Typed Clojure
An optional type system for Clojure
Gradual Typing

From Optional to Gradual Typing
Ambrose Bonnaire-Sergeant
Coming soon

We are here
What is Optional Typing?
Untyped

Reader → Analyzer → Emitter
Untyped

Reader → Analyzer → Emitter

Typed

Reader → Analyzer → Emitter

Type Checker
Untyped

Reader → Analyzer → Emitter

Typed

Reader → Analyzer → Emitter

Type Checker

Optional Types

Swap at will

Friday, 25 September 15
Untyped

Reader → Analyzer → Emitter

Optional Types

Swap at will

Runtime cannot depend on types

Typed

Reader → Analyzer → Emitter

Type Checker

Swap at will

Runtime cannot depend on types
Untyped

Reader → Analyzer → Emitter

Optional Types

Swap at will
Runtime cannot depend on types
Like a linter in practice

Typed

Reader → Analyzer → Emitter

Type Checker
Optional Types + Typed Clojure = Typed Clojure
(ann square [Int -> Int])
(defn square [a] (* a a))
Heterogeneous map type

(ann day ['{:day Int} -> Int])
(defn day [{d :day}] d)

Normal destructuring
(ann get-parent [File -> (U nil Str)]
(defn get-parent [f]
  (.getParent f))

Explicit nil/null

Java interop
(ann safe-inc [(U nil Int) -> Int])
(defn safe-inc [n]
  (if n
    (inc n)
    0))
(defalias Expr
    (U '{:op (Val :do), :exprs (Vec Expr)}
       '{:op (Val :val), :val Int}))

;; eg. {:op :do, :exprs [{:op :val, :val 1}]}
Arbitrary multimethod dispatch

(ann f [Expr -> Int])
(defmulti f :op)
(defmethod f :do [{exprs :exprs}]
  (apply + (map f exprs)))
(defmethod f :val [{val :val}]
  val)

(f {:op :do,
    :exprs [{:op :val, :val 30},
             {:op :val, :val 12}]})
=> 42
(defmulti f :op)
(defmethod f :do [{exprs :exprs}]
    (apply + (map f exprs)))
(defmethod f :val [{val :val}]
    val)
(f {:op :do,
    :exprs [{:op :val, :val 30},
        {:op :val, :val 12}]})
=> 42
(ann f [Expr -> Int])
(defmulti f :op)
(defmethod f :do [{exprs :exprs}]
  (apply + (map f exprs)))
(defmethod f :val [{val :val}]
  val)
(f {:op :do,
    :exprs [{:op :val, :val 30},
             {:op :val, :val 12}]})
=> 42
Arbitrary multimethod dispatch

(ann f [Expr -> Int])
(defmulti f :op)
(defmethod f :do [{exprs :exprs}]
  (apply + (map f exprs)))
(defmethod f :val [{val :val}]
  val)

(f {:op :do,
    :exprs [{:op :val, :val 30},
             {:op :val, :val 12}]})

;;=> 42
(ann f [Expr -> Int])
(defmulti f :op)
(defmethod f :do [{exprs :exprs}]
  (apply + (map f exprs)))
(defmethod f :val [{val :val}]
  val)

(f {:op :do,
    :exprs [{:op :val, :val 30},
            {:op :val, :val 12}]})

;=> 42
Arbitrary multimethod dispatch

(ann f [Expr -> Int])
(defmulti f :op)
(defmethod f :do [{exprs :exprs}]
  (apply + (map f exprs)))
(defmethod f :val [{val :val}]
  val)

(f {:op :do,
    :exprs [{:op :val, :val 30},
             {:op :val, :val 12}]})
=> 42
Arbitrary multimethod dispatch

(ann f [Expr -> Int])
(defmulti f :op)
(defmethod f :do [{exprs :exprs}]
 (apply + (map f exprs)))
(defmethod f :val [{val :val}]
  val)
(f {:op :do,
   :exprs [{:op :val, :val 30},
            {:op :val, :val 12}]})
=> 42
Optional Typing is all the rage
mypy

python
Typed Clojure in Practice
Why we’re supporting Typed Clojure, and you should too!

by circleci on September 27, 2013

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.

Why optional typing?

