Typed Clojure in Practice

Ambrose Bonnaire-Sergeant
Indiana University
@ambrosebs

Let’s write some Clojure code

… a hashing function

Here's how you call it

```
(summarise [1 2 3]) ;=> <Int>
(summarise nil)     ;=> <Int>
```

This is invalid input

```
(summarise [])
```

```
(summarise ())
```

The Type is roughly this

```
;; nil or (NonEmptyColl Int) -> Int
```

Untyped prototype

```
(ns sum)


(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
=> (summarise nil)
;=> 42
```

```
=> (summarise nil)
;=> 42

=> (summarise [42 33 32])
```

```
=> (summarise nil)
;=> 42

=> (summarise [42 33 32])
java.lang.StackOverflowError
    at java.lang.Number<init>(Number.java:49)
    at java.lang.Long<init>(Long.java:684)
    at java.lang.Long.valueOf(Long.java:577
    at stl2014.sum$summarise.invoke(sum.clj:6)
    at stl2014.sum$summarise.invoke(sum.clj:6)
    at stl2014.sum$summarise.invoke(sum.clj:6)
    at stl2014.sum$summarise.invoke(sum.clj:6)
    at stl2014.sum$summarise.invoke(sum.clj:6)
    ...
```

Step 1: Require core.typed

```
(ns sum)


(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

Step 1: Require core.typed

```
(ns sum
  (:require [clojure.core.typed :as t]))

(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

Step 2: Annotate vars

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

Step 2: Annotate vars

```
(ann summarise


(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

Step 2: Annotate vars

```
(ann summarise
 (IFn [(U nil (NonEmptyColl Int)) -> Int]

(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

Step 2: Annotate vars

```
(ann summarise
 (IFn [(U nil (NonEmptyColl Int)) -> Int]
      [(U nil (NonEmptyColl Int)) Int -> Int])
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

Aliases for readability

```
(ann summarise
 (IFn [(U nil (NonEmptyColl Int)) -> Int]
      [(U nil (NonEmptyColl Int)) Int -> Int])
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

Aliases for readability

```
(ann summarise
 (IFn [(U nil (NonEmptyColl Int)) -> Int]
      [(U nil (NonEmptyColl Int)) Int -> Int]))


(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
```

Aliases for readability

```
(ann summarise
 (IFn [(U nil (NonEmptyColl Int)) -> Int]
      [(U nil (NonEmptyColl Int)) Int -> Int]))


(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
```

Aliases for readability

```
(defalias NInts
  ‘‘nil or persistent non-empty coll of ints’’
  (U nil (NonEmptyColl Int)))

(ann summarise
 (IFn [                          -> Int]
      [                     Int -> Int]))

(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
```

Aliases for readability

```
(defalias NInts
  ‘‘nil or persistent non-empty coll of ints’’
  (U nil (NonEmptyColl Int)))

(ann summarise
 (IFn [                          -> Int]
      [                     Int -> Int]))

(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
```

Aliases for readability

```
(defalias NInts
  "nil or persistent non-empty coll of ints"
  (U nil (NonEmptyColl Int)))

(ann summarise
 (IFn [NInts -> Int]
      [NInts Int -> Int]))


(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
```

Aliases for readability

```
(defalias NInts
  "nil or persistent non-empty coll of ints"
  (U nil (NonEmptyColl Int)))

(ann summarise
 (IFn [NInts -> Int]
      [NInts Int -> Int]))

(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
=> (t/check-ns)

Type mismatch:

Expected:
    NInts

Actual:
    (Seq Int)

in: (rest nseq)
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(rest [1])
;=> ()
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(rest [1])
;=> ()
```

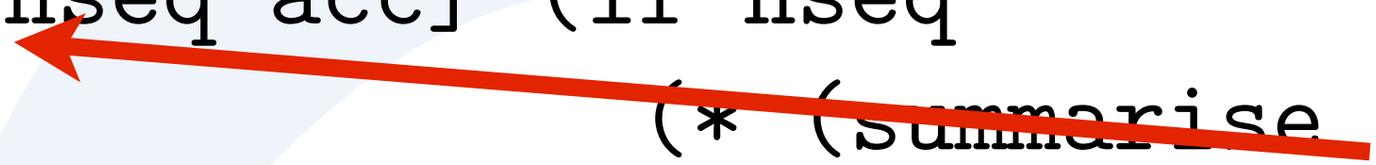
```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(rest [1])
;=> ()
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(rest [1])
;=> ()
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```



```
(rest [1])
;=> ()

(rest ())
;=> ()
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(rest [1])
;=> ()

(rest ())
;=> ()
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(rest [1])
;=> ()

(rest ())
;=> ()

(rest ())
;=> ()
```

rest 1.0 Source Usages

```
(rest coll)
(∀ [x]
   (λ [(U nil (Seqable x)) → (ASeq x)]))
```

Returns a possibly empty seq of the items after the first. Calls seq on its argument.

```
(rest [1 2])
;=> (2)

(rest nil)
;=> ()
```

```
(rest [])
;=> ()
```

next 1.0 Source Usages

```
(next coll)
(∀ [x]
   (λ [(U nil (Seqable x)) → (U nil (NonEmptyASeq x))]))
```

Returns a seq of the items after the first. Calls seq on its argument. If there are no more items, returns nil.

```
(next [1 2])
;=> (2)

(next nil)
;=> nil
```

```
(next [])
;=> nil
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (rest nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (     nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (     nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (next nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (next nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (next nseq)
                              (inc acc))
                   (first nseq))
                42)))
=> (t/check-ns)
:ok
```

```
 => (summarise [42 33 32])
;=> 1280664

 => (summarise nil)
;=> 42
```

```
 => (summarise [42 33 32])
;=> 1280664

 => (summarise nil)
;=> 42

 => (summarise [])
```

```
=> (summarise [42 33 32])
;=> 1280664

=> (summarise nil)
;=> 42

=> (summarise [])
```

```
NullPointerException
clojure.lang.Numbers.ops (Numbers.java:961)
clojure.lang.Numbers.multiply (Numbers.java
stl2014.sum/summarise (form-init6981015802
stl2014.sum/summarise (form-init6981015802
stl2014.sum/eval3757 (form-init69810158023
clojure.lang.Compiler.eval (Compiler.java:6
...
```

Update the annotation

```
(defalias NInts
  "nil or a non-empty persistent collection of integers"
  (U nil (NonEmptyColl Int)))
```

```
(ann summarise (IFn [NInts -> Int]
                    [NInts Int -> Int]))
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (next nseq) (inc acc))
                   (first nseq))
                42)))
```

Update the annotation

```
(defalias NInts
  "nil or a persistent collection of integers"
  (U nil (Coll Int)))

(ann summarise (IFn [NInts -> Int]
                    [NInts Int -> Int]))
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (next nseq) (inc acc))
                   (first nseq))
                42)))
```

Update the annotation

```
(defalias NInts
  "nil or a persistent collection of integers"
  (U nil (Coll Int)))

(ann summarise (IFn [NInts -> Int]
                    [NInts Int -> Int]))
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (next nseq) (inc acc))
                   (first nseq))
                42)))
```

```
(ann summarise (IFn [NInts -> Int]
                    [NInts Int -> Int]))
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (next nseq) (inc acc))
                   (first nseq))
                42)))
```

Check with new type

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (next nseq) (inc acc))
                   (first nseq))
                42)))
```

Check with new type

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (next nseq) (inc acc))
                   (first nseq))
                42)))
```

```
=> (t/check-ns)

                Type mismatch:

                Expected:
                                  Number
                Actual:
                                  (U nil Int)

                in: (first nseq)
```

Need a better test

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (next nseq) (inc acc))
                   (first nseq))
                42)))
```

```
=> (t/check-ns)
```

```
Type mismatch:

Expected:
                    Number
Actual:
                    (U nil Int)

in: (first nseq)
```

Need a better test

```
(defn summarise
  ([nseq] (summarise nseq 0))
  ([[] acc] (if []
              (* (summarise (next []) (inc acc))
                 (first []))
              42)))
```

```
=> (t/check-ns)

                Type mismatch:

                Expected:
                                  Number
                Actual:
                                  (U nil Int)

                in: (first nseq)
```

seq 1.0 Source Usages

```
(seq coll)
(∀ [x]
   (λ [(U nil (Coll x)) → (U nil (NonEmptySeq x))}]))
```

