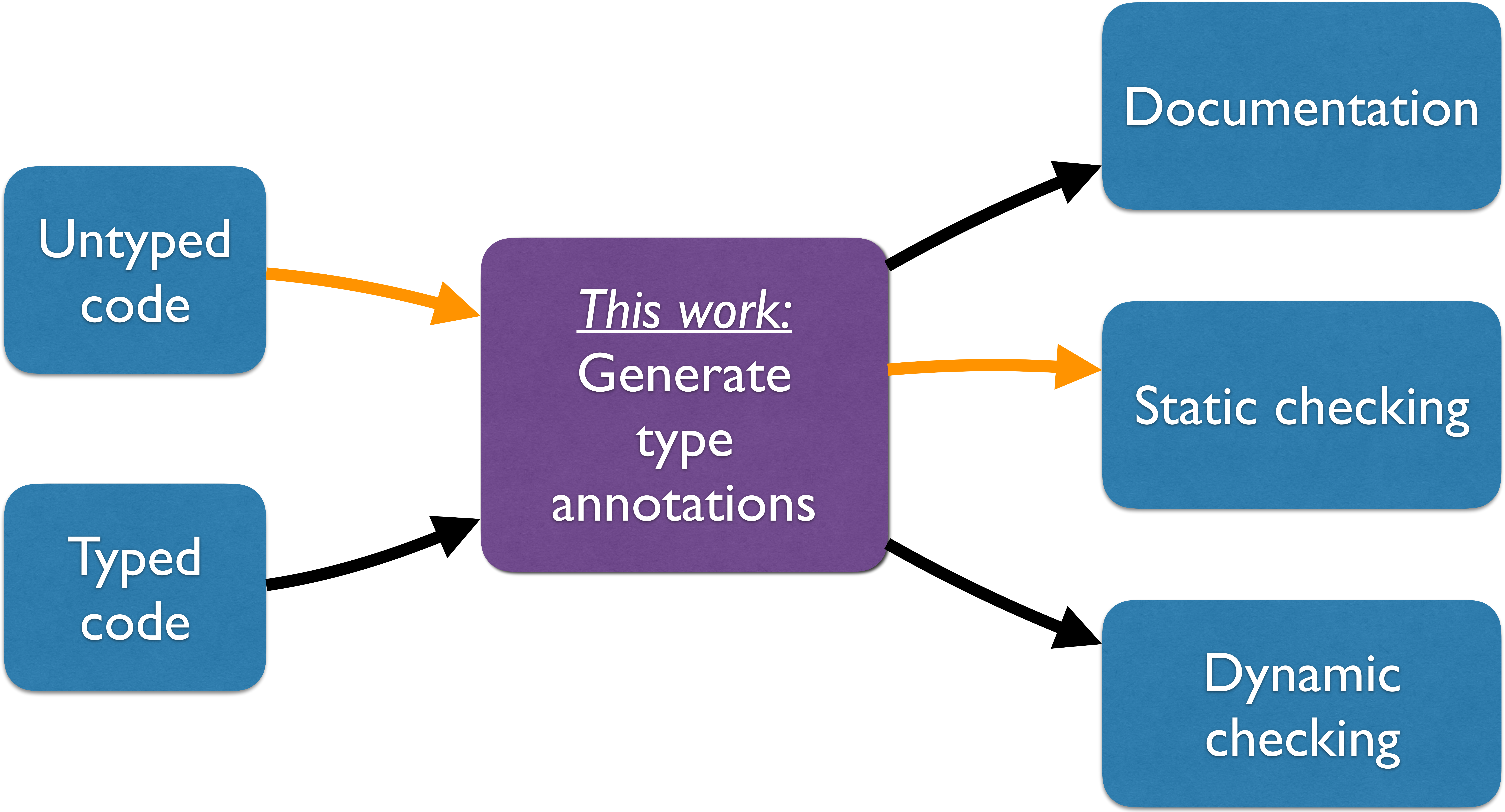
Why?

