Typed Clojure in Practice

Ambrose Bonnaire-Sergeant
Indiana University
@ambrosebs

```
(ns stl2014.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)
    ...
```

```
(ns stl2014.sum)

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

```clojure
(ns stl2014.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)))
```

```clojure
(ns stl2014.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)))
```

```
(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)
Found 5 type errors:
...
Unannotated var stl2014.sum/summarise
...
```

```
(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)))
```

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

```
(ann summarise


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

```
(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)))
```

```
(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)))
```

```
(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)))
```

```
(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))
```

```
(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))
```

```
(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))
```

```
(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))
```

```
(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))
```

```
(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)
```

```
Function summarise could not be applied to arguments:

Domains:
    NInts Int

Arguments:
    (t/ASeq Int) Int

Ranges:
    Int
in: (summarise (rest nseq) (clojure.lang.Numbers/inc acc))
```

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

```
Function summarise could not be applied to arguments:

Domains:
    NInts Int

Arguments:
    (t/ASeq Int) Int

Ranges:
    Int
in: (summarise (rest nseq) (clojure.lang.Numbers/inc acc))
```

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

```
Function summarise could not be applied to arguments:

Domains:
    NInts Int

Arguments:
    (t/ASeq Int) Int

Ranges:
    Int
in: (summarise (rest nseq) (clojure.lang.Numbers/inc acc))
```

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

```
Function summarise could not be applied to arguments:

Domains:
    NInts Int

Arguments:
    (t/ASeq Int) Int

Ranges:
    Int
in: (summarise (rest nseq) (clojure.lang.Numbers/inc acc))
```

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

```
Function summarise could not be applied to arguments:

Domains:
    NInts Int

Arguments:
    (t/ASeq Int) Int

Ranges:
    Int
in: (summarise (rest nseq) (clojure.lang.Numbers/inc acc))
```

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 [])
;=> ()

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

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

(next nil)
;=> 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 3
stl2014.sum/summarise (form-init6981015802 3
stl2014.sum/eval3757 (form-init69810158023 9
clojure.lang.Compiler.eval (Compiler.java:6
...
```

Thursday, 18 September 14

(summarise [])

Function summarise could not be applied to arguments:

Domains:

NInts

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

Arguments:

(HVec [])

...

Thursday, 18 September 14

Thursday, 18 September 14

```
(ns stl2014.sum
  (:require [clojure.core.typed :as t
             :refer [ann IFn U NonEmptyColl Int
                     defalias]]))

(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)))
```

```
(ns stl2014.sum
  (:require [clojure.core.typed :as t
             :refer [ann IFn U NonEmptyColl Int
                     defalias]]))

(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)))
```

```
(ns stl2014.sum
  (:require [clojure.core.typed :as t
             :refer [ann IFn U Coll Int
                     defalias]]))

(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)))
```

```
(ns stl2014.sum
  (:require [clojure.core.typed :as t
             :refer [ann IFn U Coll Int
                     defalias]]))

(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)))
```

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

```
Static method clojure.lang.Numbers/multiply could not be applied to
arguments:

Domains:
                    Number                    Number

Arguments:
                     Int                    (U nil Int)

...
```

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

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

Static method clojure.lang.Numbers/multiply could not be applied to arguments:

Domains:

Number

Arguments:

Int

...

Number

(U nil Int)

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)
```

