Typed Clojure in Theory and Practice

Ambrose Bonnaire-Sergeant
What is Clojure?

A programming language
running on the Java Virtual Machine
What is Clojure?

*A programming language running on the Java Virtual Machine*

3% of JVM users’ primary language is Clojure

- [JVM Ecosystem Report 2018, snyk.io]
What is Clojure?

A programming language running on the Java Virtual Machine

3% of JVM users’ primary language is Clojure
- [JVM Ecosystem Report 2018, snyk.io]

1.1% of JVM users have adopted Clojure
- [The State of Java in 2018, baeldung.com]
General Purpose

Bar chart showing:
- Web development: 80%
- Open source projects: 50%
- Commercial services: 30%
- Enterprise apps: 20%
- Math / data analysis: 10%
- "Big Data": 5%
Survey: Why Clojure?

[State of Clojure 2019 Survey, Weighted average: 0 = Not Important, 1 = Important, 2 = Very Important]
Survey: Why Clojure?

- Functional programming: Weighted average = 1.7
- Immutability: Weighted average = 1.7
- The REPL: Weighted average = 1.6
- Ease of development: Weighted average = 1.4
- Host Interop (JVM/JS/CLR): Weighted average = 1.3

[State of Clojure 2019 Survey, Weighted average: 0 = Not Important, 1 = Important, 2 = Very Important]
Survey: Why Clojure?

- Functional programming: Very Important
- Immutability: Very Important
- The REPL: Very Important
- Ease of development: Very Important
- Host Interop (JVM/JS/CLR): Important

[State of Clojure 2019 Survey, Weighted average: 0 = Not Important, 1 = Important, 2 = Very Important]
Survey: Why Clojure?

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop (JVM/JS/CLR)

[State of Clojure 2019 Survey, Weighted average: 0 = Not Important, 1 = Important, 2 = Very Important]
Frustrations with Clojure

#2 Error messages

#4 Need better tools / IDEs

#11 No static typing

[State of Clojure 2019 Survey]
Frustrations with Clojure

#2 Error messages

#4 Need better tools / IDEs

#11 No static typing

My take
Clojure programmers need help specifying and verifying their programs

[State of Clojure 2019 Survey]
Typed Clojure

Typed Clojure is an *optional type system* for Clojure
Good Response to Typed Clojure

2012

Clojure

Google Summer of Code
$35,254 USD
728 backers

2013

INDIEGOGO

$8,621 USD
73 backers

2014

strangleup

2015

strangleup

2016

INDIEGOGO

$11,695 USD by 199 backers

2017

Clojure

My Research
How Typed Clojure works
How Typed Clojure works

1. Take an existing Clojure program

```clojure
(defn say-hello [to]
  (str “Hello, ” to))

(say-hello “world!”)
;=> “Hello, world!”
```
How Typed Clojure works

1. Take an existing Clojure program
2. Add type annotations

(defn say-hello [to]
  (str "Hello, " to))

(say-hello "world!")
;=> "Hello, world!"
How Typed Clojure works

1. Take an existing Clojure program
2. Add type annotations

(ann say-hello [Any -> String])
(defn say-hello [to]
  (str "Hello, " to))

(say-hello "world!")
;;=> "Hello, world!"
How Typed Clojure works

1. Take an existing Clojure program
2. Add type annotations
3. Use the type checker to verify Clojure programs (statically)

```
(ann say-hello [Any -> String])
(defn say-hello [to]
  (str "Hello, " to))

(say-hello "world!")
;;=> "Hello, world!"
```
How Typed Clojure works

1. Take an existing Clojure program
2. Add type annotations
3. Use the type checker to verify Clojure programs (statically)

```clojure
(ann say-hello [Any -> String])
(defn say-hello [to]
  (str "Hello, " to))

(say-hello "world!")
;; => "Hello, world!" : String
```
My Thesis Statement:

Typed Clojure is a **sound** and **practical** optional type system for Clojure
My Thesis Statement:

Typed Clojure is a **sound** and **practical** optional type system for Clojure.
My Thesis Statement:

Typed Clojure is a **sound** and **practical** optional type system for Clojure.

I created a new **sound** type system for Clojure.
My Thesis Statement:

Typed Clojure is a **sound** and **practical** optional type system for Clojure.

I show Typed Clojure’s features correspond to **real** programs.
My Thesis Statement: Typed Clojure is a **sound** and **practical** optional type system for Clojure.

I created a semi-automated workflow to port Clojure programs.
My Thesis Statement: Typed Clojure is a sound and practical type system for Clojure.