- Gradual typing
- Unfinished program state
- Help write contracts

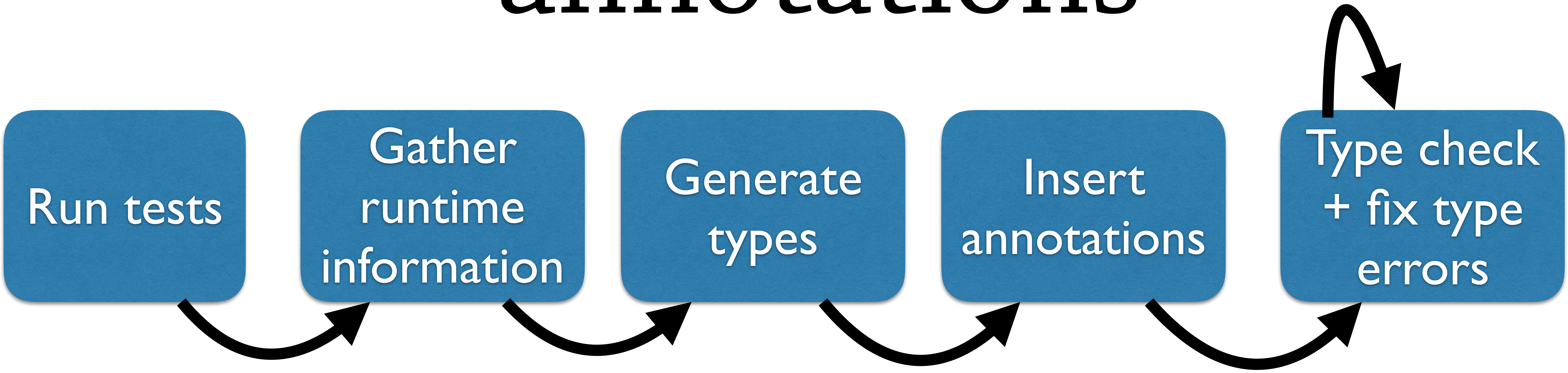
Generate contracts

Assert as contract

```
(ann g ['{:y Int} -> '{:x Int :y Int}])
(defn g [m]
  (merge m {:x 1}))
```


Untyped code
Typed code
This work:
Generate type annotations
Documentation
Static checking
Dynamic checking

Goal:
Mostly correct annotations



Annotations needed

```
(ann point [Long Long -> Point])
(defn point [x y]
  {:x x
   :y y})
```

Top-level typed bindings

Untyped libraries

```
(ann clojure.string/upper-case [Str -> Str])
```

Instrumentation

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]))
```

Library imports

```
str/upper-case
```

```
(track str/upper-case
       ['str/upper-case])
```

Example

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

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

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

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

Inferring
Flat structural types

```clojure
; Int Int -> Point
(defn point [x y]
  {:x x
   :y y})

(deftest point-test
  (is (= 1 (:x (point 1 2))))
  (is (= 2 (:y (point 1 2)))))
```

Track def

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

Wrap definition

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

Tracking functions

```
(track f [path])
```

Wrap and track
domain + range

```
(fn [x]
  (track
    (f (track x [path {:dom 0}]))
    [path :rng]))
```

Tracking *point*

```
; 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])))
```

Track x, y, and return value

Application

```
(point 1 2)
```

```
(track
  ((fn [x y]
     {:x x
      :y y})
   (track 1 ['point {:dom 0}])
   (track 2 ['point {:dom 1}]))
  ['point :rng])
```

point : [Long ? -> ?]

```
(track
  ((fn [x y]
     {:x x
      :y y})
   1  ; ['point {:dom 0}] : Int
   (track 2 ['point {:dom 1}]))
  ['point :rng])
```

```
point : [Long Long -> ?]
```

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

```
point : [Long Long -> ?]
```

```clojure
(track
  {:x 1
   :y 2}
  ['point :rng])
```

```
point : [Long Long -> ?]
```

```
{:x (track 1 ['point :rng (key :x)])
 :y (track 2 ['point :rng (key :y)])}
```

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

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

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

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

Higher-order functions

```clojure
; [A -> B] (List A) -> (List B)
(def my-map map)
```

```clojure
(deftest my-map-test
  (is (= [2 3 4] (my-map inc [1 2 3]))))
```

map : [? ? -> ?]