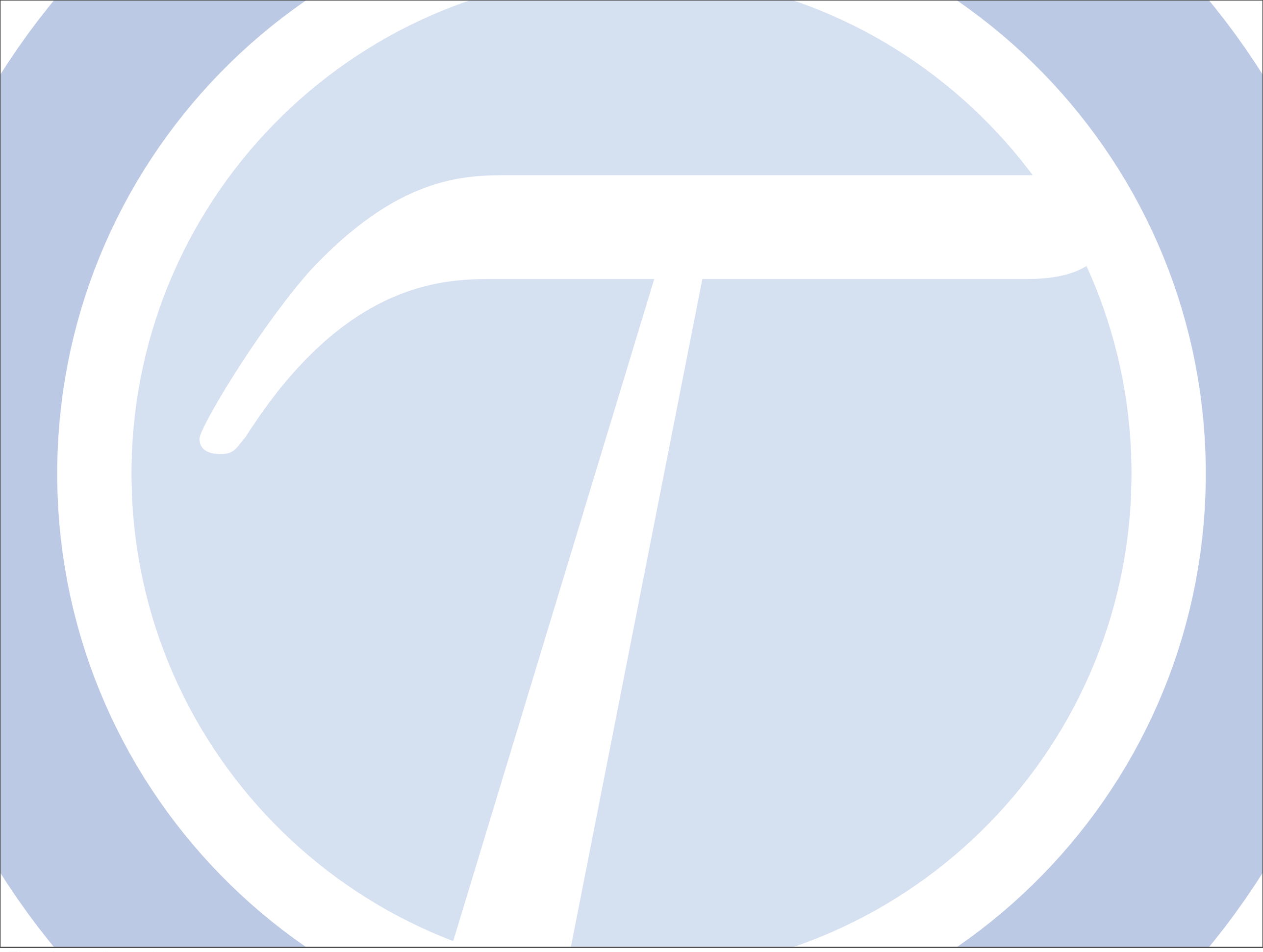
```
(seq nil)
;=> nil
```

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

```
(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 (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

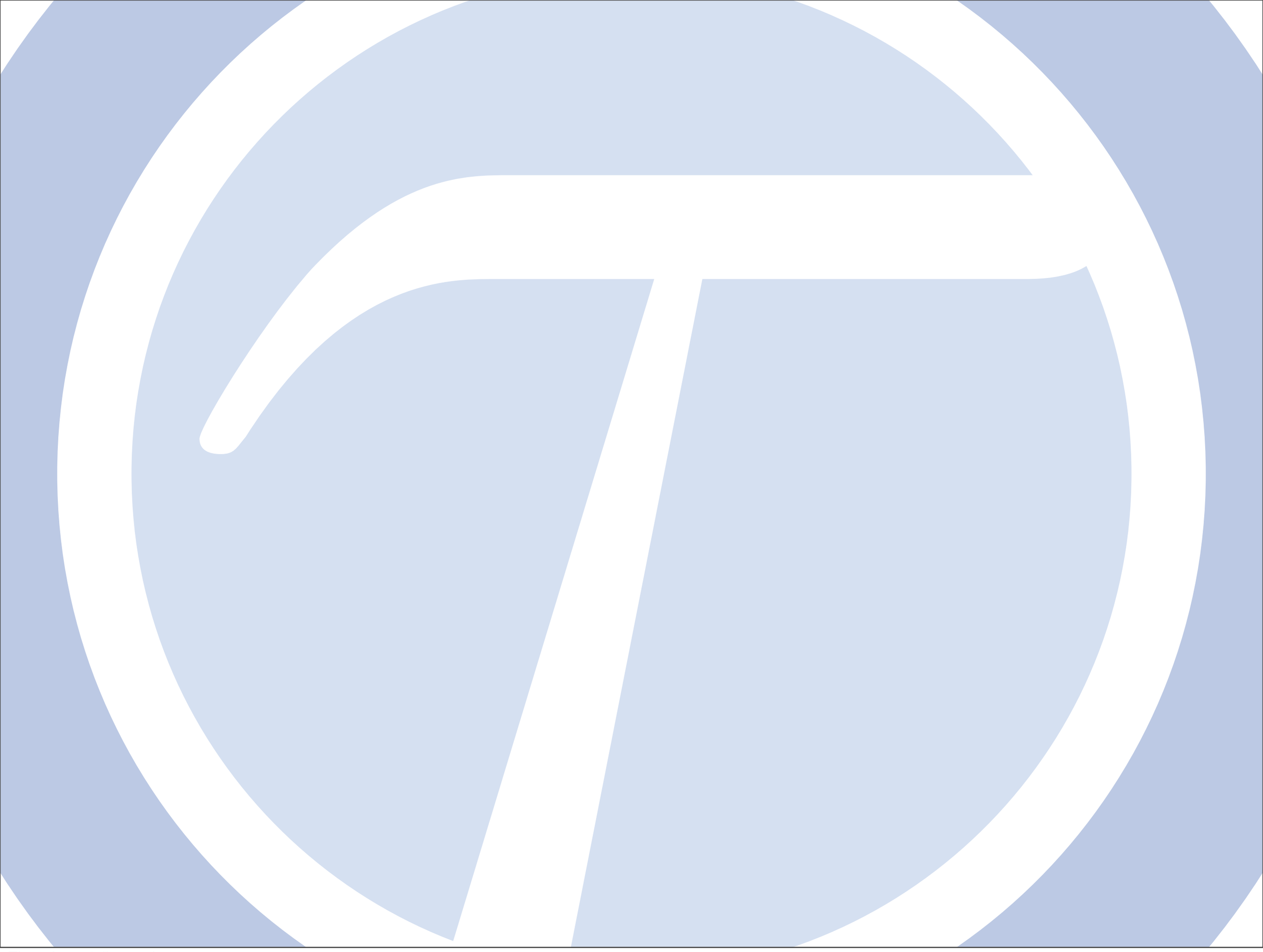
Optional Type System

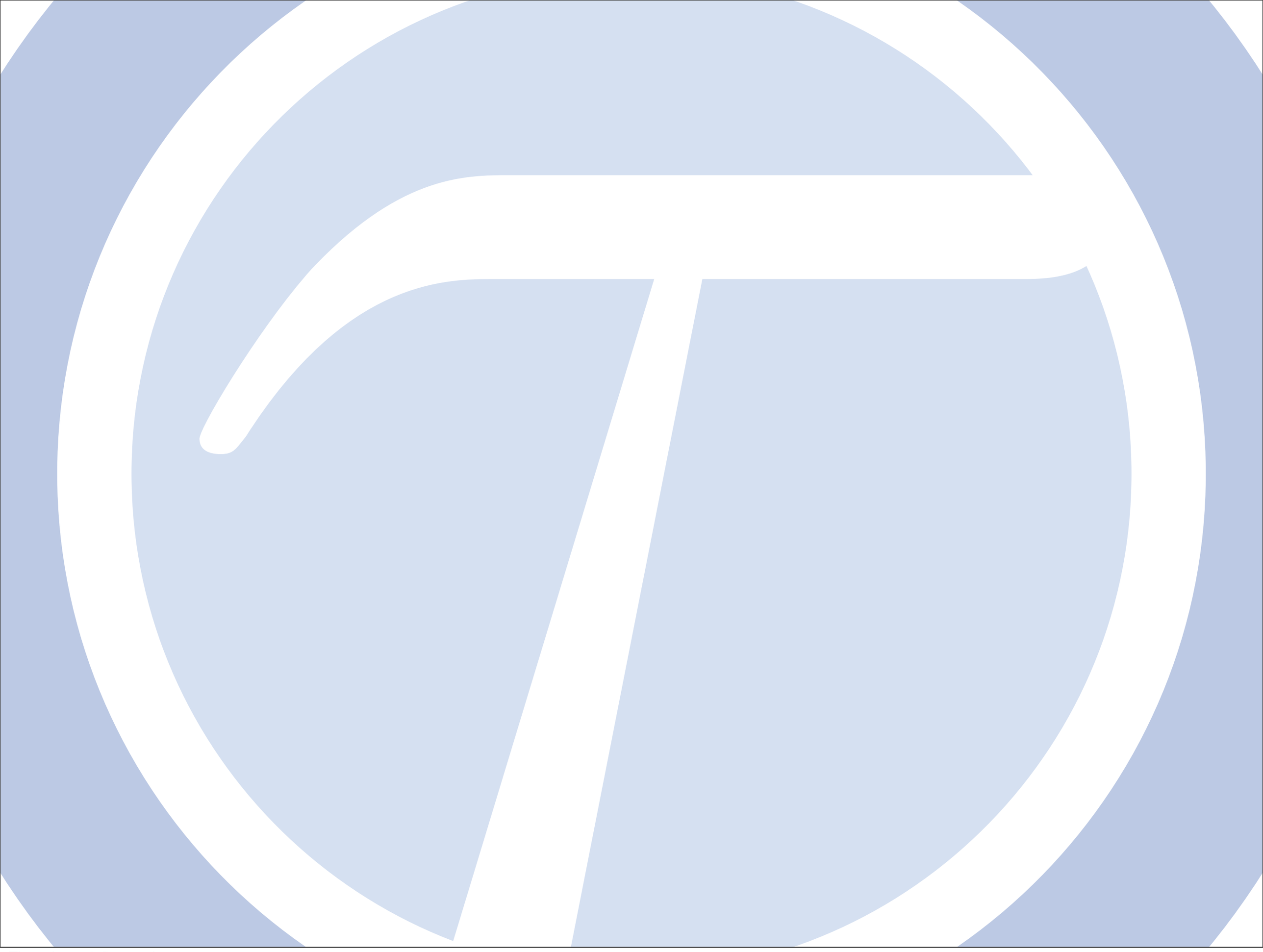
that

catches type errors

in

real Clojure code





Where have we come from?

2011


simplicity

Thursday, 18 September 14

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

clojure / core.typed

branch: master core.typed / README.md

frenchy64 13 days ago Bump readme version

1 contributor

199 lines (148 sloc) 5.577 kb Raw Blame History