Returns a seq on the collection. If the collection is
empty, returns nil. (seq nil) returns nil. seq also works on
Strings, native Java arrays (of reference types) and any objects
that implement Iterable.

```
(seq [1 2])
;=> (1 2)
```

```
(seq nil)
;=> nil
```

```
(seq [])
;=> nil
```

Update implementation

```
(ann summarise (IFn [NInts -> Int]
                    [NInts Int -> Int]))
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if nseq
                (* (summarise (next nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

Update implementation

```
(ann summarise (IFn [NInts -> Int]
                    [NInts Int -> Int]))
(defn summarise
  ([nseq] (summarise nseq 0))
  ([nseq acc] (if (seq nseq)
                (* (summarise (next nseq)
                              (inc acc))
                   (first nseq))
                42)))
```

```
=> (t/check-ns)
```

```
:ok
```

```
=> (summarise [42 33 32])
;=> 1280664

=> (summarise nil)
;=> 42

=> (summarise [])
;=> 42
```

Typed Clojure

is an

Optional Type System

that

catches type errors

in

real Clojure code

Where have we come from?

2011


simplicity


Myanmar (Burma)
Laos
Thailand
Vietnam
Cambodia
Hong Kong
Macau
South China Sea
Kalusunan
Catanduanes
Philippines
Kabisayaan
Palawan
Mindanao
Basilan
Celebes Sea
Bay of Bengal
Andaman Sea
Gulf of Thailand
Arabian Sea
Sri Lanka
Laccadive Sea
Malaysia
Singapore
Java Sea
Indonesia
Banda Sea
Bismarck Sea
Papua New Guinea
Arafura Sea
Solomon Sea
Timor Sea
Perth WA
Australia
Directions
Search nearby
Save to map
more
Indian Ocean
Australia
Coral Sea
Great Australian Bight
Tasman Sea
A

A Practical Optional Type System for Clojure

Ambrose Bonnaire-Sergeant

Supervised by Rowan Davies

The University of Western Australia

2012

CHAPTER 1

Introduction

THE UNIVERSITY OF WESTERN AUSTRALIA

1.1 Thesis

It is practical and useful to design and implement an optional typing system for the Clojure programming language using bidirectional checking that allows Clojure programmers to continue using idioms and style found in current Clojure code.

1.2 Motivation

frenchy64 13 days ago Bump readme version

1 contributor

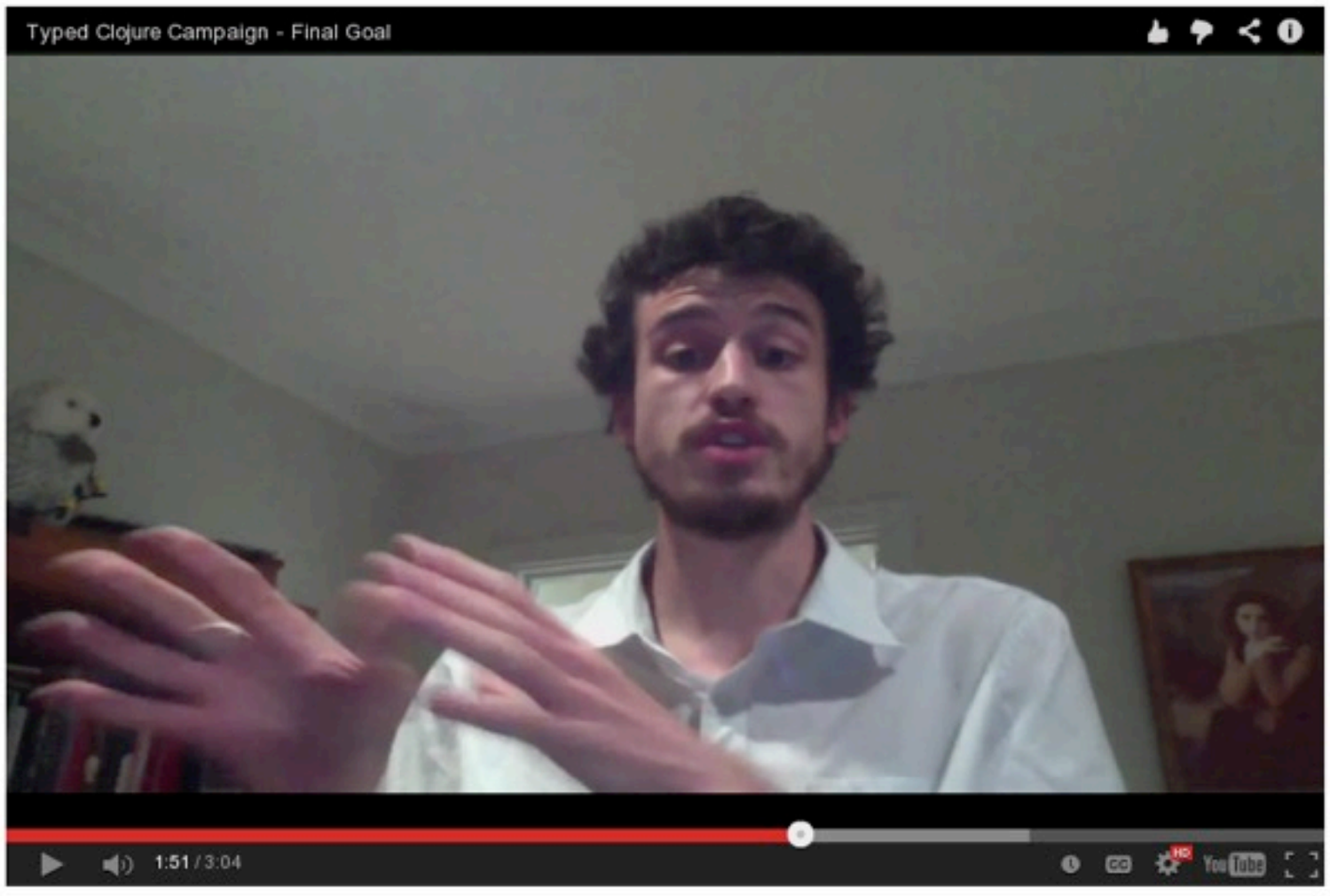
core.typed

Gradual typing in Clojure, as a library.

Releases and Dependency Information


Clojure
conj
Sheraton
Raleigh

Typed Clojure



Help build optional type systems for Clojure and Clojurescript.

$35,254 USD
RAISED OF $20,000 GOAL

176%
0 time left

This campaign started on Sep 27 and closed on November 11, 2013 (11:59pm PT).

Flexible Funding

CAMPAIGN CLOSED
This campaign ended on November 11, 2013

SELECT A PERK

$5 USD

Typed Clojure Hangouts

Prismatic

sonian

Cursive Clojure

circleci

cognitect

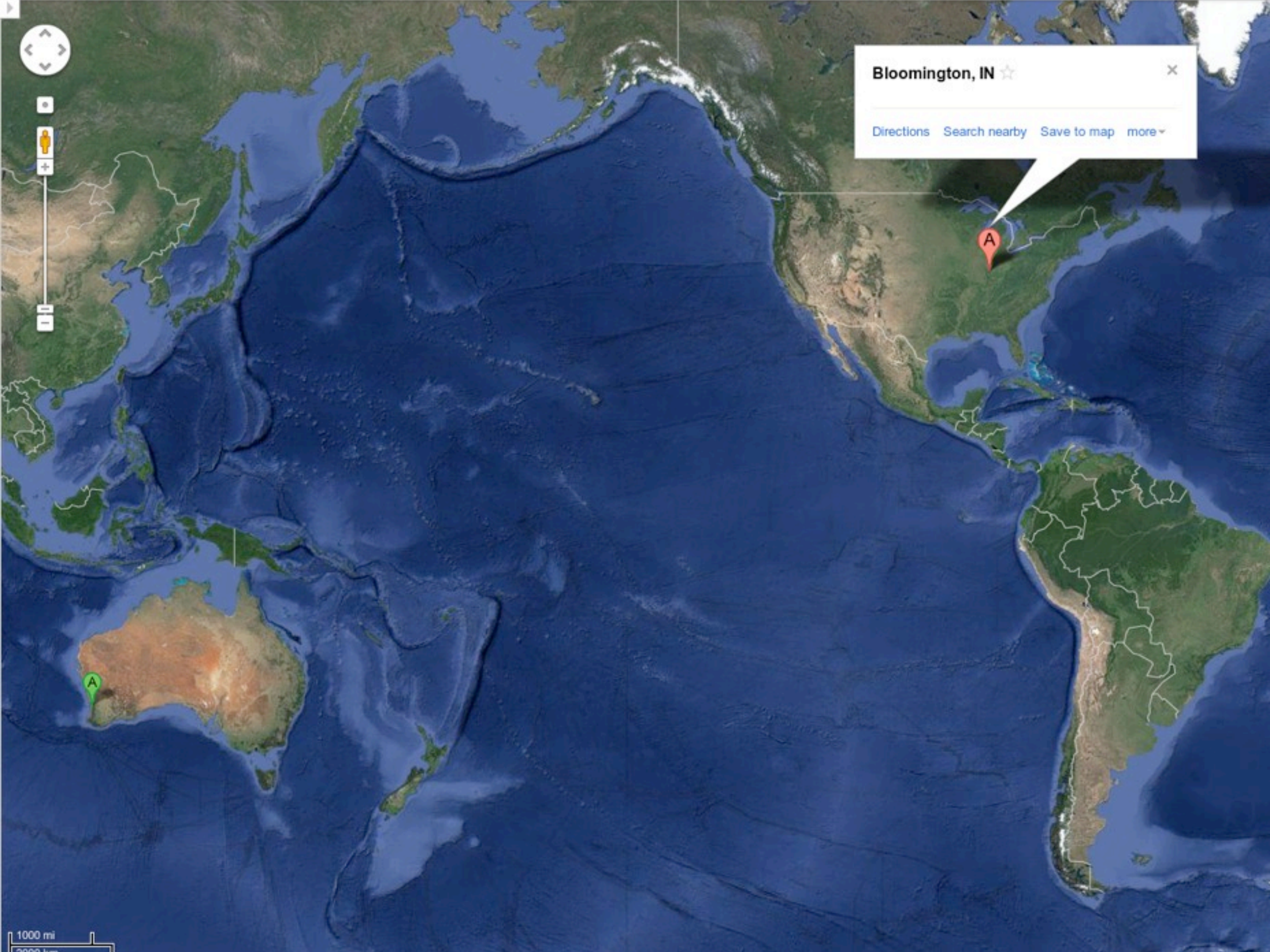
Hacker School

SNOWPLOW

Leonidas

BRICK ALLOY

THORTECH SOLUTIONS


Bloomington, IN
Directions
Search nearby
Save to map
more
A
A
1000 mi

INDIANA UNIVERSITY

Ambrose
Bonnaire-Sergeant

Sam
Tobin-Hochstadt

2014

Where are we now?

stl2014 › src › stl2014 › cursive.clj

Project

stl2014 (/nfs/nfs4/home/a
- .idea
- doc
- resources
- src
  - stl2014
    - async.clj
    - core.clj
    - cursive.cl
    - encrypt.c
    - java.clj
    - keypair.cl
    - keypair_u
- test
- .gitignore
- .nrepl-port
- LICENSE
- project.clj
- README.mc
- stl2014.iml

.gitignore
.nrepl-port
LICENSE
project.clj
README.mc
stl2014.iml
External Libraries

stl2014.cursive