```
(track
  (map
    (track inc ['my-map {:dom 0}])
    (track [1 2 3] ['my-map {:dom 1}]))
  ['my-map :rng])
```

```
map : [[? -> ?] ? -> ?]
```

```clojure
(track
  (map
    ; ['my-map {:dom 0}] : ? -> ?
    (fn [n]
      (track
        (inc
          (track n ['my-map {:dom 0} {:dom 0}]))
        ['my-map {:dom 0} :rng]))
    (track [1 2 3] ['my-map {:dom 1}]))
  ['my-map :rng])
```

```
map : [[? -> ?] (Seqable Long) -> ?]
```

```
(track
  (map
    ; ['my-map {:dom 0}] : ? -> ?
    (fn [n]
      (track
        (inc
          (track n ['my-map {:dom 0} {:dom 0}]))
        ['my-map {:dom 0} :rng]))
    ; ['my-map {:dom 1} {:index 0}] : Long
    ; ['my-map {:dom 1} {:index 1}] : Long
    ; ['my-map {:dom 1} {:index 2}] : Long
    [1 2 3])
  ['my-map :rng])
```

```
map : [[Long -> Long] (Seqable Long) -> ?]
```

Side effects of *map* ...

```
; ['my-map {:dom 0} {:dom 0}] : Long
; ['my-map {:dom 0} :rng] : Long
; ['my-map {:dom 0} {:dom 0}] : Long
; ['my-map {:dom 0} :rng] : Long
; ['my-map {:dom 0} {:dom 0}] : Long
; ['my-map {:dom 0} :rng] : Long
(track
  [2 3 4]
  ['my-map :rng])
```

```
map : [[Long -> Long] (Seqable Long) -> (Seqable Long)]
```

```
; ['my-map :rng {:index 0}] : Long
; ['my-map :rng {:index 1}] : Long
; ['my-map :rng {:index 2}] : Long
[2 3 4]
```

Recursive HMaps

```
;;  t  ::= [x : t -> t] | (not t) | (or t t) | (and t t) | #f | N | Any
;;  p  ::= (is e t) | (not p) | (or p p) | (and p p) | (= e e)
```

```
; P -> Any
(defn unparse-prop [p]
  {:pre [(contains? p :P)]}
  (case (:P p)
    :is `(~'is ~(unparse-exp (:exp p))
               ~(unparse-type (:type p)))
    := `(~'= ~@(map unparse-exp (:exps p)))
    :or `(~'or ~@(map unparse-prop (:ps p)))
    :and `(~'and ~@(map unparse-prop (:ps p)))
    :not `(~'not ~(unparse-prop (:p p)))))
```

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

How to compact?

Heuristic: Group by common dispatch entry

```
(defalias P
  "Propositions"
  (U '{:P ':is  ...}
     '{:P ':=   ...}
     '{:P ':or  ...}
     '{:P ':and ...}
     '{:P ':not ...}))
```

```
(case (:P p)
  :is  ...
  :=   ...
  :or  ...
  :and ...
  :not ...)
```

Heuristic: Merge by keyset

```
{:car Bool :cdr Bool}   {:car Int :cdr Int}

{:car (U Int Bool) :cdr (U Int Bool)}
```

Heuristic: Ignore contravariance

```
[Int -> Int]                    [Bool -> Bool]