core.typed

Part of the

Typed Clojure

project

Gradual typing in Clojure, as a library.

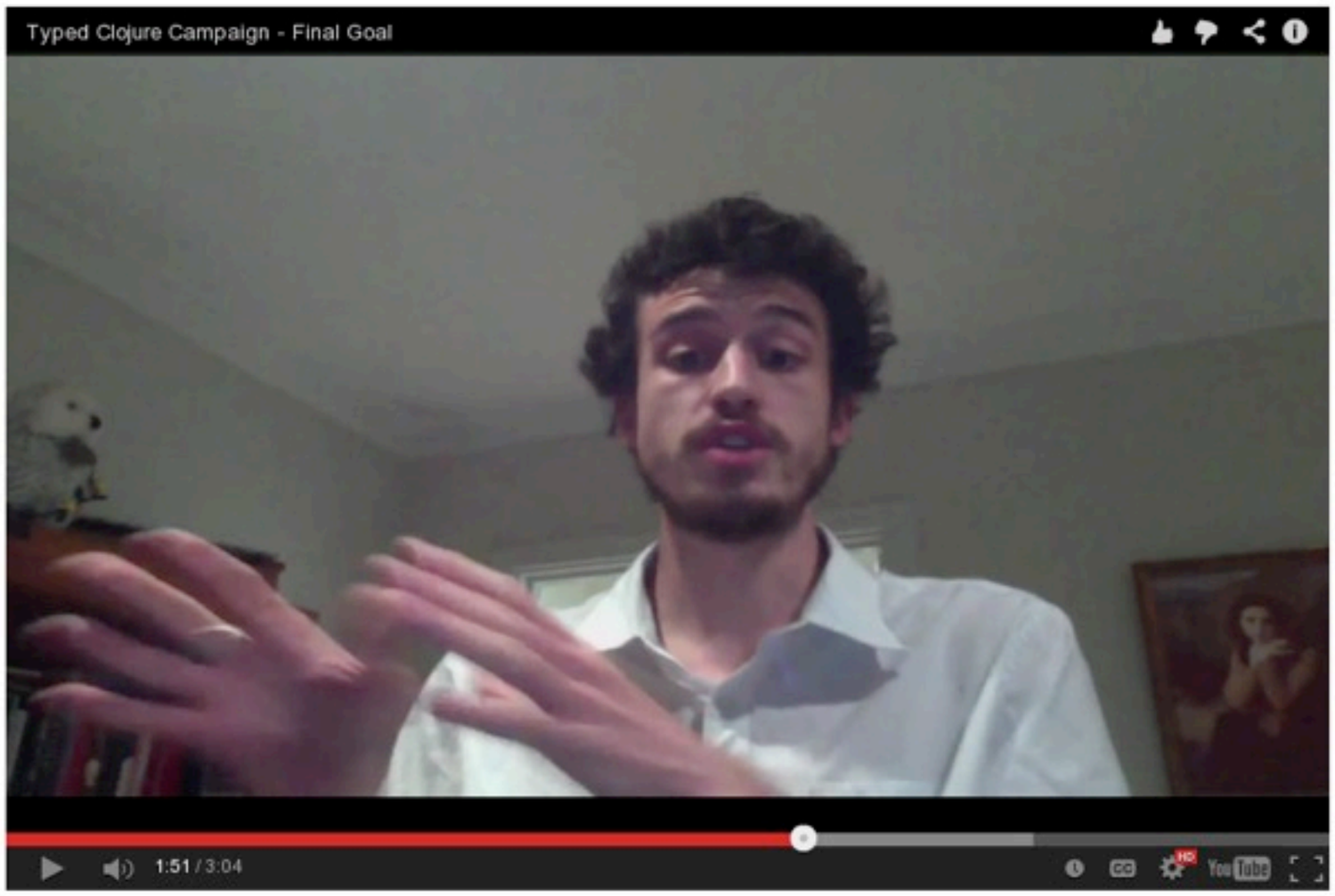
Releases and Dependency Information

Latest stable release is 0.2.68


Clojure
conj
Sheraton
Raleigh

Typed Clojure

Story | Updates 36 | Comments 22 | Funders 545 | Gallery 7



82 Share | 272 Tweet | 24 +1 | Email | Embed | Link | Follow

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

Cursive Clojure

sonian

circleci

SNOWPLOW

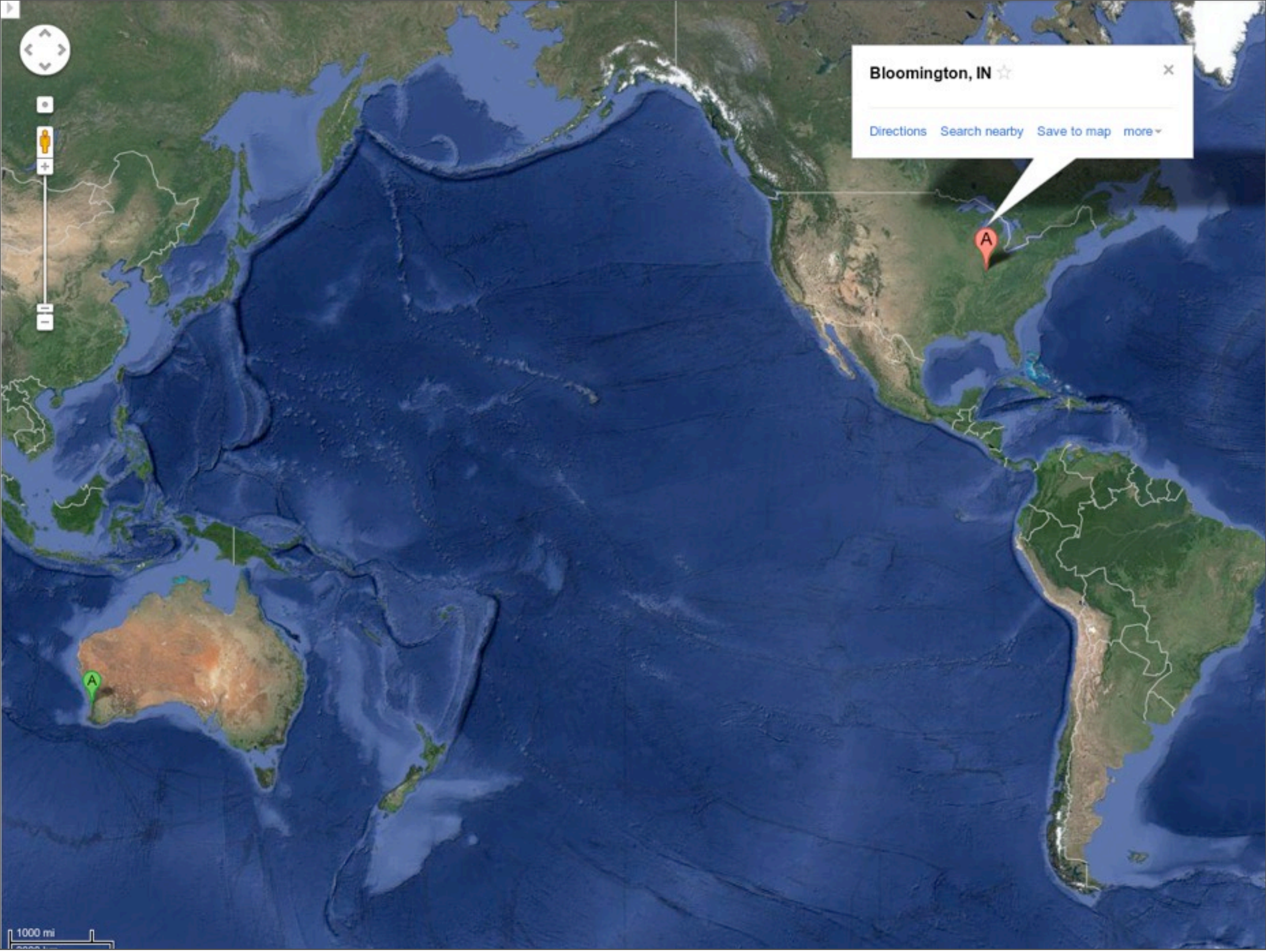
cognitect

Hacker School

Leonidas

BRICK ALLOY

THORTECH SOLUTIONS



Thursday, 18 September 14

Thursday, 18 September 14

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