“Incomprehensible errors!” - Users

Typed Racket (prior work)

Typed Clojure

Automatic Annotations

Extensible Typing Rules

I demonstrate how to extend Typed Clojure to support custom rules

Prototype

Formalize+Sound

Design+Implement

Evaluation

Design+Implement

Formalize

Evaluation

Formalize+Sound

Design+Implement
My Thesis Statement:

Typed Clojure is a **sound** and **practical** optional type system.

“Check more programs!” - Users

I show how to mix symbolic execution with type checking.
Part I
Design and Evaluation of Typed Clojure
Published:
“Practical Optional Types for Clojure”, Ambrose Bonnaire-Sergeant, Rowan Davies, Sam Tobin-Hochstadt; ESOP 2016
Check with Typed Clojure
Simple Functions

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

(:x (point 1 2))
;;=> 1

(:y (point 1 2))
;;=> 2

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop
Simple Functions

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

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

(ann point [Int Int -> Point])

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

(:x (point 1 2))
=> 1

(:y (point 1 2))
=> 2
Simple Functions

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

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

(ann point [Int Int -> Point])

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

(:x (point 1 2))
=> 1

(:y (point 1 2))
=> 2
Higher-order functions

Scorecard
- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

(defn combine [p f]
  (f (:x p) (:y p)))

(combine (point 1 2) +)
;=> 3
(combine (point 1 2) str)
;=> "12"
Higher-order functions

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

(ann combine
  (All [a]
    [Point [Int Int -> a] -> a])
)(defn combine [p f]
  (f (:x p) (:y p)))

(combine (point 1 2) +)
;=> 3
(combine (point 1 2) str)
;=> "12"
Higher-order functions

Scorecard

- Functional programming
  
  - Immutability
  
  - The REPL
  
  - Ease of development
  
  - Host Interop

```
(ann combine
 (All [a]
  [Point [Int Int -> a] -> a]))
(defn combine [p f]
  (f (:x p) (:y p)))

(combine (point 1 2) +)
  => 3
(combine (point 1 2) str)
  => "12"
```
Type-Based Control flow

Scorecard
- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

(defn to-int [m]
  (if (string? m)
    (Integer/parseInt m)
    m))

(to-int 1)
;;=> 1
(to-int "2")
;;=> 2
Type-Based Control flow

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

(defn to-int [m]
  (if (string? m)
    (Integer/parseInt m)
    m))

(to-int 1)
  ;=> 1
(to-int "2")
  ;=> 2
Type-Based Control Flow

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

```
(ann to-int [(U Int Str) -> Int])
(defn to-int [m]
  (if (string? m)
    (Integer/parseInt m)
    m))
(to-int 1) ;=> 1
(to-int "2") ;=> 2
```
Type-Based Control flow

Scorecard
- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

(defn to-int [m]
  (if (string? m)
    (Integer/parseInt m)
    m))

(to-int 1) => 1
(to-int "2") => 2
Multimethods

Scorecard

- Functional programming

Immutability

The REPL

Ease of development

Host Interop

(defmulti to-int-mm class)
(defmethod to-int-mm String [m] (Integer/parseInt m))
(defmethod to-int-mm Number [m] m)

(to-int-mm 1) ;=> 1
(to-int-mm "2") ;=> 2
Multimethods

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

(defmulti to-int-mm (class))
(defmethod to-int-mm String [m]
  (Integer/parseInt m))
(defmethod to-int-mm Number [m] m)

(to-int-mm 1) ;=> 1
(to-int-mm "2") ;=> 2
Multimethods

(defmulti to-int-mm class)
(defmethod to-int-mm String [m]
  (Integer/parseInt m))
(defmethod to-int-mm Number [m] m)

(to-int-mm 1) ;=> 1
(to-int-mm "2") ;=> 2
Multimethods

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

(defmulti to-int-mm class)
(defmethod to-int-mm String [m]
  (Integer/parseInt m))
(defmethod to-int-mm Number [m] m)

(to-int-mm 1) ;=> 1
(to-int-mm "2") ;=> 2
Multimethods

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

(ann to-int-mm
 [(U Int Str) -> Int])

(defmulti to-int-mm class)
(defmethod to-int-mm String [m]
   (Integer/parseInt m))
(defmethod to-int-mm Number [m] m)

(to-int-mm 1) => 1
(to-int-mm "2") => 2
Multimethods

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

(ann to-int-mm
 [(U Int Str) -> Int])

defmulti to-int-mm class)
defmethod to-int-mm String [m]
   (Integer/parseInt m) Str
defmethod to-int-mm Number [m] m)

(to-int-mm 1) ;=> 1
(to-int-mm "2") ;=> 2
Multimethods

(ann to-int-mm
   [(U Int Str) -> Int])

(defmulti to-int-mm class)
(defmethod to-int-mm String [m] (Integer/parseInt m) Str)
(defmethod to-int-mm Number [m] m)

(to-int-mm 1) ;=> 1
(to-int-mm "2") ;=> 2
Formalism

1. Based on Occurrence Typing[1] (big-step semantics)
2. Add Typed Clojure features: HMaps, Multimethods
3. Add (some) Java Interop: Classes, Methods, Fields...