```
(ns stl2014.cursive
  (:refer-clojure :exclude [defn])
  (:require [clojure.core.typed :refer [U Int defn]]))

(defn pos-or-neg? [n :- Int]
  (or (pos? n) (neg? n)))

(defn nonzero-or-nil? [n :- (U nil Int)]
  (or (pos-or-neg? n) (nil? n)))
```

```
Function pos-or-neg? could not be applied to arguments:


Domains:
        Int

Arguments:
        (U Int nil)

Ranges:
        boolean

in: (pos-or-neg? n)
```

REPL: Local: stl2014.core | Local: user | Local: stl2014.keypair

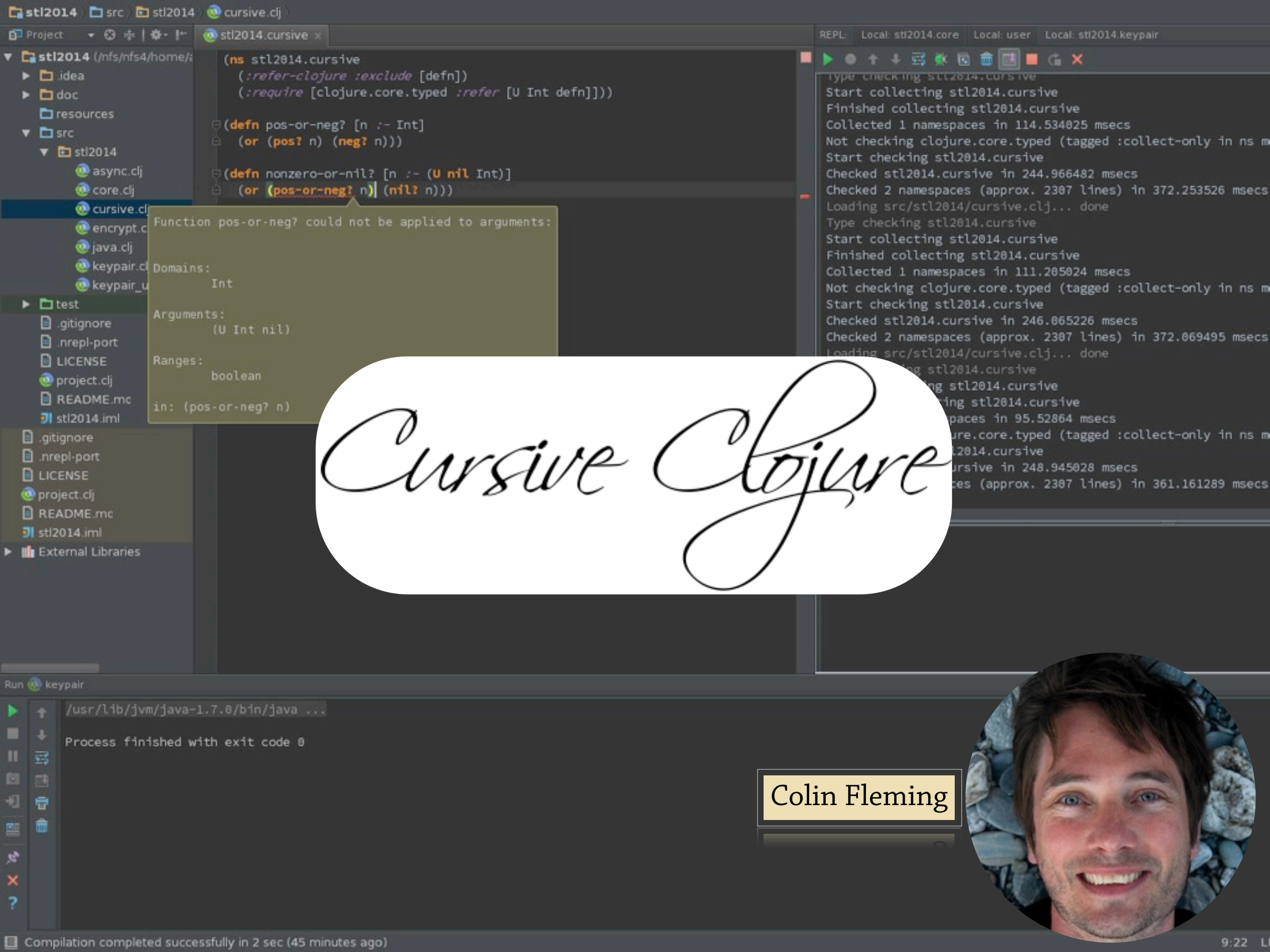
```
Type checking stl2014.cursive
Start collecting stl2014.cursive
Finished collecting stl2014.cursive
Collected 1 namespaces in 114.534025 msecs
Not checking clojure.core.typed (tagged :collect-only in ns m
Start checking stl2014.cursive
Checked stl2014.cursive in 244.966482 msecs
Checked 2 namespaces (approx. 2307 lines) in 372.253526 msecs
Loading src/stl2014/cursive.clj... done
Type checking stl2014.cursive
Start collecting stl2014.cursive
Finished collecting stl2014.cursive
Collected 1 namespaces in 111.205024 msecs
Not checking clojure.core.typed (tagged :collect-only in ns m
Start checking stl2014.cursive
Checked stl2014.cursive in 246.065226 msecs
Checked 2 namespaces (approx. 2307 lines) in 372.069495 msecs
Loading src/stl2014/cursive.clj... done
Type checking stl2014.cursive
Start collecting stl2014.cursive
Finished collecting stl2014.cursive
Collected 1 namespaces in 95.52864 msecs
Not checking clojure.core.typed (tagged :collect-only in ns m
Start checking stl2014.cursive
Checked stl2014.cursive in 248.945028 msecs
Checked 2 namespaces (approx. 2307 lines) in 361.161289 msecs
```

Run keypair

```
/usr/lib/jvm/java-1.7.0/bin/java ...