```
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

Cursive Clojure

Colin Fleming

```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

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 typ

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
sely
ably
e-path
unchecked-math
*lassload
cords
ction
+
+'
-
-'
->
->>
->ArrayChunk
->Iteration
->Vec
->Vec
->Vec
-cache
-reset
..
/
<
<=
=

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

Francesco Bellomi

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

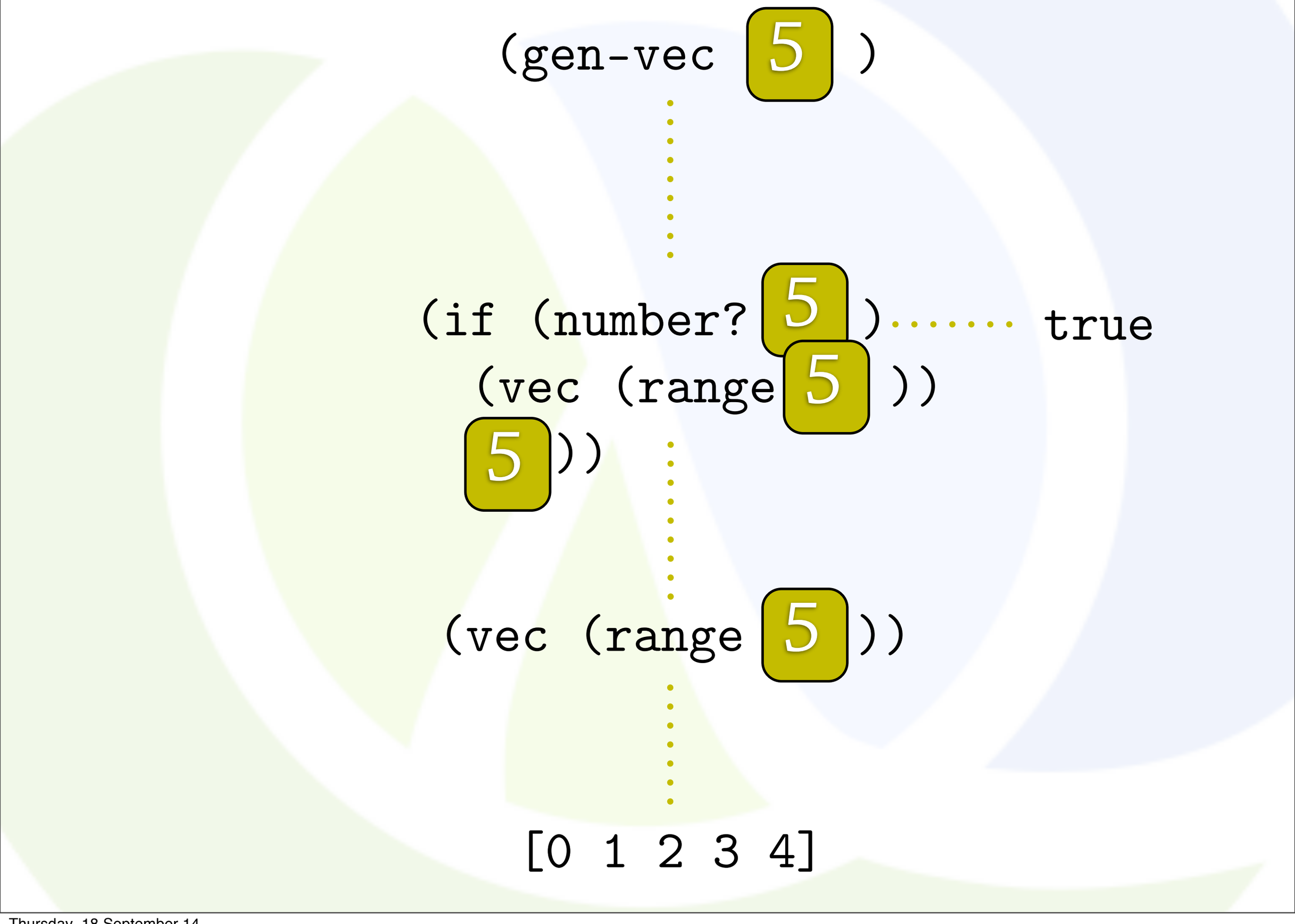
Features

Occurrence Typing