[1] ICFP ’10 - Tobin-Hochstadt, Felleisen
Type soundness

\[ \lambda T C \]

**Theorem** Well-typed programs don’t “go wrong”

**Corollary** Well-typed programs *don’t throw null-pointer exceptions*
Symbolic Execution

Extensible Typing Rules

Prototype

Typed Racket (prior work)

Typed Clojure

Automatic Annotations

Design+Implement

Formalize

Prototype

Evaluation

Formalize+Sound

Plantuml syntax:
```
@startuml
    Typed Racket (prior work) --> Typed Clojure
    Typed Clojure --> Automatic Annotations
    Automatic Annotations --> Extensible Typing Rules
    Extensible Typing Rules --> Symbolic Execution
    Evaluation --> Prototype
```

Typed Clojure

Automatic Annotations

Extensible Typing Rules

Prototype

Formalize

Prototype

Evaluation

Formalize+Sound

Design+Implement

Prototype

Evaluation

Formalize

Prototype

Design+Implement

Prototype
Empirical Evaluation of Typed Clojure

19k lines of Typed Clojure
Not Enough FP Support

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

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

```
(map (fn [p :- Point]
  (+ (:x p)
    (:y p)))
  [(point 1 2) (point 3 4)])
```
Not Enough FP Support

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

Required!

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

Required!

(map (fn [p :- Point]
    (+ (:x p)
      (:y p)))
  [(point 1 2) (point 3 4)])
Not Enough FP Support

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

Not Enough FP Support Required!

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

Not Enough FP Support Required!

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

Not Enough FP Support Required!

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

Not Enough FP Support Required!

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

Not Enough FP Support Required!
Global Annotation Burden

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop
Global Annotation Burden

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

(defalias Point
  '{:<x Int :<y Int})
(ann point [Int Int -> Point])
(ann combine
  (All [a]
   [Point [Int Int -> a] -> a]))
(ann extract-int
  ['{:value (U Int Str)} -> Int])
(ann extract-int-mm
  ['{:value (U Int Str)} -> Int])

Burden!
Global Annotation Burden

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

Burden!

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

(ann point [Int Int -> Point])

(ann combine
  (All [a]
   [Point [Int Int -> a] -> a]))

(ann extract-int
 [':{value (U Int Str)} -> Int])

(ann extract-int-mm
 [':{value (U Int Str)} -> Int])
Poor Errors with Macros

Scorecard
- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop
Poor Errors with Macros

Scorecard

- Functional programming

Immutability

The REPL

Ease of development

Host Interop

(inc nil)
Poor Errors with Macros

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

(inc nil)

Type Error:
Static method clojure.lang.Numbers/inc does not accept nil
Poor Errors with Macros

\[(\text{inc \ nil})\]

Type Error:
Static method clojure.lang.Numbers/inc does not accept nil

\(\text{Who??:}\)
Poor Errors with Macros

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

(inc nil) ; Expands to (Numbers/inc nil)

Type Error:
Static method clojure.lang.Numbers/inc does not accept nil

Who??


Poor Errors with Macros

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

(inc nil) ; Expands to (Numbers/inc nil)
Type Error:
Static method clojure.lang.Numbers/inc does not accept nil

Who??

(for [a [1 2 3]]
  (inc a))
Poor Errors with Macros

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

(inc nil) ; Expands to (Numbers/inc nil)

Type Error:
Static method clojure.lang.Numbers/inc does not accept nil

Who??

(for [a [1 2 3]]
  (inc a))

Type Error:
Static method clojure.lang.Numbers/inc does not accept Any
Poor Errors with Macros

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

(inc nil) ; Expands to (Numbers/inc nil)
Type Error:
Static method clojure.lang.Numbers/inc does not accept nil

Who??

(for [a [1 2 3]]
  (inc a))
Type Error:
Static method clojure.lang.Numbers/inc does not accept Any

Huh? But it’s an Int...
Poor Errors with Macros

Scorecard

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

(inc nil) ; Expands to (Numbers/inc nil)
Type Error:
Static method clojure.lang.Numbers/inc does not accept nil

X Who??

(for [a [1 2 3]]
  (inc a))
Type Error:
Static method clojure.lang.Numbers/inc does not accept Any

X Huh? But it’s an Int...

(t/for [a :- t/Int, [1 2 3]]
  (inc a))
Poor Errors with Macros

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

(inc nil) ; Expands to (Numbers/inc nil)
Type Error:
Static method clojure.lang.Numbers/inc does not accept nil

Who??