Process finished with exit code 0
```

Compilation completed successfully in 2 sec (45 minutes ago)

9:22


stl2014 src stl2014 cursive.clj
Project
stl2014 (/nfs/nfs4/home/a
.idea
doc
resources
src
stl2014
async.clj
core.clj
cursive.cl
encrypt.c
java.clj
keypair.cl
keypair_u
test
.gitignore
.nrepl-port
LICENSE
project.clj
README.mc
stl2014.iml
.gitignore
.nrepl-port
LICENSE
project.clj
README.mc
stl2014.iml
External Libraries
stl2014.cursive
(ns stl2014.cursive
(:refer-clojure :exclude [defn])
(:require [clojure.core.typed :refer [U Int defn]]))
(defn pos-or-neg? [n :- Int]
(or (pos? n) (neg? n)))
(defn nonzero-or-nil? [n :- (U nil Int)]
(or (pos-or-neg? n) (nil? n)))
Function pos-or-neg? could not be applied to arguments:
Domains:
Int
Arguments:
(U Int nil)
Ranges:
boolean
in: (pos-or-neg? n)
REPL:
Local: stl2014.core
Local: user
Local: stl2014.keypair
Start collecting stl2014.cursive
Finished collecting stl2014.cursive
Collected 1 namespaces in 114.534025 msecs
Not checking clojure.core.typed (tagged :collect-only in ns m
Start checking stl2014.cursive
Checked stl2014.cursive in 244.966482 msecs
Checked 2 namespaces (approx. 2307 lines) in 372.253526 msecs
Loading src/stl2014/cursive.clj... done
Type checking stl2014.cursive
Start collecting stl2014.cursive
Finished collecting stl2014.cursive
Collected 1 namespaces in 111.205024 msecs
Not checking clojure.core.typed (tagged :collect-only in ns m
Start checking stl2014.cursive
Checked stl2014.cursive in 246.065226 msecs
Checked 2 namespaces (approx. 2307 lines) in 372.069495 msecs
Loading src/stl2014/cursive.clj... done
Cursive Clojure
Run keypair
/usr/lib/jvm/java-1.7.0/bin/java ...
Process finished with exit code 0
Colin Fleming
Compilation completed successfully in 2 sec (45 minutes ago)
9:22

```clojure
(ns stl2014.cursive
  (:refer-clojure :exclude [defn])
  (:require [clojure.core.typed :refer [U Int defn]]))

(defn pos-or-neg? [n :- Int]
  (or (pos? n) (neg? n)))

(defn nonzero-or-nil? [n :- (U nil Int)]
  (or (pos-or-neg? n) (nil? n)))
```

1 errors type

```
Function pos-or-neg? could not be applied to arguments:

Domains:
        Int

Arguments:
        (U Int nil)

Ranges:
        boolean

in: (pos-or-neg? n)
```

Cursive Clojure

```clojure
(ns stl2014.cursive
  (:refer-clojure :exclude [defn])
  (:require [clojure.core.typed :refer [U Int defn]]))

(defn pos-or-neg? [n :- Int]
  (or (pos? n) (neg? n)))

(defn nonzero-or-nil? [n :- (U nil Int)]
  (or (pos-or-neg? n) (nil? n)))
```

```
Function pos-or-neg? could not be applied to arguments:

Domains:
        Int

Arguments:
        (U Int nil)

Ranges:
        boolean

in: (pos-or-neg? n)
```

1 errors type

Function application error

Cursive Clojure

```clojure
(ns stl2014.cursive
  (:refer-clojure :exclude [defn])
  (:require [clojure.core.typed :refer [U Int defn]]))

(defn pos-or-neg? [n :- Int]
  (or (pos? n) (neg? n)))

(defn nonzero-or-nil? [n :- (U nil Int)]
  (or (pos-or-neg? n) (nil? n)))
```

Function pos-or-neg? could not be applied to arguments:

Domains:
        Int

Arguments:
        (U Int nil)

Ranges:
        boolean

in: (pos-or-neg? n)

1 errors type

CrossClj org.clojure/clojure clojure.core 1.7.0-alpha2 docs

Source Docs

index

NAMESPACES
clojure
core
protocols
reducers
data
edn
inspector
instant
java
browse
browse-ui
io
javadoc
shell
main
parallel
pprint
reflect
repl
set
stacktr
string
templat
test
junit
tap
uuid
walk
xml
zip

VARS
*
*'
*1
*2
*3
agent
*allow-unresolved-vars
assert
clojure-version
command-line-args
compile-files
compile-path
compiler-options
data-readers
*default-data-reader-f
*e
err
file
n-newline
ely
ably
e-path
unchecked-math
lassload
ords*
tion*
+
+'
-
-'
->
->>
->ArrayChunk
->Iteration
->Vec
->Vec
->Vec
-cache
-reset
..
/
<
<=
=

Francesco Bellomi

RECENT
clojure.core
clojure.core.typed.error
clojure.core.typed.frees
clojure.core.typed.test.
org.clojure/core.typed

accept 2 arguments, index and item.

map? 1.0 Source Usages

```
(map? x)
```

```
(λ [Any → Boolean :filters {:then (is (Map Any Any) 0), :else (! (Map Any Any) 0)}])
```

Return true if x implements IPersistentMap

mapcat 1.0 Source Usages

```
(mapcat f)
(mapcat f & colls)
```

```
(∀ [c b ...]
   (λ [(λ [b ... b → (Option (Seqable c))]) (Option (Seqable b)) ... b → (ASeq c)]))
```

Returns the result of applying concat to the result of applying map
to f and colls. Thus function f should return a collection. Returns
a transducer when no collections are provided

mapv 1.4 Source Usages

```
(mapv f coll)
(mapv f c1 c2)
(mapv f c1 c2 c3)
(mapv f c1 c2 c3 & colls)
```

```
(∀ [c a b ...]
   (λ [(λ [a b ... b → c]) (NonEmptySeqable a) (NonEmptySeqable b) ... b → (NonEmptyAVec c)]
      [(λ [a b ... b → c]) (U nil (Seqable a)) (U nil (Seqable b)) ... b → (AVec c)]))
```

Returns a vector consisting of the result of applying f to the
set of first items of each coll, followed by applying f to the set
of second items in each coll, until any one of the colls is
exhausted. Any remaining items in other colls are ignored. Function
f should accept number-of-colls arguments.

max 1.0 Source Usages

```
(max x)
(max x y)
(max x y & more)
```

```
(λ [Number Number * → Number])
```

Returns the greatest of the nums.

max-key 1.0 Source Usages

CrossClj.info

mapcat 1.0 Source Usages

```
(mapcat f)
(mapcat f & colls)
```

```
(∀ [c b ...]
   (λ [(λ [b ... b → (Option (Seqable c))]) (Option (Seqable b)) ... b → (ASeq c)]))
```

Returns the result of applying concat to the result of applying map
to f and colls. Thus function f should return a collection. Returns
a transducer when no collections are provided

mapv 1.4 Source Usages

```
(mapv f coll)
(mapv f c1 c2)
(mapv f c1 c2 c3)
(mapv f c1 c2 c3 & colls)
```

```
(∀ [c a b ...]
   (λ [(λ [a b ... b → c]) (NonEmptySeqable a) (NonEmptySeqable b) ... b → (NonEmptyAV
      [(λ [a b ... b → c]) (U nil (Seqable a)) (U nil (Seqable b)) ... b → (AVec c)]))
```

Returns a vector consisting of the result of applying f to the
set of first items of each coll, followed by applying f to the set
of second items in each coll, until any one of the colls is
exhausted. Any remaining items in other colls are ignored. Function
f should accept number-of-colls arguments.

max 1.0 Source Usages