```
(ns stl2014.gen-vec)

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

```
(ns stl2014.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 stl2014.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 stl2014.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

```
(defalias Expr
  (Rec [Expr]
       (U '{:op (Val :if) :test Expr :then Expr :else Expr}
          '{:op (Val :const) :val Num})))

(ann walk [Expr [Num -> Num] -> Expr])
(defmulti walk (fn [e f] (:op e)))

(defmethod walk :if
  [{:keys [test then else]} f]
  {:op :if
   :test (walk test f)
   :then (walk then f)
   :else (walk else f)})

(defmethod walk :const
  [{:keys [val]} f]
  {:op :const :val (f val)})

(walk {:op :if
       :test {:op :const :val 1}
       :then {:op :const :val 2}
       :else {:op :const :val 3}}
      inc)
```

Polymorphism

```
(ann lift-chan (All [x y]
                    [[x -> y] -> [(Chan x) -> (Chan y)]]))
(defn lift-chan [function]
  (fn [in :- (Chan x)] :- (Chan y)
    (let [out (chan :- y)]
      (go
        (loop []
          (let [rcv (<! in)]
            (when rcv
              (>! out (function rcv))))
          (recur)))
      out)))

(ann upcase [(Chan String) -> (Chan String)])
(def upcase (lift-chan upper-case))
```

Variable-Arity Polymorphism

```
(map (fn [a :- Num,
          b :- Str,
          c :- Boolean]
       (+ a
          (count b)
          (if c 0 1)))
     [42 66 243]
     [“a” “foo” “bar”]
     [true false false])
```

Datatypes and Protocols

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

(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)))
```

```
(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
```