    [(U Int Bool) -> (U Int Bool)]
```

Is this sound?

Example

Test data

```
{:a nil}
{:a {:a nil}}
{:a {:a {:a nil}}}
```

Final Type

```
(defalias As
  '{:a (U As nil)})
```

Naive join

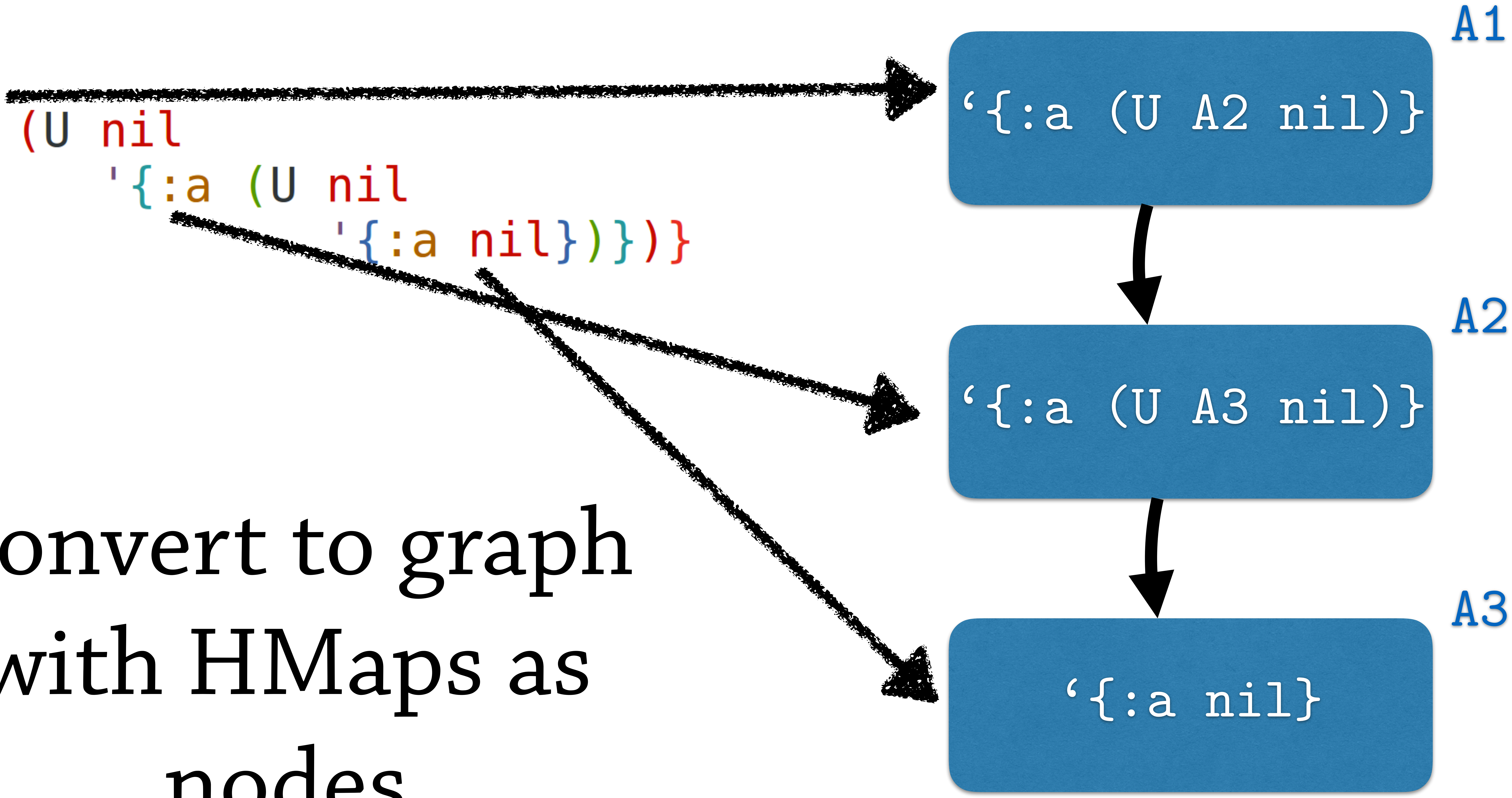
Test data