```
(max x)
(max x y)
(max x y & more)
```

```
(λ [Number Number * → Number])
```

```
 1 (ns clojure.core.typed.test.collatz
 2   (:require [clojure.core.typed :refer [ann] :as t]))
 3
 4 (ann collatz [Number -> Number])
 5 (defn collatz [n]
 6   (cond
 7     (= 1 n)
 8      1
 9     (and (integer? n)
10          (even? n))
11      (collatz (/ n 2))
12     :else
13      (collatz (inc (* 3 n)))))
14
15 (collatz 10)
~
~
~
~
~
~
```

(t/IFn [java.lang.Number -> java.lang.Number])

Part of the Typed Clojure project

TypedClojure.vim

Minori Yamashita

Di Xu

Typed Clojure 101

Occurrence Typing

```
(ns gen-vec)

(defn gen-vec [n-or-v]
  (if (number? n-or-v)
    (vec (range n-or-v))
    n-or-v))
```

```
(ns gen-vec)

(defn gen-vec [n-or-v]
  (if (number? n-or-v)
    (vec (range n-or-v))
    n-or-v))

(gen-vec 5)
;=> [0 1 2 3 4]

(gen-vec [1 2 3 4])
;=> [1 2 3 4]
```

```
(gen-vec      )
```

```
(gen-vec 5 )
```

```
(if (number? 5 )..... true
  (vec (range 5 ))
5 ))
```

```
(vec (range 5 ))
```

```
[0 1 2 3 4]
```

```
(gen-vec                    )
```

(gen-vec [1 2 3])

(if (number? [1 2 3]) ..... false
(vec (range [1 2 3]))
[1 2 3]))

[1 2 3]

```
(ns gen-vec
  (:require [clojure.core.typed
             :refer [U Num Vec Int ann]
             :as t]))

(ann gen-vec [(U Num (Vec Int)) -> (Vec Int)])
(defn gen-vec [n-or-v]
(if (number? n-or-v)
  (vec (range n-or-v))
  n-or-v))
```

```
(ns gen-vec
  (:require [clojure.core.typed
             :refer [U Num Vec Int ann]
             :as t]))

(ann gen-vec [(U Num (Vec Int)) -> (Vec Int)])
(defn gen-vec [n-or-v]
  (if (number? n-or-v)
    (vec (range n-or-v))
    n-or-v))

=> (t/check-ns)
:ok
```

```
(ann clojure.core/number?
     (Pred Num))
```

HMaps

```
(assoc {} :a 1 :b “foo” :c ‘baz)
```

is of type

```
(HMap :mandatory {:a Num :b Str :c Sym})
```

```
(assoc {} :a 1 :b ‘‘foo’’ :c ‘baz)
```

is of type

```
(HMap :mandatory {:a Num :b Str :c Sym})
```

aka.

```
‘{:a Num :b Str :c Sym}
```

Multimethods

```clojure
(ann f [(U Num Sym) -> Num])
(defmulti f class)

(defmethod f Number [n] (inc n))
(defmethod f Symbol [s] (count (name s)))
```

Variable-Arity Polymorphism

Calling *map*

```
;; 2 args
(map (fn [a :- Int] (inc a))
     [1 2 3])
```

```
;; 3 args
(map (fn [a :- Int
          b :- Sym]
       [(inc a) (name b)])
     [1 2 3]
     ['a 'b 'c])
```

map types

```
;; 2 args
(All [x y]
  [[x -> y]
   (U nil (Seqable x))
   -> (Seq y)])
```

```
;; 3 args
(All [x y z]
  [[x z -> y]
   (U nil (Seqable x))
   (U nil (Seqable z))
   -> (Seq y)])
```

All *map* types

```
;; n args
(All [x y z ...]
  [[x z ... z -> y]
   (U nil (Seqable x))
   (U nil (Seqable z)) ... z
   -> (Seq y)])
```

Datatypes and Protocols

```
(t/defprotocol Adder
  (add [this y :- Num] :- Adder))

(t/deftype A [x :- Num]
  Adder
  (add [this y] (+ x y)))

(add (A. 1) 34)
```

Java Interop

```
(ann grand-parent [File -> (U nil File)])
(defn grand-parent [^File f]
  (let [p1 (.getParentFile f)]
    (.getParentFile p1)))
```

```clojure
(ann grand-parent [File -> (U nil File)])
(defn grand-parent [^File f]
  (let [p1 (.getParentFile f)]
    (.getParentFile p1)))
```

```
=> (t/check-ns)
Cannot call instance method
  java.io.File/getParentFile

on type
  (U nil File)
```

```
(ann grand-parent [File -> (U nil File)])
(defn grand-parent [^File f]
  (let [p1 (.getParentFile f)]
    (.getParentFile p1)))
```

```
(ann grand-parent [File -> (U nil File)])
(defn grand-parent [^File f]
  (let [p1 (.getParentFile f)]
```

```
(ann grand-parent [File -> (U nil File)])
(defn grand-parent [^File f]
  (let [p1 (.getParentFile f)]
    (when p1
      (.getParentFile p1))))
```

```
=> (t/check-ns)
:ok
```

Sample Application

https://github.com/typedclojure/core.typed-example

lein Config

```
(defproject fire.simulate "0.1.0-SNAPSHOT"
  ...
  :profiles {:dev {:dependencies
                   [[org.clojure/core.typed "0.2.72"]]}}
  :dependencies [[org.clojure/core.typed.rt "0.2.72"]
                 ...]

  :plugins [[lein-typed "0.3.5"]]
  :core.typed {:check [fire.simulate
                       fire.main
                       fire.gnuplot
                       fire.simulate.percolation
                       fire.simulate-test]})
```

lein-typed

https://github.com/typedclojure/lein-typed

```
$ lein typed check
...
Start checking fire.simulate
...
Start checking fire.simulate.percolate
...
:ok
$
```

Refer clojure.core

```
(ns fire.simulate
  "This namespace defines operations for the study of
percolation in the forest fire simulation."
(:refer-clojure :exclude [for fn doseq dotimes])
(:require [clojure.core.typed
           :refer [for fn doseq dotimes]
           :as t])
(:import (java.io Writer)))
```

Aliases

```
(defalias Point
  "A point in 2d space."
  '[Int Int])
```

Aliases

```
(defalias Grid
  "An immutable snapshot of the world state.

  - :grid   the grid
  - :rows   number of rows
  - :cols   number of columns
  - :history  a chronological vector of interesting data of
              previous states.
              See GridHistory.
- :q   the initial probability of a green tree at time 0 at a site
- :p   the probability a tree will grow in an empty site
- :f   the probability a site with a tree will burn (lightning)
- :frame    the frame number corresponding to the grid,
              displayed in the gnuplot graph"
'{:grid GridVector :rows Int :cols Int :history GridHistory
  :q Num :p Num :f Num :frame Int})
```

Aliases

```
;-------------------------------------------------------------
; Type Aliases
;-------------------------------------------------------------

(defalias State
  "A point can either be empty, a tree, or a burning tree."
  (U ':burning ':tree ':empty))

(defalias GridHistoryEntry
  "Some data about a particular state.

  - :nburning  number of burning points at this state
  - :ntrees  number of green trees at this state"
  '{:nburning Int :ntrees Int :nempty Int})

(defalias GridHistory
  "A vector of history states on a grid"
  (Vec GridHistoryEntry))

(defalias GridVector
  "A vector of vectors of states representing a cellular
  automata lattice."
  (Vec (t/Vec State)))

(defalias Grid
  "An immutable snapshot of the world state.

  - :grid   the grid
  - :rows   number of rows
  - :cols   number of columns
  - :history  a chronological vector of interesting data of previous states.
              See GridHistory.
  - :q   the initial probability of a green tree at time 0 at a site
  - :p   the probability a tree will grow in an empty site
  - :f   the probability a site with a tree will burn (lightning)
  - :frame    the frame number corresponding to the grid, displayed in the gnuplot graph"
  '{:grid GridVector :rows Int :cols Int :history GridHistory
    :q Num :p Num :f Num :frame Int})

(defalias Point
  "A point in 2d space."
  '[Int Int])
```

Succinct Annotations

```
(ann state->number [State -> Long])
(defn state->number
  "Convert the keyword representation of a state to
  a number usable by Gnuplot for plotting color gradient."
  [k]
  (case k
    :empty 0
    :tree 1
    :burning 2))
```

Static Assertions

```
(defalias GridVector (Vec (Vec State)))
(defalias Grid       '{:grid GridVector ...})

...