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

$ mvn dependency:resolve

database

Save cache

test

$ mvn test

```
$ mvn test
[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.

thanks, core.typed!

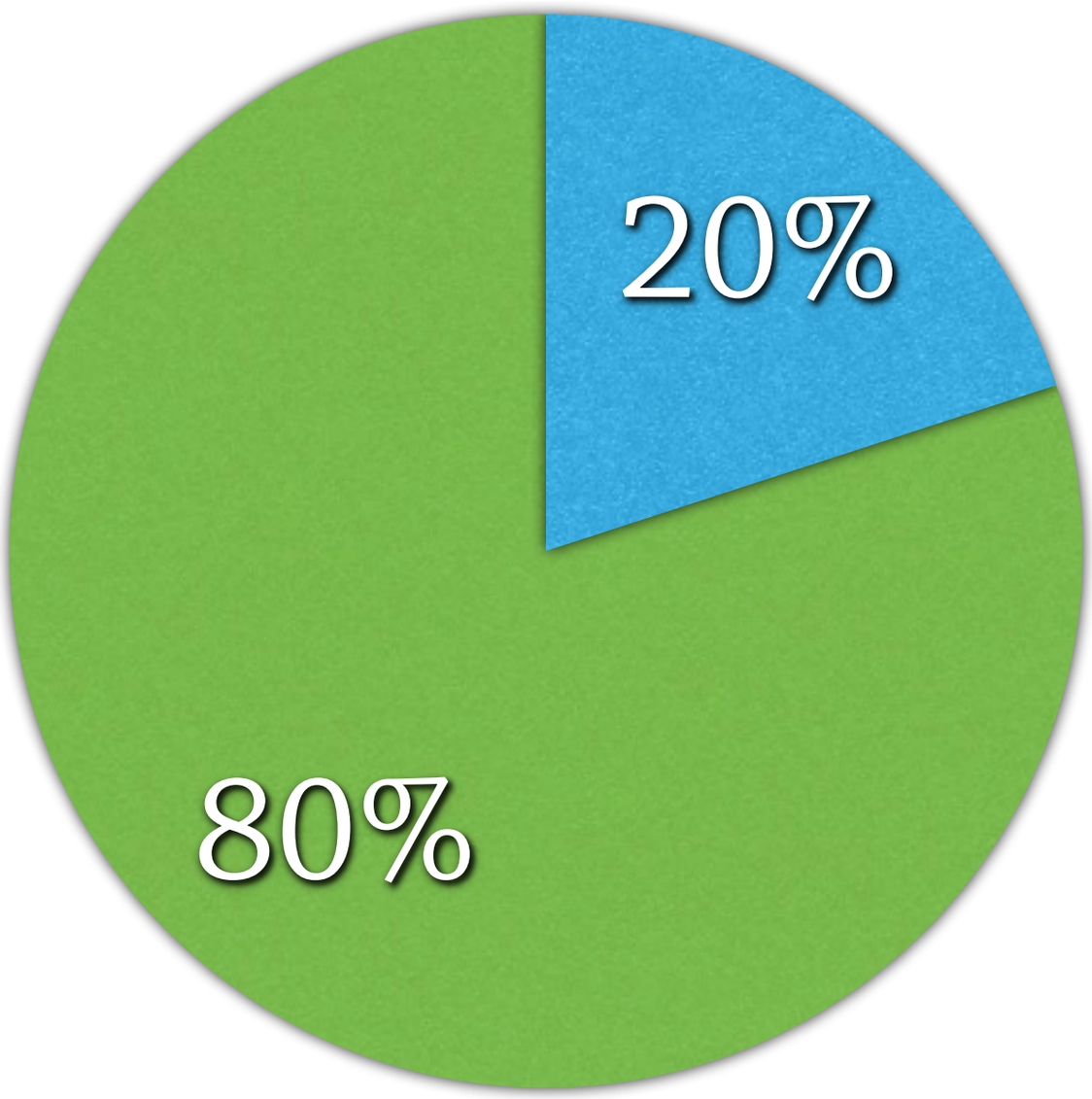
fs

dlowe authored 4 days ago 1 paren

Showing **1 changed file** with **1 addition** and **1 deletion**.

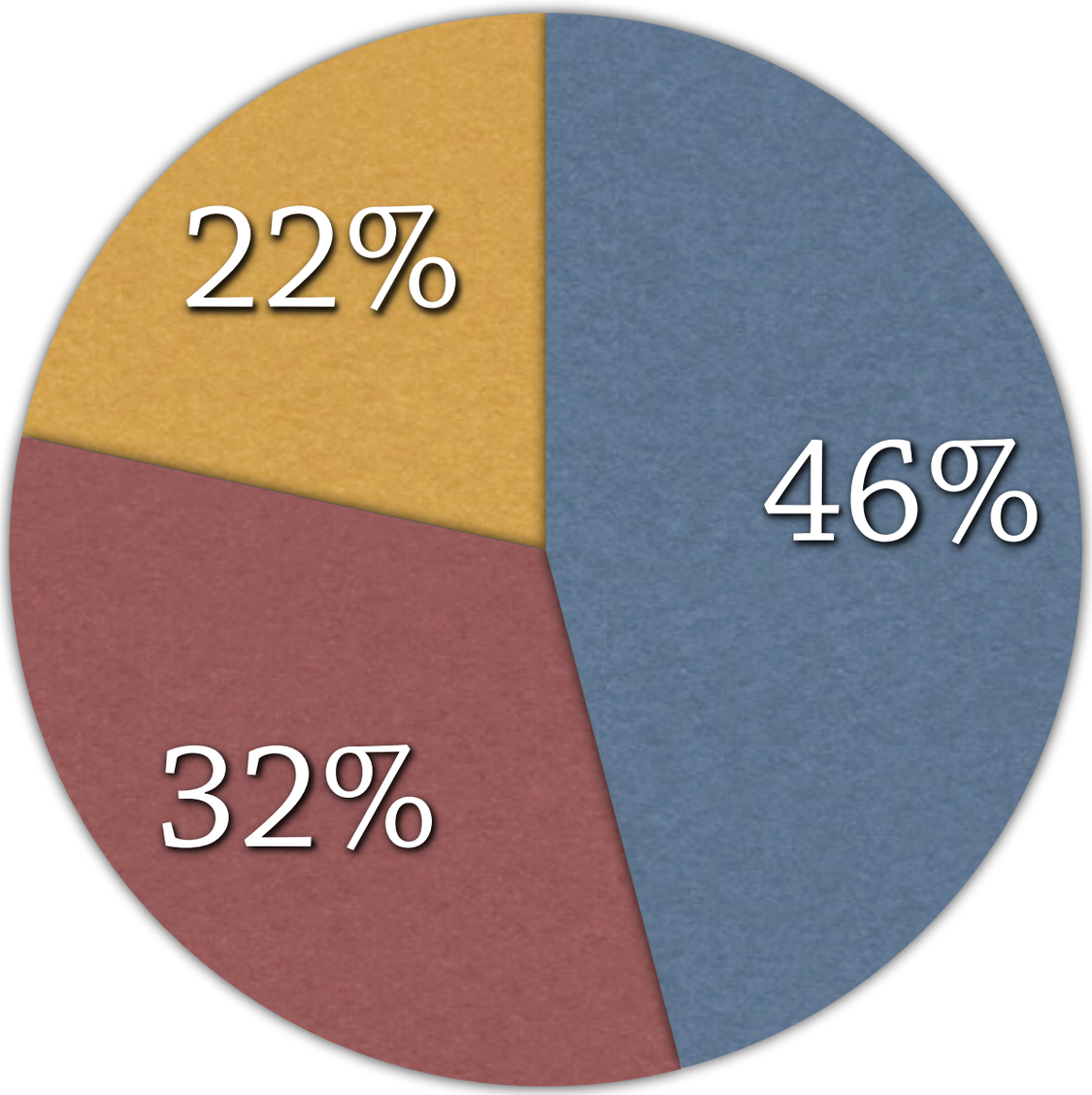
2 src/circle/backend/container_fs/fs.clj

```
...  ...  @@ -76,7 +76,7 @@
76   76       (sudo mkfs.fs
79        -           ~(str/join " " ~(disk/devices)))
     79   +           ~(str/join " " (disk/devices)))
80   80       (sudo mount
81   81             ~(first (disk/devices)) ; any of the s
82   82             "/mnt")
```



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)))
```

Thursday, 18 September 14

Thursday, 18 September 14

```
(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))
```

Thursday, 18 September 14

Thursday, 18 September 14

```
(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)))
```