(for [a [1 2 3]]
  (inc a))
Type Error:
Static method clojure.lang.Numbers/inc does not accept Any

Huh? But it’s an Int...

(t/for [a :- t/Int, [1 2 3]]
  (inc a))

How was I supposed to know about t/for?
Poor Errors with Macros

**Scorecard**

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop

---

(inc nil) ; Expands to (Numbers/inc nil)
Type Error:
Static method clojure.lang.Numbers/inc does not accept nil

Who??

(for [a [1 2 3]]
  (inc a))
Type Error:
Static method clojure.lang.Numbers/inc does not accept Any

Huh? But it’s an Int...

(t/for [a :- t/Int, [1 2 3]]
  (inc a))
How was I supposed to know about t/for?
Scorecard: Typed Clojure’s initial design

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop
Scorecard: Typed Clojure’s initial design

- Functional programming
- Immutability
- The REPL
- Ease of development
- Host Interop
Scorecard: Typed Clojure’s initial design

- Functional programming: Yes
- Immutability: Yes
- The REPL: No
- Ease of development: No
- Host Interop: Yes
Scorecard: Typed Clojure’s initial design

- Functional programming:
  - Yes
  - No
- Immutability:
  - Yes
- The REPL:
  - No
- Ease of development:
  - No
- Host Interop:
  - Yes
Scorecard: Typed Clojure’s initial design

- Functional programming: ✔
- Immutability: ✔
- The REPL: ✗
- Ease of development: ✗
- Host Interop: ✔

Typed Racket (prior work) ⊂ Typed Clojure ⊂ Automatic Annotations

“Annotation burden!”
Scorecard: Typed Clojure’s initial design

- Functional programming: ✔️
- Immutability: ✗
- The REPL: ✗
- Ease of development: ✔️

Typed Racket (prior work)

Typed Clojure

Automatic Annotations

Extensible Typing Rules

“Annotation burden!”

“Incomprehensible errors!”

(“Annotation burden!” and “Incomprehensible errors!” are quotes from the text content.)
Scorecard: Typed Clojure’s initial design

- Functional programming: ✔️ (prior work: ✗)
- Immutability: ✔️
- The REPL: ✗
- Ease of development: ✔️
- Host Interop: ✔️

- Typed Racket
- Typed Clojure
- Automatic Annotations
- Extensible Typing Rules
- Symbolic Execution

- “Annotation burden!”
- “Incomprehensible errors!”
- “Check more programs!”
Part II
Automatic Annotations
In submission:
“Squash the work: A Workflow for Typing Untyped Programs that use Ad-Hoc Data Structures”, Ambrose Bonnaire-Sergeant, Sam Tobin-Hochstadt
Annotation burden

(defalias Point
  '(:x Int :y Int))

(ann point [Int Int -> Point])

(ann extract-int
  ['{:value (U Int Str)} -> Int])

(ann combine
  (All [a]
      [Point [Int Int -> a] -> a]))

(ann extract-int-mm
  ['{:value (U Int Str)} -> Int])
Annotation burden

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

(ann combine
  (All [a]
    [Point [Int Int -> a] -> a]))

(ann extract-int
  ['{:value (U Int Str)} -> Int])
(ann extract-int-mm
  ['{:value (U Int Str)} -> Int])
(def forty-two 42)       Tool design
Tool design

\[ \Gamma = \{\text{forty-two} : \text{Long}\} \]
(def forty-two 42)

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

Instrument

Collection Phase

Tool design

\( \Gamma = \{\text{forty-two} : \text{Long}\} \)
Tool design

\[ \Gamma = \{\text{forty-two} : \text{Long}\} \]
Tool design

\[ \Gamma = \{\text{forty-two} : \text{Long}\} \]
(def forty-two 42)

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

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

\( \Gamma = \{ \text{forty-two} : \text{Long} \} \)

\( \Gamma_0 \)

\( \Gamma_1 \)

Inference Phase

Global “Squashing”

Inference Phase

Local “Squashing”

Inference Phase

Naive Translation

Instrument

Collection Phase

Track

Collection Phase

(def forty-two 42)

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

; Inference result:
; ['forty-two] : Long

(def forty-two 42)
Porting workflow

... Auto-generate annotations
Porting workflow

... Auto-generate annotations

Type check with Typed Clojure
Porting workflow

Auto-generate annotations

Type check with Typed Clojure

Type error?

Manually fix according to error message
Porting workflow

Auto-generate annotations

Type check with Typed Clojure

Type error?

Manually fix according to error message
Porting workflow

Auto-generate annotations

Type check with Typed Clojure

Type checks?