(ann flat-grid [Grid -> (Coll State)])
(defn flat-grid [{:keys [grid]}]
  (ann-form grid GridVector)
  (apply concat grid))
```

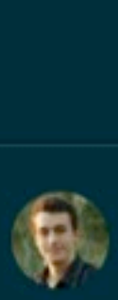
Resolve Reflection

```
(defalias GnuplotP
  "A gnuplot process.

  - :proc  The Process instance
  - :out   A Writer piping to gnuplot's stdin
  - :in    A Reader reading from gnuplot's stdout"
  '{:proc Process, :out Writer, :in Reader})


(ann stop [GnuplotP -> Any])
(defn stop
  "Stop gnuplot process."
  [{:keys [^Process proc]}]
  (.destroy proc))
```

Java Overrides

```
; We know these method never return null.
(non-nil-return java.lang.Process/getOutputStream :all)
(non-nil-return java.lang.Process/getInputStream :all)
(non-nil-return java.lang.ProcessBuilder/start :all)

(ann start [-> GnuplotP])
(defn start
  "Start gnuplot process."
  []
  (let [proc (-> (doto (ProcessBuilder. '("gnuplot" "-persist"))
                   (.redirectErrorStream true))
                 .start)
        out  (io/writer (.getOutputStream proc))
        in  (io/reader (.getInputStream proc))]
    {:proc proc :out out :in in}))
```

Case Study



frenchy64

Documentation

Report Bug

Live Support

Add Projects

Changelog

Collapse

Your Branch Activity

circleci/circle

- coretyped-65 ✗
- master ✓
- noslim ✗

frenchy64/core.typed

- master ✗

circleci

machine

Configure the build

Restore cache

dependencies

$ lein deps *in*

$ mvn dependency:resolve *in*

database

Save cache

test

$ mvn test

```
$ mvn test                                                    E
[INFO] Scanning for projects...
[INFO]
------------------------------------------------------------------
---------
[INFO] Reactor Build Order:
[INFO]
[INFO] core.typed-pom
[INFO] core.typed.rt
[INFO] core.typed
[INFO]
[INFO] Using the builder
org.apache.maven.lifecycle.internal.builder.singlethr
```

Why we're supporting Typed Clojure, and you should too!

by circleci on September 27, 2013

tl;dr Typed Clojure is an important step for not just Clojure, but all dynamic languages. CircleCI is supporting it, and you should too.

Typed Clojure is one of the biggest advancements to dynamic programming languages in the last few decades. It shows that you can have the amazing flexibility of a dynamic language, while providing lightweight, optional typing. Most importantly, this can make your team more productive, and it's ready to use in production.

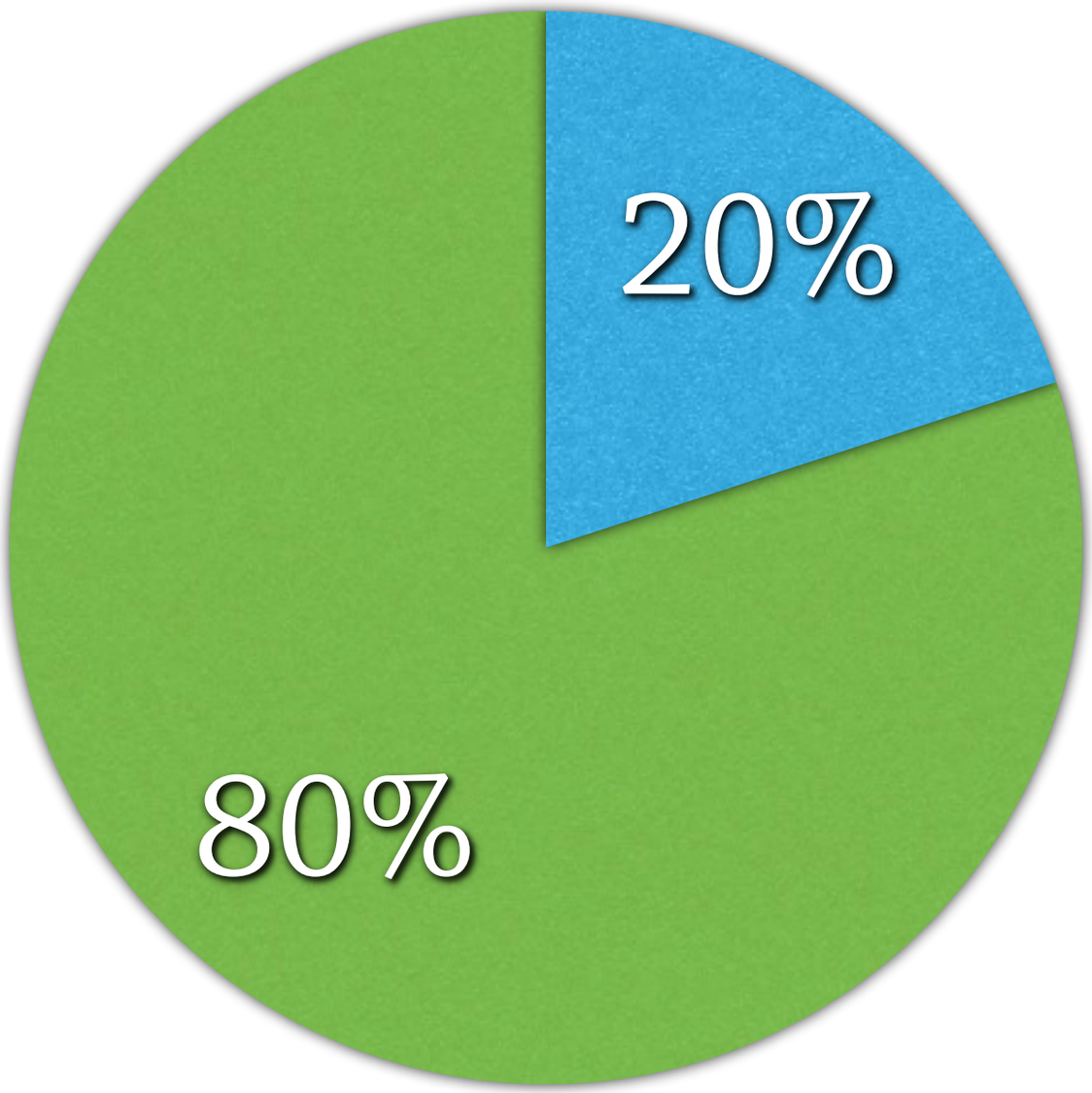
Even if you don't use Clojure, you should support the Typed Clojure campaign, because its success will help developers in your language realize how great optional typing can be in everyday code. Whether you write Ruby or Python or JavaScript or whatever, what we're learning from Typed Clojure can be applied to your language.

Does it work?

Oh yes! CircleCI has been using it in production for 3 months. We started by using it in areas that are hard to test, such as code which allocates AWS machines and VMs. That's right, we're type-checking devops!

We now using it in 45 namespaces, about 20% of our codebase. Those are covered using only 300 type annotations. And obviously, this has made us more productive, by finding bugs before we ship them.

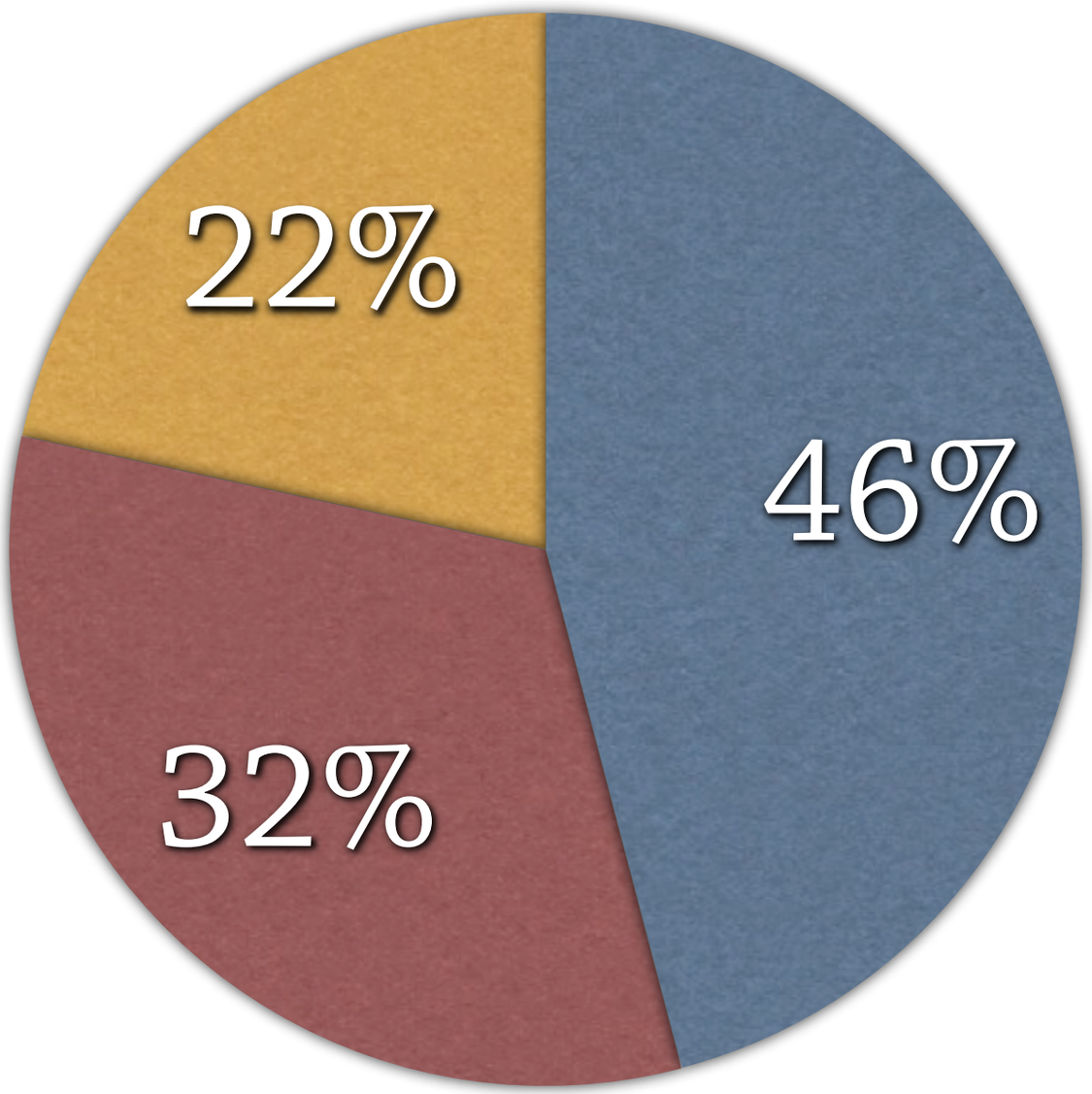
circleci



Approx. Lines of Code (Total ~50,000)

- Typed Clojure
- Clojure

Var Annotations (Total 588)



- Checked
- Not checked
- Libraries

Sample code