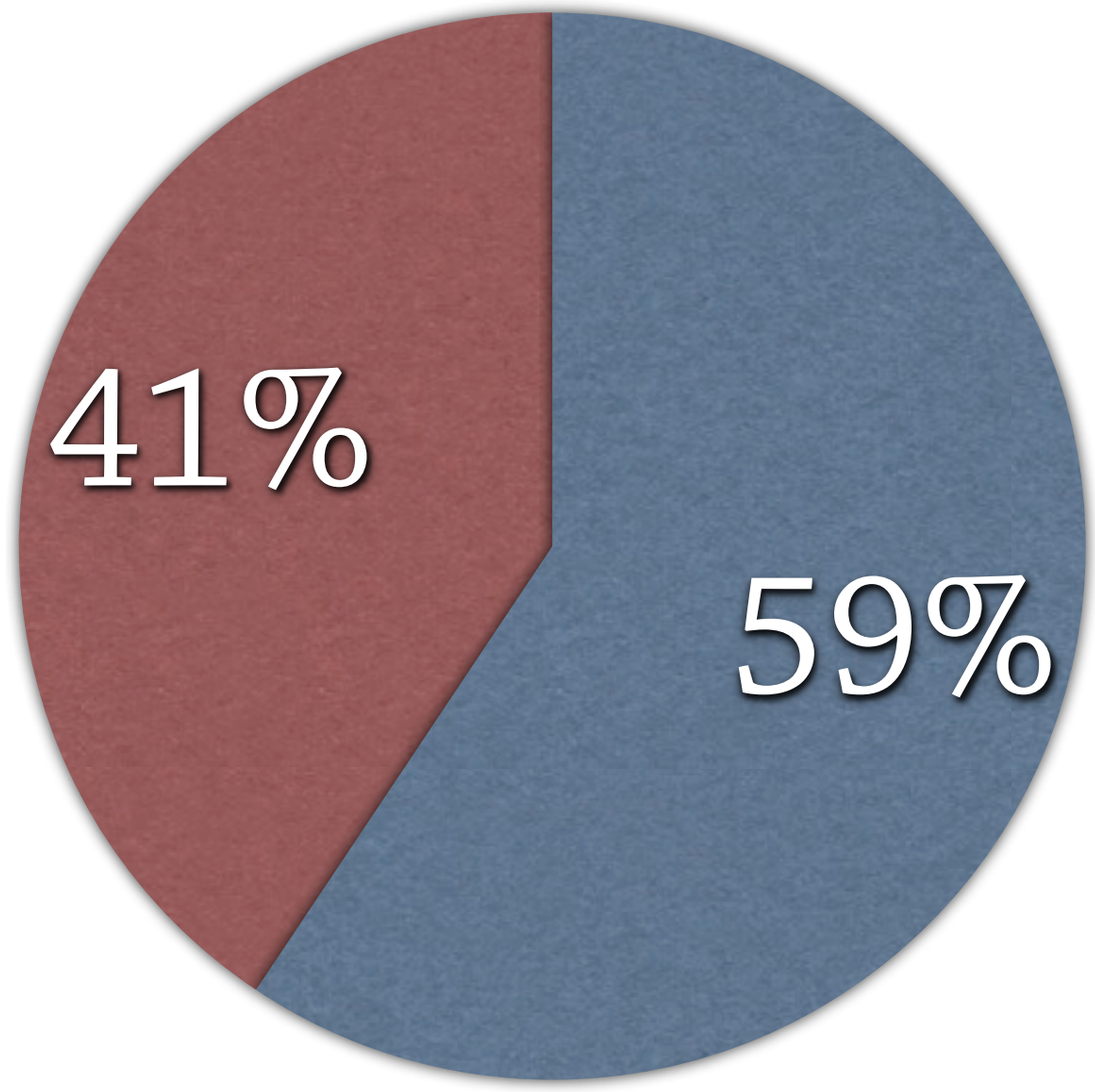
```
=> (t/check-ns)

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)



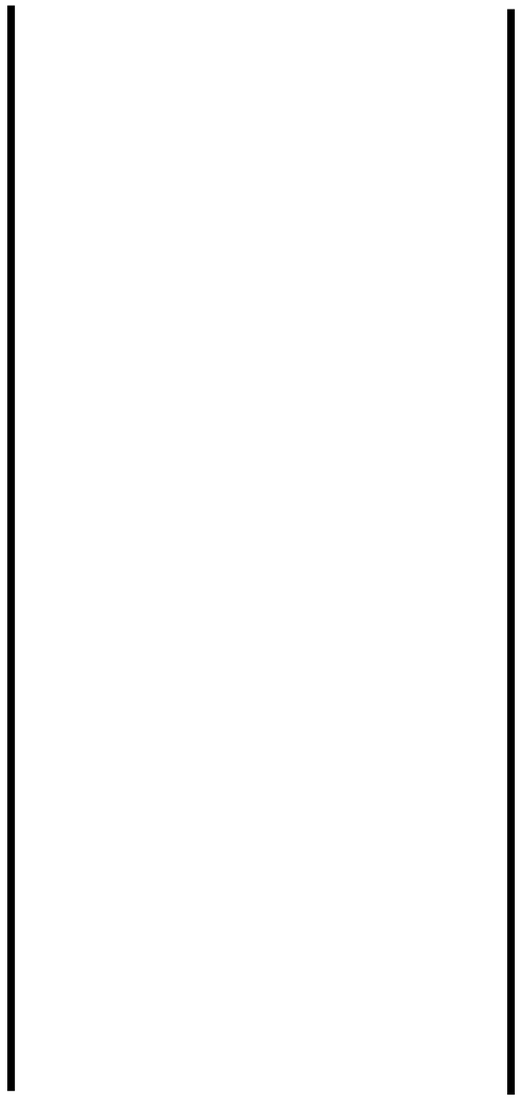
- HMap
- Non-HMap

The Future

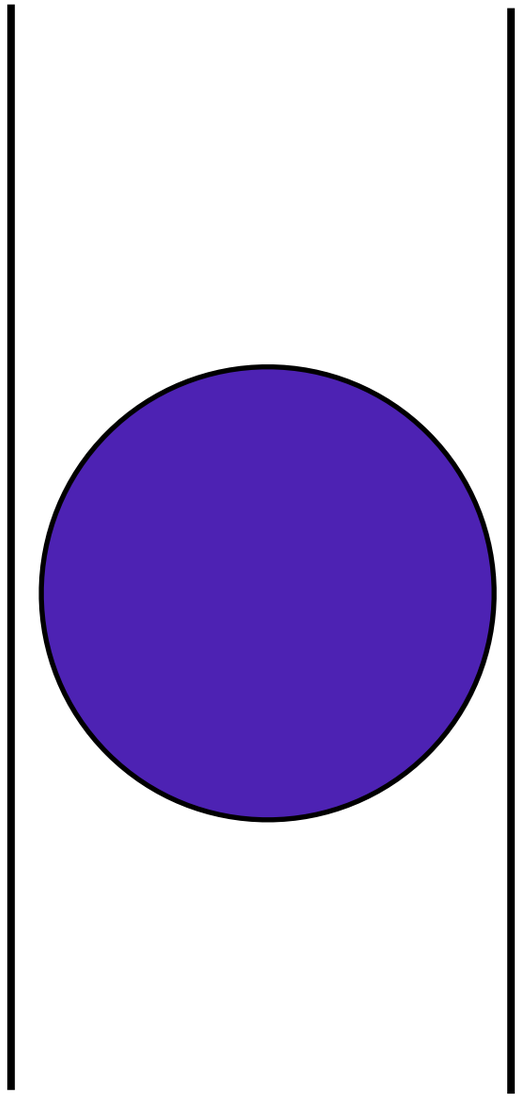
Typed ClojureScript

Towards Gradual Typing

