Typed Clojure: Wishful Thinking
Ambrose Bonnaire-Sergeant
This talk

• Quick intro to Typed Clojure
• List of challenges/solutions to improve Typed Clojure
  • Barriers to Entry
  • Annotation Burden
  • Strictness
  • ClojureScript
• Hopefully some discussion
What is Typed Clojure?

• Optional type system for Clojure
  • Write expected types for your program, checker will validate
• Static analysis
  • Checks your program without running it
(defalias Point
  "A point with x-y coordinates"
  '{:x Int :y Int})

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

(ann add-xy [Point -> Int])
(defn add-xy [{:keys [x y] :as p}]
  (+ x y))
True unions + Flow typing

(ann maybe-add-xy [(U nil Point) -> (U nil Int)])
(defn maybe-add-xy [{:keys [x y] :as p}]
  (when p
    (+ x y))) ; x : Int, y : Int
Expands before checking

(defmacro my-when [& body] `(when ~@body))

(ann maybe-add-xy [(U nil Point) -> (U nil Int)])
(defn maybe-add-xy [{:keys [x y] :as p}]
  (my-when p
    (+ x y)))
Challenges
Part 1: Barriers to entry

- initialization time in production
- bad error messages
- lack of library annotations
Challenge

Don’t want to increase initialization time for type checked libraries

- collecting annotations
- expanding/defining wrapper macros
Solution

Delay loading annotations

(ns foo ...) ; lazily load ann
(t/register-ns!
  'foo.annotations)

(def f 1)

(ns foo.annotations (:require [...]))

(t/ann f Int)
(t/ann g Int)

...
Challenge

No source of type annotations for libraries

- libraries don’t provide their own types
- no central place for annotations like DefinitelyTyped for TypeScript
Solutions

- Start a suite of annotations under typedclojure GitHub org
- reuse specs as a type annotations
  - unfortunately, specs don’t often make good types
    - no polymorphism
    - how to translate semantics to types? (eg. fspec, every)
    - s/keys’s implicit optional entries
  - can we retrofit these specs to be more useful as types?
- Guidelines for how to add type annotations to your own libraries
Challenge

Error messages from macro expansions point to code the user didn’t write

\[(\text{inc (when foo 1)})\]

Type error: Expected Number, found nil
in: \text{nil}
in: \text{(if foo 1 nil)}
Solution

Custom typing rules

The custom rule:

```clojure
(if foo
  1
  (with-blame {:form '(when foo 1)
               :msg "Else branch of `when` expected nil"}
    nil))
```

Type error: Expected Number, found nil

Message: Else branch of `when` expected nil

in: (when foo 1)
Part 2: Annotation burden

- too many `fn` annotations

- brittle polymorphic inference

- need “wrapper” macros to help check complex expansions

- these macros need their own annotations
Challenge

Need to annotate “obvious” function arguments

(let [f (fn [x :- Int] (inc x))]
  (f 1))
Solution

Delay type checking `fn` body until called

(let [f #(inc %)]
  ; f : (Lambda #(inc %))
  (f 1)) ; checking happens here

Caveats:
- Need to handle infinite recursion (eg. checking y-combinator)
- Can we avoid redundantly re-checking body of `fn`?
Challenge

Need to annotate polymorphic higher-order function arguments

(map (fn [a :- Int] ...) [1 2 3])
Solution: Smarter inference

Deduce an optimal “ordering” for checking arguments

\[(\text{All } [a \ b]) \left[ [a \rightarrow b] \ (\text{Seqable } a) \rightarrow (\text{Seqable } b) \right] \]

1. Check collection first
2. Use collection type to seed function argument
3. Now we have the return argument type
4. Which travels to the return of the entire function
Type checking comp

(ann f [Number -> Number])
(def f (comp #(inc %) #(dec %)))
Type checking comp

(ann f [Number -> Number])
(def f (comp #(inc %) #(dec %)))