```
(defn encrypt-keypair [{:keys [private-key] :as keypair}]
  (assoc (dissoc keypair :private-key)
    :encrypted-private-key (encrypt private-key)))
```

```
(ann encrypt-keypair [RawKeyPair -> EncryptedKeyPair])
```

```
(defn encrypt-keypair [{:keys [private-key] :as keypair}]
  (assoc (dissoc keypair :private-key)
    :encrypted-private-key (encrypt private-key)))
```

```
(defalias RawKeyPair
  "A keypair with a raw private key"
  (HMap :mandatory {:public-key RawKey,
                    :private-key RawKey}
        :complete? true))
```

```
(defalias EncryptedKeyPair
  "A keypair with an encrypted private key"
  (HMap :mandatory {:public-key RawKey,
                    :encrypted-private-key EncryptedKey}
        :complete? true))
```

```
(ann encrypt-keypair [RawKeyPair -> EncryptedKeyPair])
(defn encrypt-keypair [{:keys [private-key] :as keypair}]
  (assoc (dissoc keypair :private-key)
    :encrypted-private-key (encrypt private-key)))
```

```
(ann encrypt-keypair [RawKeyPair -> EncryptedKeyPair])
(defn encrypt-keypair [{:keys [private-key] :as keypair}]
  (assoc keypair
    :encrypted-private-key (encrypt private-key)))
```

```
=> (t/check-ns)

Type mismatch:

Expected:  EncryptedKeyPair

Actual:    (HMap :mandatory
                 {:encrypted-private-key EncryptedKey,
                  :public-key RawKey,
                  :private-key RawKey}
                 :complete? true)
```

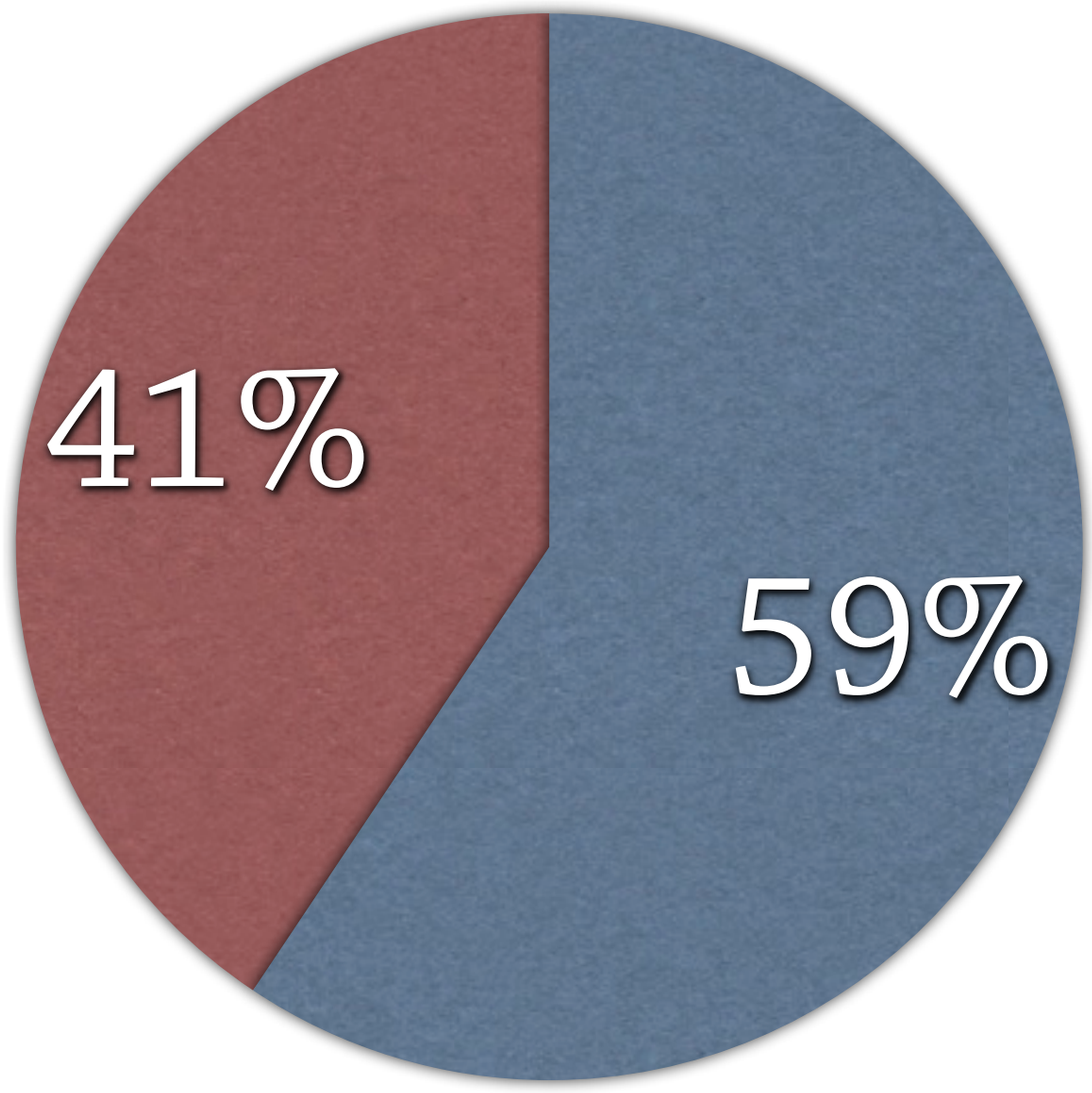
circleci

{:a 1}

Type aliases

(Total 64)

- HMap
- Non-HMap

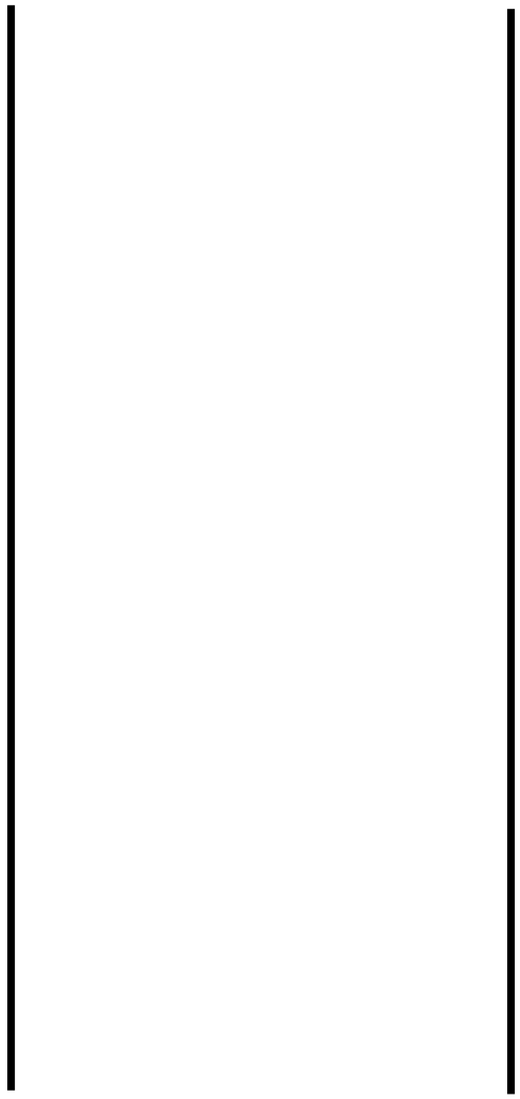


The Future

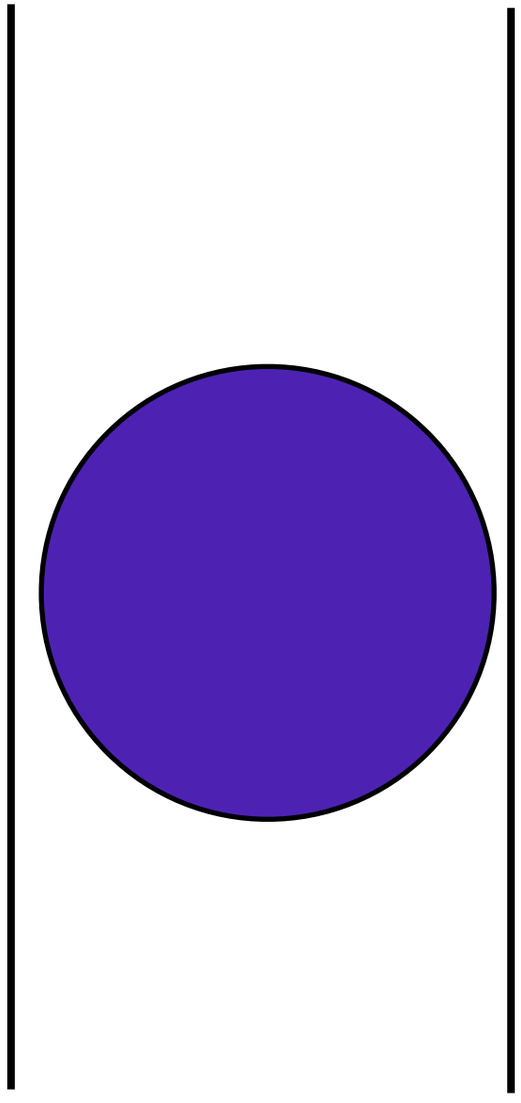
Typed ClojureScript

Towards Gradual Typing

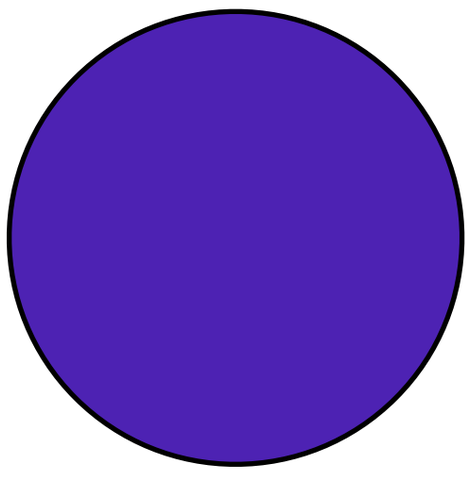
Towards Gradual Typing



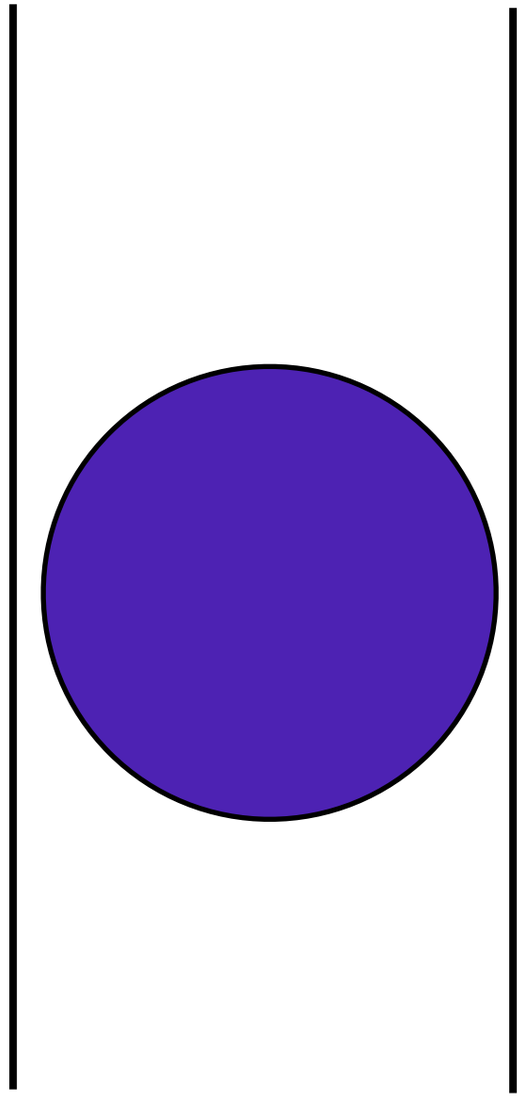
Towards Gradual Typing

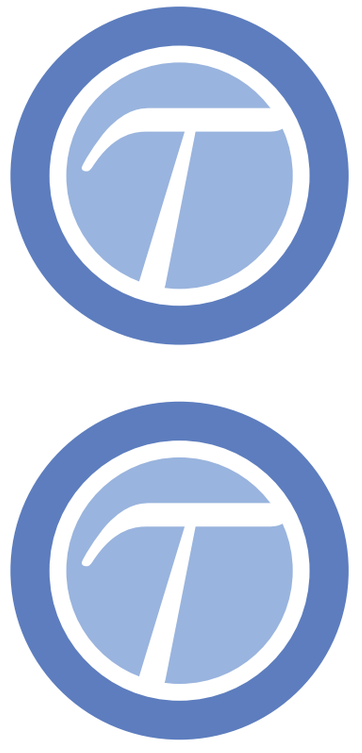


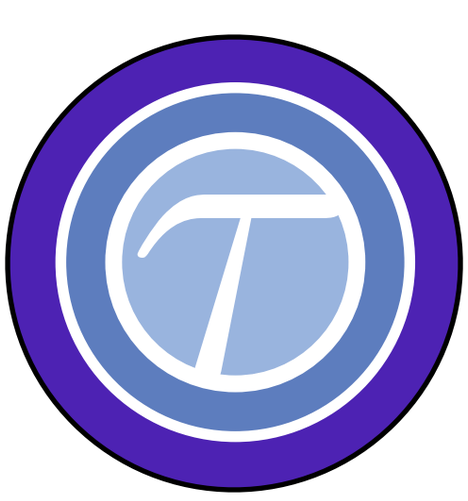
Towards Gradual Typing



Towards Gradual Typing







Call to Action

Annotate your

libraries

Conclusion

- Typed Clojure works in Production
- Get it at typedclojure.org
- Annotate your libraries

Conclusion

- Typed Clojure works in Production
- Get it at typedclojure.org
- Annotate your libraries

Thank you

Ambrose Bonnaire-Sergeant

@ambrosebs