Dynamic languages have long been criticised for being hard to maintain at scale. When you grow to a large team or a large code base, it becomes more difficult to refactor a code base, to understand how it works, and to make sure it does what it should.
CircleCI Data

- 2 year trial
- 87 typed namespaces
- 105 Java interactions
- 328 HMap operations
- 11 multimethods, 89 defmethods
- 407 (22%) checked def’s, 1427 (78%) unchecked
Why we’re no longer using Core.typed

by Marc O’Morain on September 23, 2015

In September 2013 we blogged about why we’re supporting Typed Clojure, and you should tool. Now, 2 years later, our engineering team has made a collective decision to stop using Typed Clojure (specifically the core.typed library). As part of this decision, we wanted to write a blog-post about our experience using core.typed.

The reason that we decided to stop using core.typed was because we found that the cost of using it was greater than the benefit we gained. This is a subjective view, of course, so we will detail our reasoning below.

The core.typed library is part of the Typed Clojure project. It is a library that adds optional typing to Clojure code. Core.typed allows the developer to add type-annotations to Clojure forms, and then a type-checking process can be run to verify the type-information of your program.

First of all, some background on our project. The core for our main app contains 0.3 MB
Postmortem

- Slow type checking
- Incomplete support for Clojure idioms
- Missing third-party annotations
Postmortem

- Slow type checking
- Incomplete support for Clojure idioms
- Missing third-party annotations

407 (22%) checked def’s, 1427 (78%) unchecked
Pitch: Gradual Typing

Check the 22% at compile-time

Sanely handle interaction

Check the 78% at runtime

407 (22%) checked def’s, 1427 (78%) unchecked
What is Gradual Typing?
Gradual typing forces all code to respect static invariants
Optional Typing

```
(ann square [Int -> Int])
(defn square [a] (* a a))
```
(ann square [Int -> Int])
(defn square [a] (* a a))

(square 2)
;;=> 4

Exception:
NullPointerException

Optional Typing
(ann square [Int -> Int])
(defn square [a] (* a a))

(square 2)
;;=> 4

(square nil)
; ; Expected Int,
; ; found nil

Optional Typing
(ann square [Int -> Int])
(defn square [a] (* a a))

(square 2)
;;=> 4

(square nil)
;
; Expected Int,
; found nil

Optional Typing
(ann square [Int -> Int])
(defn square [a] (* a a))

(square 2)
;;=> 4

(square nil)
;; Expected Int,
;; found nil

(Optional Typing)

(square 2)
;;=> 4

(square nil)
;; Exception:
;; NullPointerException
;; ...

Friday, 25 September 15
(ann square [Int -> Int])
(defn square [a] (* a a))

(square 2)
; => 4

(square nil)
; Expected Int,
; found nil

(square 2)
; => 4

(square nil)
; Exception:
;  NullPoinerException
; ...

Optional Typing
Typed Clojure
An optional type system for Clojure
Gradual Typing
(ann square [Int -> Int])
(defn square [a] (* a a))

(square 2)
;;=> 4

(square 2)
;;=> 4

Gradual Typing
(ann square [Int -> Int])
(defn square [a] (* a a))

(square 2)
;;=> 4

(square nil)
;; Expected Int,
;; found nil

Gradual Typing
(ann square [Int -> Int])
(defn square [a] (* a a))

(square 2)
;; => 4

(square nil)
;; Expected Int,
;; found nil

Gradual Typing
(ann square [Int -> Int])
(defn square [a] (* a a))

(square 2)
;=> 4

(square nil)
Gradual Typing

```
(ann square [Int -> Int])
(defn square [a] (* a a))
```

```
(square 2)
;;=> 4
```

```
(square nil)
```

Compiles to:
```
(square (cast Int nil))
```
Optional Typing
Optional Typing

1 → 2 → 3 → 4

Depends on
Optional Typing

1

2 3

4

Depends on
Optional Typing

1 → 2 → 3 → 4

Depends on
Optional Typing

Typed invariants cannot be violated

Depends on
Depends on

Friday, 25 September 15
Gradual Typing

1

2 3

4

Depends on
Gradual Typing

 Depends on

1

2 3

4
Gradual Typing

Typed invariants cannot be violated

Depends on
Typed Racket

Language boundary mediator

Racket
Typed Racket

Racket
Typed Racket

Safe optimisations

Racket
Typed Racket

Blame

Racket
Typed Racket

Blame

Racket
Typed Racket

Blame

Racket
What’s done in Typed Clojure?
Typed Clojure
An optional type system for Clojure