Done

Manually fix according to error message

Type error?
\( \lambda \text{track} \)

\[
\text{annotate} : e, \overline{x} \rightarrow \Delta
\]

\text{annotate} = \text{infer} \circ \text{collect}
\lambda \text{track}

\text{annotate} : e, \overline{x} \rightarrow \Delta

\text{annotate} = \text{infer} \circ \text{collect}

“Track and annotate x’s in program e”
\lambda track
\[
\text{define } f = \lambda m. (\text{get } m : a)
\]
\[ \text{track} \]

define \( f = \lambda m. (\text{get } m :a) \)

\( (f \{ :a \ 42 \}) \Rightarrow 42 \)
\[ \lambda \text{track} \]

**Definition**

\[
\text{define } f = \lambda m. (\text{get } m : a) \\
(f \{a \ 42\}) \Rightarrow 42
\]

**Test**

\[
\text{annotate}((f \{a \ 42\}), [f]) = \{f : [\{a \ N\} \rightarrow N]\}
\]
\( \lambda \text{track} \)

\[
\begin{align*}
\text{define } f &= \lambda m.(\text{get } m :a) \\
(f \ {:a \ 42}) &= \Rightarrow 42
\end{align*}
\]

annotate((f \ {:a \ 42}), [f]) = \{ f : [{:a N} \rightarrow N] \}
\[
\lambda\text{track}
\]

Define \(f = \lambda m. (\text{get } m : a)\)

\((f \{ :a \ 42 \}) \Rightarrow 42\)

\[
\text{annotate}\((f \{ :a \ 42 \}), [f]) = \{ f : [\{ :a \ N \} \rightarrow \ N] \}\]

Test Track-me
\[\text{define } f = \lambda m. (\text{get } m : a)\]

\[(f \ {::a \ 42}) \Rightarrow 42\]

\[\text{annotate}((f \ {::a \ 42}), [f]) = \{ f : \{::a \ N \} \to N \}\]
Intentionally unsound

Aggressively combines types to create compact aliases and recursive types

Tailored for the workflow
Symbolic Execution
Ω
Extensible Typing Rules
Typed Racket (prior work)
Typed Clojure
Automatic Annotations
Evaluation
Formalize+SOUND
Design+Implement
Prototype
Symbolic Execution
Prototype
Typing
Formalize+Sound
Design+Implement
Prototype
Evaluation
Design+Implement
Evaluation

Ported 5 open-source programs (~1500 LOC)

Measured the kinds of manual changes needed
Auto-generated types

(\texttt{ann mult [Int Int :\rightarrow\ Int]})
Auto-generated types

Manual changes

(ann mult [Int Int :-> Int])

(ann mult [Int * :-> Int])
Auto-generated types

Manually changed

Auto-generated
types

(ann mult [Int Int :--> Int])

(ann mult [Int * :--> Int])

(ann initial-perm-numbers [(Map Int Int) :--> (Coll Int)]]
Auto-generated types

(ann mult [Int Int :-> Int])

Manual changes

(ann mult [Int * :-> Int])

Auto-generated types

(ann initial-perm-numbers [(Map Int Int) :-> (Coll Int)])

Manual changes

(ann initial-perm-numbers [(Map Any Int) :-> (Coll Int)])
Has an interesting type

(defn parse-exp [e]
  (cond
    (symbol? e) {:E :var, :name e}
    (false? e) {:E :false}
    (= 'n? e) {:E :n?}
    ... ...)
  ...
  ...
  ... ...))
(defalías E
  (U

'{:E ':app, :args (Vec E), :fun E}
'{:E ':false}
'{:E ':if, :else E, :test E, :then E}
'{:E ':lambda, :arg Sym, :arg-type T, :body E}
'{:E ':var, :name Sym}))

Has an interesting type

Auto-generated types

(ann parse-exp [Any :-> E])
(defn parse-exp [e]
  (cond
    (symbol? e) {:E :var, :name e}
    (false? e) {:E :false}
    (= 'n? e) {:E :n?}
    ... ... ... ...))
Manual changes

Auto-generated types

Has an interesting type

(defalies E
 (U
  '{:E :add1}
  '{:E :n?}
  {:E :app, :args (Vec E), :fun E}
  '{:E :false}
  '{:E :if, :else E, :test E, :then E}
  '{:E :lambda, :arg Sym, :arg-type T, :body E}
  '{:E :var, :name Sym}))

(ann parse-exp [Any :-> E])

(defn parse-exp [e]
 (cond
   (symbol? e) {:E :var, :name e}
   (false? e) {:E :false}
   (= 'n? e) {:E :n?}
   ... ...))
Manual effort

Mostly deleting/upcasting types

Adding missing cases to (generated) recursive types
Scorecard

- Functional programming: ✔️
- Immutability: ✔️
- The REPL: ❌
- Ease of development: ❌
- Host Interop: ✔️

“Annotation burden!”
Scorecard

Automatic annotations make porting Clojure programs easier

“Annotation burden!”
Part III
Extensible Typing Rules
Symbolic Execution

Typed Clojure

Automatic Annotations

Extensible Typing Rules

Typed Racket (prior work)

Formalize+Sound

Design+Implement

Evaluation

Formalize+Sound

Design+Implement

Prototype

Prototype

"Incomprehensible errors!"
Problem

(for [a [1 2 3]]
  (inc a))
Problem

(for [a [1 2 3]]
  (inc a))

Type Error:
Static method clojure.lang.Numbers/inc does not accept Any
Problem

How to propagate type information?