```
{:a nil}
{:a {:a nil}}
{:a {:a {:a nil}}}
```

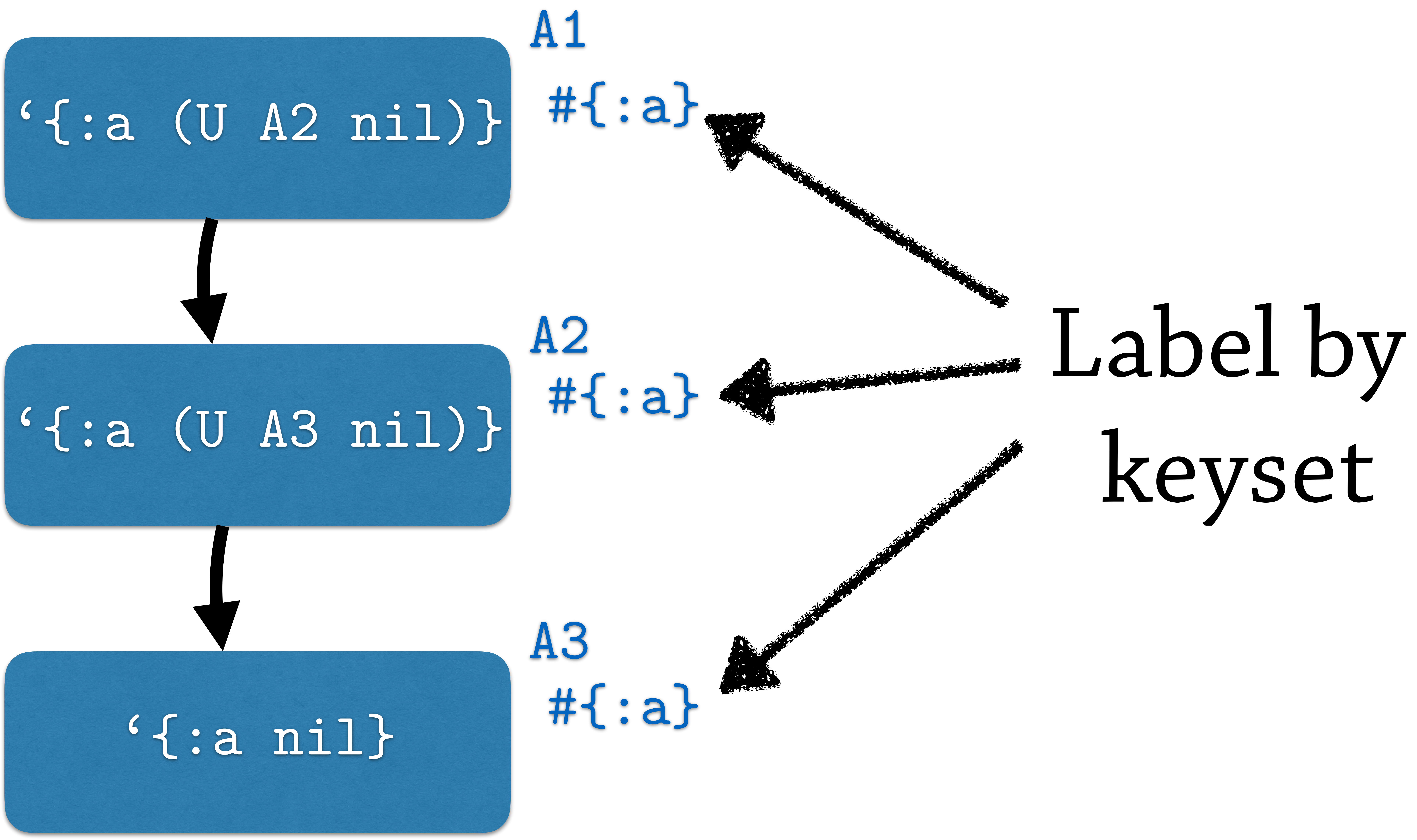
Naive type