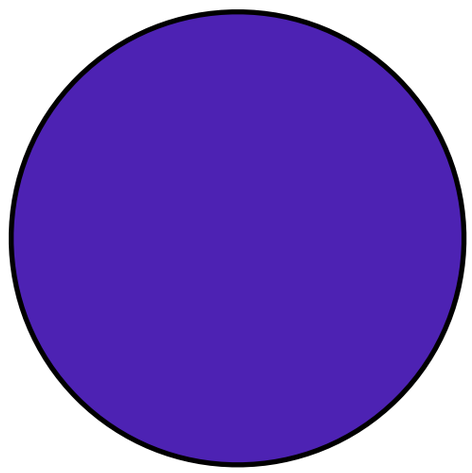
Towards Gradual Typing



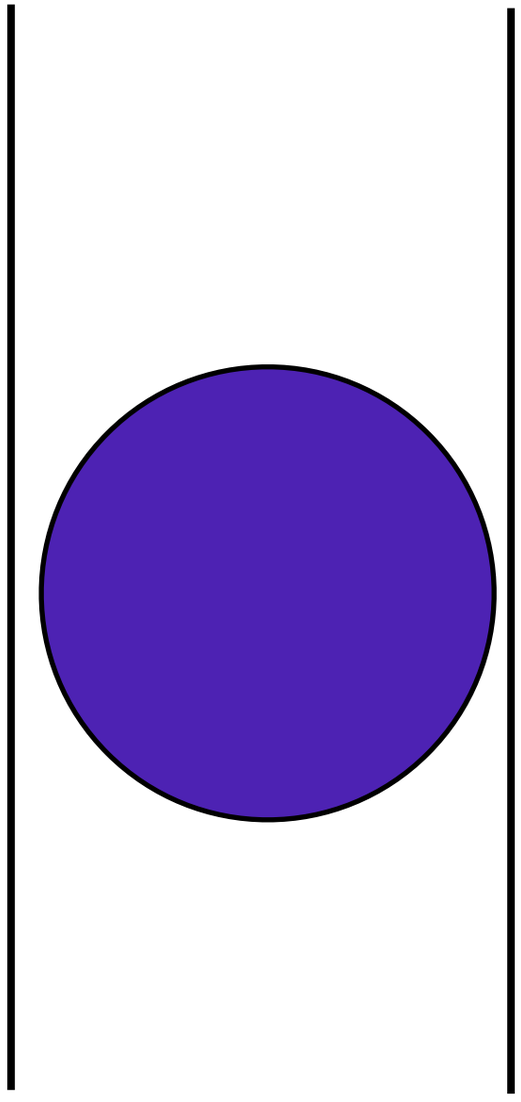
Towards Gradual Typing

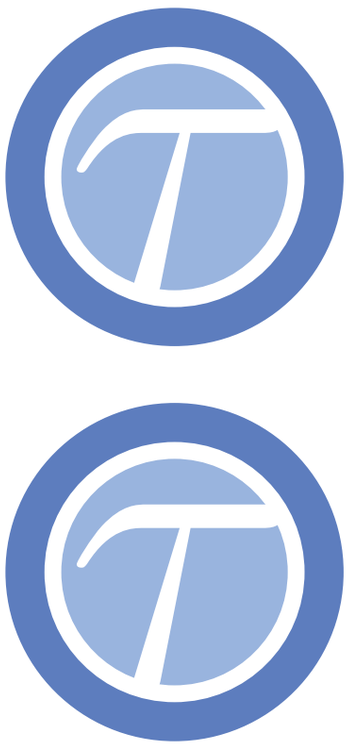


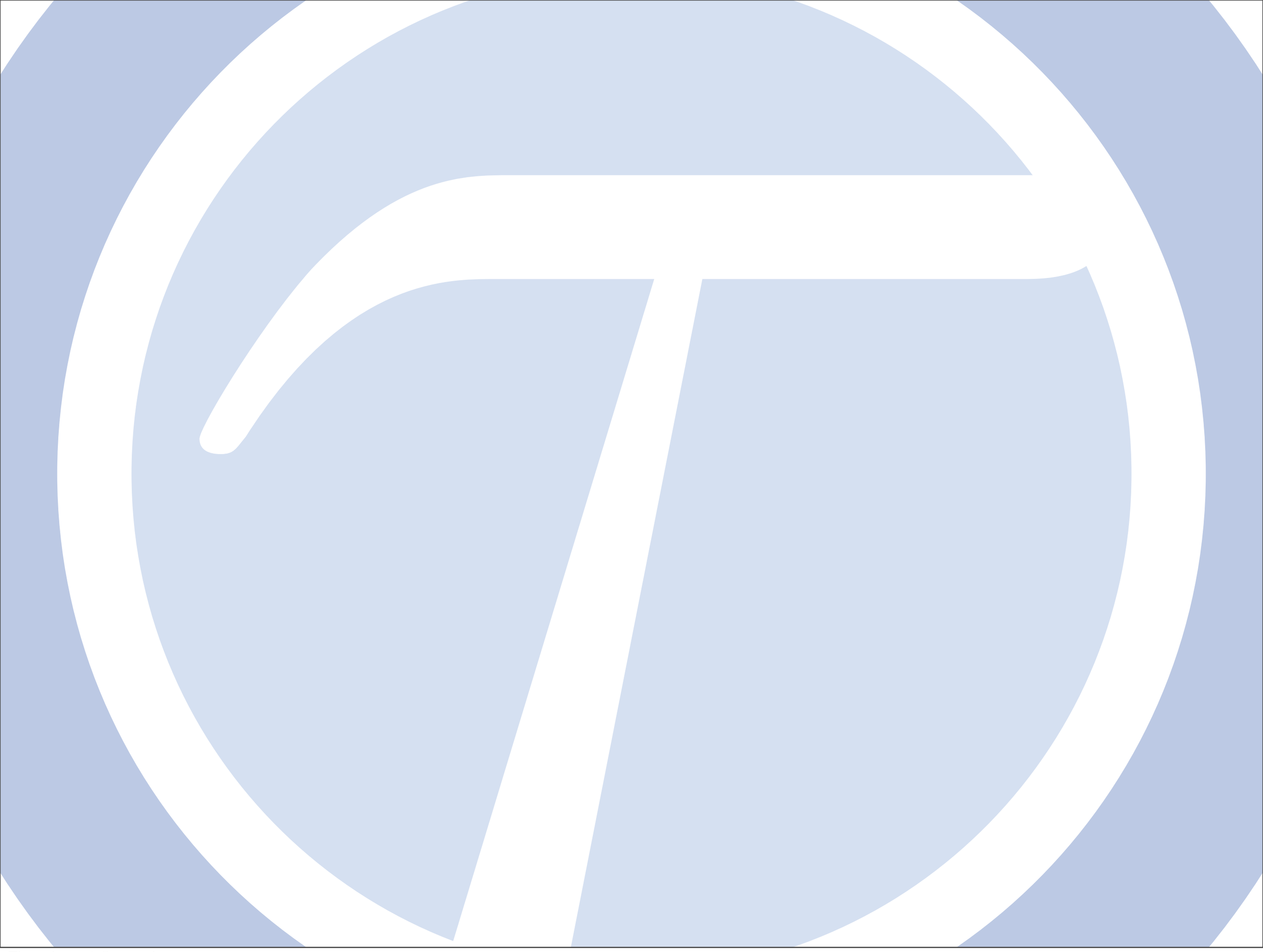
Towards Gradual Typing



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