(for [a [1 2 3]]
  (inc a))

Type Error:
Static method clojure.lang.Numbers/inc does not accept Any
Idea

(for [a [1 2 3]]
  (inc a))
Allow the user to define custom typing rules for macros
Roadblock:
Expansion comes *before* check

... → Fully expand → Type check → Run
Roadblock:
Expansion comes *before* check

```
<table>
<thead>
<tr>
<th>Time</th>
<th>(let [..])</th>
<th>(cond ...)</th>
<th>(+ ..))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>analyze^</td>
<td>analyze^</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>analyze^</td>
<td>analyze^</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>analyze&lt;</td>
<td>analyze&lt;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>pre-passes^</td>
<td>pre-passes&lt;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>pre-passes&lt;</td>
<td>pre-passes&lt;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>post-passes&lt;</td>
<td>post-passes&lt;</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>check^</td>
<td>check^</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>check&lt;</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>check&lt;</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>check&lt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Roadblock:
Expansion comes *before* check

... 

Fully expand 

Type check 

Run 

<table>
<thead>
<tr>
<th>Time</th>
<th>(let [...])</th>
<th>(cond ...)</th>
<th>(+ ...))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>analyze⇒</td>
<td>analyze⇒</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>analyze≤</td>
<td>analyze≤</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>pre-passes⇒</td>
<td>pre-passes⇒</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>analyze≤</td>
<td>analyze≤</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>pre-passes⇒</td>
<td>pre-passes⇒</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>post-passes≤</td>
<td>post-passes≤</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>check⇒</td>
<td>check⇒</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>check≤</td>
<td>check≤</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>check≤</td>
<td>check≤</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>check≤</td>
<td>check≤</td>
<td></td>
</tr>
</tbody>
</table>

Already expanded!
Solution

Allow Typed Clojure to interleave macroexpansion and type checking
Checker controls expansion

Expand → Type check → Run

…
I wrote a new
Clojure code analyzer

<table>
<thead>
<tr>
<th>Time</th>
<th>(let [...])</th>
<th>(cond ...)</th>
<th>(+ ...))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>unanalyzed&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>analyze-outer*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>run-pre-passes&gt;</td>
<td>analyze-outer*</td>
<td>run-pre-passes&gt;</td>
</tr>
<tr>
<td>3</td>
<td>check&gt;</td>
<td>run-pre-passes&gt;</td>
<td>check&gt;</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>run-post-passes&lt;</td>
<td>check&lt;</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>run-post-passes&lt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>check&lt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This was non-trivial

Must also interleave *evaluation*

Maintains correct lexical scope

Interacts with Clojure’s type hinting system
Example type checker with new analyzer

```
(defn check-expr
  "Check an AST node has the expected type."
  [expr expected]
  (if (= :unanalyzed (:op expr))
    (case <resolved-op-sym-for-expr>
      clojure.core/cond (check-special-cond expr expected)
      ; default case
      (check-expr (analyze-outer expr) expected))
  (run-post-passes
    (check (run-pre-passes expr)
      expected)))
```
Example type checker with new analyzer

```
(defn check-expr
  "Check an AST node has the expected type."
  [expr expected]
  (if (= :unanalyzed (:op expr))
    (case <resolved-op-sym-for-expr>
      clojure.core/cond (check-special-cond expr expected)
      ; default case
      (check-expr (analyze-outer expr) expected))
    (run-post-passes
      (check (run-pre-passes expr) expected)))))
```
Example type checker with new analyzer

If partially expanded...

Custom rules