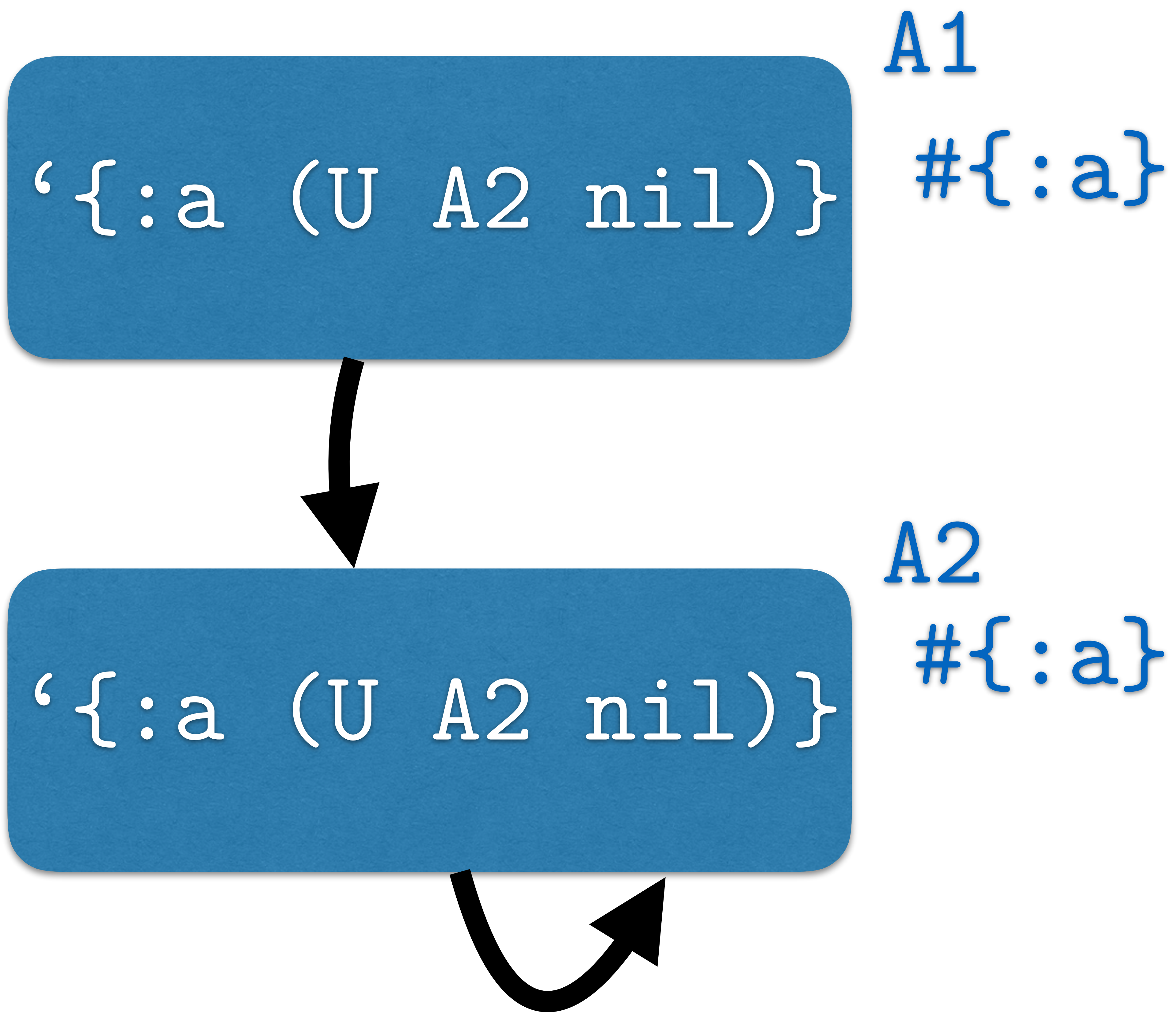
```
'{:a (U nil
        '{:a (U nil
                '{:a nil})})}
```