; Type of `comp`

(All [a b c]
  [[b -> c] [a -> b] -> [a -> c]])
Type checking map transducer

(ann f (Transducer [Num -> Num]))
(def f (map #(inc %)))

; Type of the `map` transducer

(All [a b]
  [[[a -> b] -> (All [r] [[r b -> r] -> [r a -> r]]))])
Scale to comp+transducers

; Call (sequence (comp (map #(inc %))) (map #(dec %))) [1 2 3])

; (map #(inc %))
(All [a b]
  [[a -> b] ->
   (All [r] [[r b -> r] -> [r a -> r]]))

; (map #(dec %))
(All [a b]
  [[a -> b] ->
   (All [r] [[r b -> r] -> [r a -> r]]))

; (comp ..)
(All [a b c]
  [[b -> c] [a -> b] -> [a -> c]])

; (sequence ..)
(All [a b]
  ((All [r] [[r b -> r] -> [r a -> r]]) (Seqable a) -> (Seqable b)))
Challenge

Need to write typed wrappers for macros with complex expansions

(require 'clojure.core.typed.async :as ta)
(ta/go
  (when foo
    1))
Solution

Support custom rules for macros that don’t require expansion, then expand them after type checking

```
(go
  (when foo 1))
```

Extract

```
(when foo 1)
```

Expand & check

```
(if foo 1 nil)
```

Reinsert

```
(go (if foo 1 nil))
```
Challenge

Need to write local annotations for wrapper macros

(t/doseq [a :- Int, [1 2 3]] :- Int ...
...)}
Solution

Write custom typing rules to direct inference.

(t/doseq [a :- Int, [1 2 3]] :- Int ...)
Part 3: Strictness

- stricter map operations
- opt-in unsoundness
Challenge

Map ops don’t catch enough type errors

**Assoc wrong key**

(ann m1 (HMap :optional {:foo Int}))
(def m1 (assoc {} :foob 1))

**Get wrong key**

(ann v (U nil Int))
(def v (get {:exists 1} :non-existent-key))
Solutions (?)

More restrictive subtyping for HMap’s

**Assoc wrong key**

```
(ann m1 (HMap :optional {:foo Int}))
(def m1 (assoc {} :foob 1))
```

Error: unknown key :foob

**Get wrong key**

```
(ann v (U nil Int))
(def v (get {:exists 1} :non-existent-key))
```
Challenge

Typed Clojure is too strict with unannotated code

(defn my-fn [...] (let [a (lib1 ...)
                          b (lib2 ...)
                          c (lib3 ...)]
                    (lib4 ...)))
Solution

Opt-out of soundness—closer to TypeScript when needed

```clojure
(check-ns 'my-ns
 :check-config {:check-ns-dep :never
 :unannotated-def :unchecked
 :unannotated-var :unchecked
 :unannotated-arg :unchecked})
```
Part 4: ClojureScript

- analyzer that supports partial analysis
- undefined/nil
- Closure/TypeScript annotations
Challenge

Analyzer that can partially expand code (does not exist yet)

(go (when foo 1))

(go (if foo 1 nil))
Challenge

ClojureScript mostly treats nil/undefined as equivalent

(defalias Nilable
  (TFn [x] (U nil x)))

(defalias Nilable
  (TFn [x] (U undefined nil x)))
More problems...

Solution: Introduce new base types js/Null and js/Undefined

But now…:
- Is nil == js/Null?
- Is js/Undefined <: nil? since (nil? js/undefined) => true
  - Either choice has interesting consequences
Challenge

How to use Closure/TypeScript annotations to our advantage?
What is Typed Clojure good at?

- Flow typing
- Checking higher-order idioms (channels, functions, atoms)
- Specifying polymorphic functions
Problems with Typed Clojure

- Insufficient local type inference
- Large annotation burden
- Slow (checking speed & dev iterations)
- Macro usages hard to check
- Sometimes too strict, sometimes too loose
Possible Solutions

• custom typing rules
  • better inference, error messages
• “directed” local type inference
• more flexible checking
Thanks