STL 2014

Typed Clojure

Clojure
Typed Clojure
An optional type system for Clojure
Gradual Typing

Automatic type hints
Typed REPL
STL 2014
Typed REPL

(ns ^:core.typed my-ns)
...
my-ns=> (inc 1)
:- Long
2
(ns ^=:core.typed my-ns)
...
my-ns=> (inc 1)
:- Long
2
my-ns=> (inc nil)
Expected Number, found nil
in: (clojure.lang.Numbers/inc nil)

Found 1 error
my-ns=>
Enabling Typed REPL

project.clj:

... :repl-options
  {:nrepl-middleware
   [clojure.core.typed.repl/wrap-clj-repl]}
require+check w/ typed REPL

(ns
  my-inc-fail)

(inc nil)

; (require ‘my-inc-fail)
; NullPointerException
require+check w/ typed REPL

(ns my-inc-fail)
(inc nil)
; (require 'my-inc-fail)
; NullPointerException

(ns ^:core.typed my-inc-fail)
(inc nil)
; (require 'my-inc-fail)
; Type Error:
; Expected Num, given nil
Cache for free

;; Cached
(require 'my-ns)
Cache for free

;;; Cached
(require 'my-ns)

(load "my_ns")
Cache for free

;;; Cached
(require 'my-ns)

;;; Transitive deps cached

(load "my_ns")

Check via Typed REPL
Automatic type hints

(ns \textasciicircum:\texttt{core.typed} my-ns)

(defn get-parent [a : Any]
  {:pre [(instance? java.io.File a)]}
  (.getParent a))

Non-reflective via static types
Next steps
Typed Clojure
An optional type system for Clojure

Automatic type hints
Typed REPL
STL 2014

Typed REPL
Clojure
Typed Clojure
An optional type system for Clojure
Gradual Typing

Check typed exports
Check untyped imports
Better proxy story
Automatic type hints
Typed REPL
STL 2014
Typed Racket
Proxy problem

(deftype A [])

(proxy [A] [])

; CompilerException java.lang.VerifyError:
; Cannot inherit from final class
How to intercept methods?

(defprotocol IPoint
  (get-x [this])
  (get-y [this]))

(deftype Point [x y] IPoint
  (get-x [this] x)
  (get-y [this] y))

(proxy [Point] []
  (get-x [this]
    {:post [(integer? %)]}
    (get-x this))
  (get-y [this]
    {:post [(integer? %)]}
    (get-y this)))

; CompilerException java.lang.VerifyError:
; Cannot inherit from final class
Racket VM

Chaperones and Impersonators
Racket VM

Chaperones and Impersonators

Clojure

JVM
Check untyped imports

(ns ^:core.typed my-ns
 (:require [my-untyped :as u]
   [clojure.core.typed :as t]))
(t/import-untyped u/uinc [Int -> Int])
(u/uinc 41)

(ns my-untyped)
(defn uinc [n]
  “hello”)
(ns ^:core.typed my-ns
  (:require [my-untyped :as u]
    [clojure.core.typed :as t]))
(t/import-untyped u/uinc [Int -> Int])
(u/uinc 41)

(ns my-untyped)
(defn uinc [n]
  "hello")
Check untyped imports

(ns ^:core.typed my-ns
  (:require [my-untyped :as u]
    [clojure.core.typed :as t]))

(t/import-untyped u/uinc [Int -> Int])

(u/uinc 41)

(ns my-untyped)
(defn uinc [n]
  “hello”)

Blame
Typed Clojure
An optional type system for Clojure

STL 2014

Typed Clojure

Clojure
Typed Clojure

An optional type system for Clojure

Gradual Typing

Automatic type hints
Typed REPL
STL 2014

Typed Clojure

Clojure
Typed Clojure

An optional type system for Clojure

Gradual Typing

Typed REPL

Automatic type hints

Better proxy story

Check typed exports

Check untyped imports

STL 2014

Typed Clojure

Clojure

Friday, 25 September 15
Typed Clojure

Gradual Typing

Check typed exports
Check untyped imports
Better proxy story
Automatic type hints
Typed REPL
STL 2014

Typed Clojure

Clojure
Typed Clojure
An optional type system for Clojure

Gradual Typing for Clojure
Check typed exports
Check untyped imports
Better proxy story
Automatic type hints
Typed REPL
STL 2014

Thanks!

Friday, 25 September 15