```
'{:a (U nil
      '{:a (U nil
            '{:a nil})})}
```

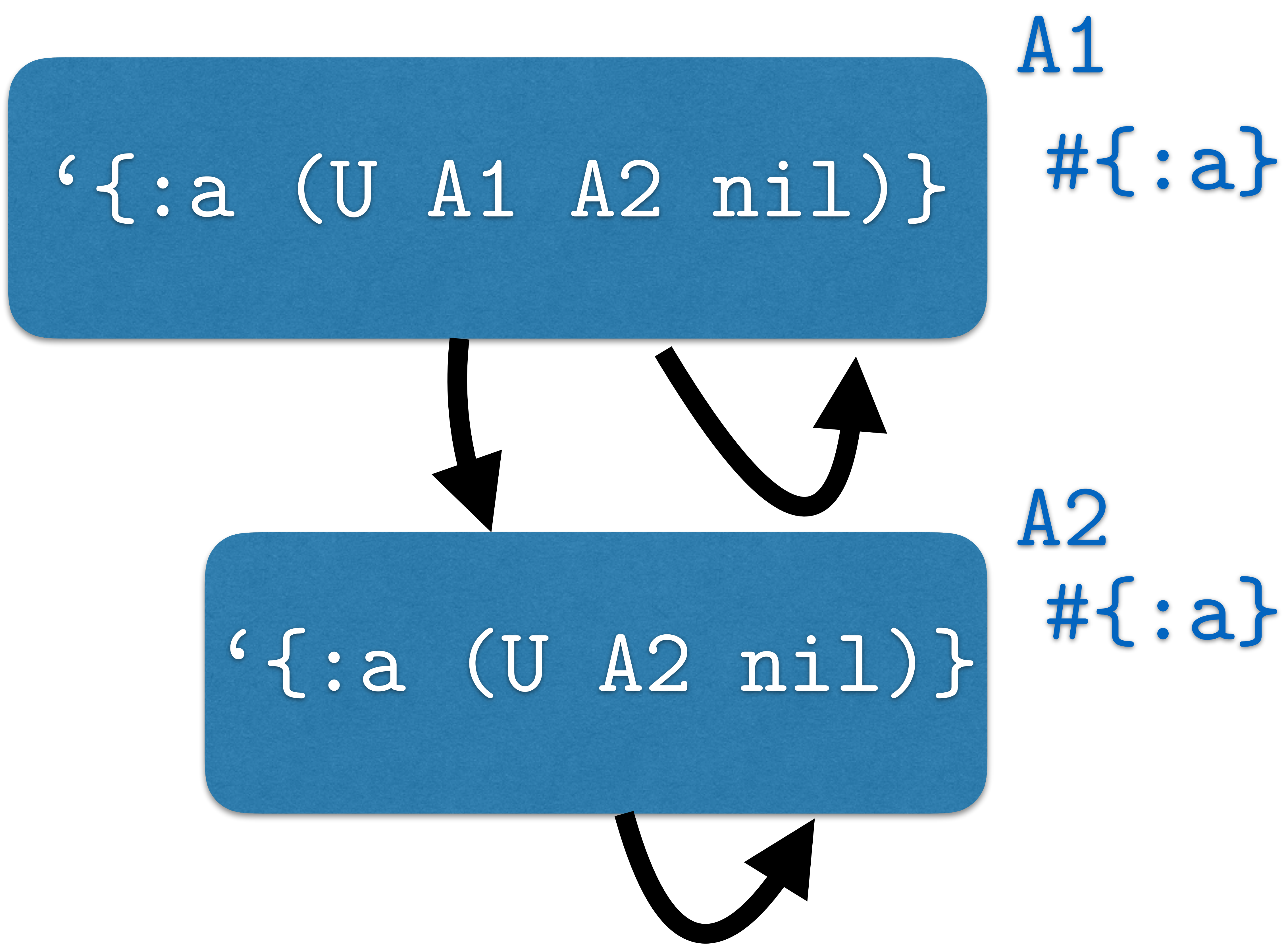


Convert to graph with HMaps as nodes

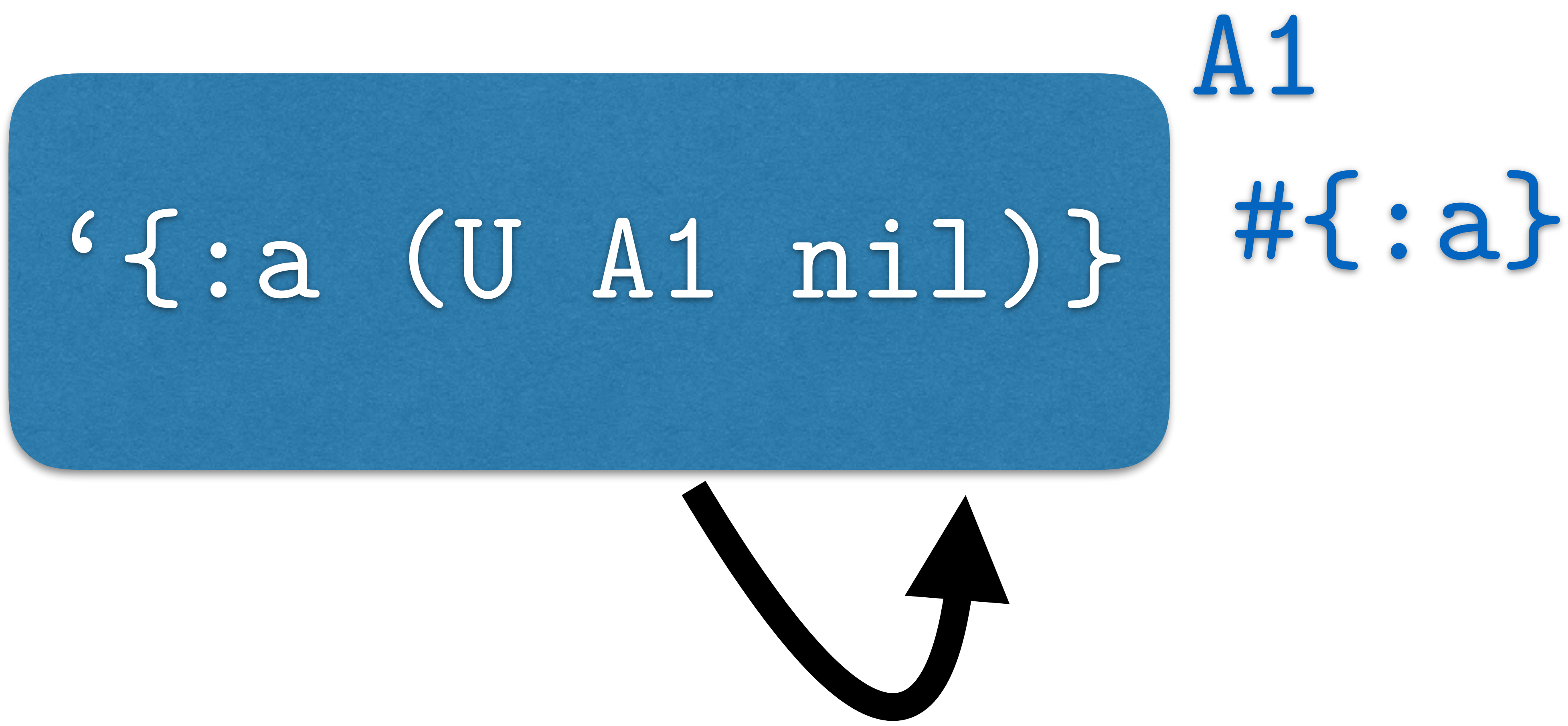

A1
‘{:a (U A2 nil)}
#{:a}
A2
‘{:a (U A3 nil)}
#{:a}
A3
‘{:a nil}
#{:a}
Label by keyset



Merge nodes on keyset



Merge nodes on keyset



Merge nodes on keyset

Future work:
Polymorphism

Idea: Associate hashes with known paths

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

[point (dom 0)]

[point :rng (key :x)]

[point (dom 1)]

[point :rng (key :y)]

Future work

- Implementation
  - Performance?
- Evaluation
  - Are annotations “good enough” in practice?

Inferring

Structural Types from Tests

Inferring

Recursive

Structural

Types from Tests

Inferring *Polymorphic* *Recursive* Structural Types from Tests

Thanks!

https://github.com/clojure/core.typed

@ambrosebs

Ambrose Bonnaire-Sergeant