```
(defn check-expr
  "Check an AST node has the expected type."
  [expr expected]
  (if (= :unanalyzed (:op expr))
    (case <resolved-op-sym-for-expr>
      clojure.core/cond (check-special-cond expr expected)
      ; default case
      (check-expr (analyze-outer expr) expected)
    (run-post-passes
      (check (run-pre-passes expr) expected))))
```
Scorecard

- Functional programming: ✔️❌
- Immutability: ✔️
- The REPL: ❌
- Ease of development: ❌
- Host Interop: ✔️

“Incomprehensible errors!”
Scorecard

- Functional programming: ✔️ ✗
- Immutability: ✔️
- The REPL: ✗
- Ease of development: ✗
- Host Interop: ✔️

"Incomprehensible errors!"

Extensible rules Prototype:
Improve errors, check more programs
Part VI
Symbolic Execution
Symbolic Execution

Typed Racket (prior work)
Typed Clojure
Automatic Annotations
Extensible Typing Rules

Prototype
Prototype

“Check more programs!”
Goal: Reduce local annotations

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

(map (fn [p :- Point]
  (+ (:x p)
     (:y p)))
  [(point 1 2) (point 3 4)])
Goal: Reduce local annotations

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

(map (fn [p :- Point]
  (+ (:x p)
     (:y p)))
  [(point 1 2) (point 3 4)])
Goal: Reduce local annotations

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

(map (fn [p :- Point]
        (+ (:x p)
           (:y p)))
      [(point 1 2) (point 3 4)])
Setting: Bidirectional Checking

(let [f (fn [x :? x] x)]
  (f 1))

(map (fn [p :? p]
          (+ (:x p)
             (:y p)))
     [(point 1 2) (point 3 4)])
Setting: Bidirectional Checking

_Type checking proceeds outside-in_

```lisp
(let [f (fn [x :- ???] x)]
  (f 1))
```

```lisp
(map (fn [p :- ?????] 
       (+ (:x p) 
          (:y p)))
     [(point 1 2) (point 3 4)])
```
Setting: Bidirectional Checking

Type checking proceeds outside-in

```
(let [f (fn [x :- ???] x)]
  (f 1))
```

Must have type of x here

```
(map (fn [p :- ????] [
  (+ (:x p) (:y p))
[(point 1 2) (point 3 4)]])
```
Setting: Bidirectional Checking

**Type checking proceeds outside-in**

```clojure
(let [f (fn [x :? x] x)]
  (f 1))
```

*Must have type of x here*

```clojure
(map (fn [p :??] (+ (:x p) (:y p)))
  [(point 1 2) (point 3 4)])
```

*Must have type of p here*
(let [f (fn [x :? x] x)]
  (f 1))

(map (fn [p :? p]
  (+ (:x p)
    (:y p))
  [(point 1 2) (point 3 4)])
Intuition

(let [f (fn [x :- ???] x)]
  (f 1))

(map (fn [p :- ?????] 
    (+ (:x p) 
        (:y p)))
  [(point 1 2) (point 3 4)])
Intuition

(let [f (fn [x :- ???] x)]
 (f 1))

(map (fn [p :- ?????] 
 (+ (:x p)
    (:y p)))
 [(point 1 2) (point 3 4)])
Intuition

(let [f (fn [x :? x] x)]
  (f 1))

(map (fn [p :? p]
         (+ (:x p)
            (:y p)))
     [(point 1 2) (point 3 4)])
Approach

New type rule for checking (unannotated) functions:

```plaintext
(let
  [f (fn [x] x)]
; f : ?????????
(f 1))
```
New type rule for checking (unannotated) functions:

```
(let [f (fn [x] x)]
  ; f : (fn [x] x)
  (f 1))
```

The type of a function is its **code**
New type rule for checking (unannotated) functions:

\[
\begin{align*}
\text{(let } [f (fn \ [x] \ x)] & \\
& ; \ f : \forall (fn \ [x] \ x) \ \\
(f 1))\end{align*}
\]

The type of a function is its code...
...and the type environment it was “defined” at
Approach

New type rule for checking (unannotated) functions:

```
(let [f (fn [x] x)]
  ; f : Γ @(fn [x] x)
  (f 1))
```

Symbolic Closure Types

Resembles runtime closures, except executed *symbolically*
Approach

(let [f (fn [x] x)]
    ; f : f @(fn [x] x)
    (f 1))

Application rule?
Approach

(let [f (fn [x] x)]
  ; f : @(fn [x] x)
  (f 1))
Tradeoffs

Undecidable in general

However, many local functions are only used once and are non-recursive

Can rely on top-level annotations to drive the symbolic execution
Symbolic Execution

Typed Clojure

Automatic Annotations

Extensible Typing Rules

Typed Racket (prior work)

Design+Implement

Formalize+Sound

Evaluation

Prototype

Prototype

Formalize

Design+Implement

Evaluation

Prototype
Naive formalism

\[
\begin{align*}
\text{UAbs} & \quad \text{UApp} \\
\Gamma \vdash \lambda(x)f : \Gamma \odot \lambda(x)f & \quad \Gamma' \vdash e_1 : \Gamma \odot \lambda(x)f \quad \Gamma' \vdash e_2 : \sigma \\
& \quad \Gamma, x:\sigma \vdash f : \tau \\
& \quad \Gamma' \vdash e_1(e_2) : \tau
\end{align*}
\]
Naive formalism

\[\begin{align*}
\text{UAbs} & \quad \Gamma \vdash \lambda(x)f : \Gamma \circ @ \lambda(x)f \\
\text{UApp} & \quad \Gamma' \vdash e_1 : \Gamma \circ @ \lambda(x)f \quad \Gamma' \vdash e_2 : \sigma \\
\quad & \quad \Gamma, x:\sigma \vdash f : \tau \\
& \quad \Gamma' \vdash e_1(e_2) : \tau
\end{align*}\]
Naive formalism

\[
\begin{align*}
\text{UApp} & \\
\Gamma' \vdash e_1 : \Gamma \odot \lambda(x)f & \quad \Gamma' \vdash e_2 : \sigma \\
\Gamma, x:\sigma \vdash f : \tau & \\
\implies \Gamma' \vdash e_1(e_2) : \tau
\end{align*}
\]
Naive formalism

\[
\text{UAbs} \quad \frac{\Gamma \vdash \lambda(x)f : \Gamma @ \lambda(x)f}{\Gamma \vdash \lambda(x)f : \Gamma @ \lambda(x)f} \\
\text{UApp} \quad \frac{\Gamma' \vdash e_1 : \Gamma @ \lambda(x)f \quad \Gamma' \vdash e_2 : \sigma}{\Gamma' \vdash e_1(e_2) : \tau}
\]
Naive formalism

\[
\text{UAbs} \\
\frac{\Gamma \vdash \lambda(x)f : \Gamma \circ \lambda(x)f}{\Gamma \vdash \lambda(x)f : \Gamma \circ \lambda(x)f}
\]

\[
\text{UApp} \\
\frac{\Gamma' \vdash e_1 : \Gamma \circ \lambda(x)f \quad \Gamma' \vdash e_2 : \sigma}{\Gamma', x : \sigma \vdash f : \tau}
\frac{\Gamma' \vdash e_1 : \Gamma \circ \lambda(x)f \quad \Gamma' \vdash e_2 : \sigma}{\Gamma' \vdash e_1(e_2) : \tau}
\]
Naive formalism

\[
\begin{align*}
\text{UApp} & \quad \Gamma' \vdash e_1 : \Gamma @ \lambda(x) f \\
\text{UAbs} & \quad \Gamma' \vdash e_2 : \sigma \\
& \quad \Gamma, x : \sigma \mid f : \tau \\
& \quad \Gamma' \vdash e_1(e_2) : \tau
\end{align*}
\]
Prototype Implementation
Prototype Implementation

\[(\text{tc} \ ? \ 1)\]

\[\Rightarrow \text{Int}\]
Prototype Implementation

(tc ? 1)
=> Int

(tc [Int :-> Int] (fn [x] x))
=> [Int :-> Int]
Prototype Implementation

\[(tc \ ? \ 1)\]
\[\Rightarrow \ Int\]

\[(tc \ [Int \ :-\ > \ Int] \ (fn \ [x] \ x))\]
\[\Rightarrow \ [Int \ :-\ > \ Int]\]
Prototype Implementation

\[(tc \ ? \ 1)\]
\[\Rightarrow \ Int\]

\[(tc \ [Int \ :-\ > \ Int] \ (fn \ [x] \ x))\]
\[\Rightarrow \ [Int \ :-\ > \ Int]\]
Prototype Implementation

\[
(tc ? 1) \\
\Rightarrow \text{Int}
\]

\[
(tc \ [\text{Int} :\!\to\! \text{Int}] \ (fn \ [x] \ x)) \\
\Rightarrow \ [\text{Int} :\!\to\! \text{Int}]
\]

\[
(tc ? (fn \ [x] \ x)) \\
\Rightarrow \ (\text{Closure} \ \{\} \ (fn \ [x] \ x))
\]
Prototype Implementation

\[
(tc \ ? \ 1) \\
\Rightarrow \text{Int}
\]

\[
(tc \ [\text{Int} :\text{->} \text{Int}] \ (\text{fn } [\text{x}] \text{ x})) \\
\Rightarrow [\text{Int} :\text{->} \text{Int}]
\]

\[
(tc \ ? \ (\text{fn } [\text{x}] \text{ x})) \\
\Rightarrow (\text{Closure} \ {} \ (\text{fn } [\text{x}] \text{ x}))
\]

\[
(tc \ ? \ ((\text{fn } [\text{x}] \text{ x}) \ 1)) \\
\Rightarrow \text{Int}
\]
Prototype Implementation

\[
(tc \ ? \ 1) \\
\Rightarrow \text{Int}
\]

\[
(tc \ [\text{Int} \ :\to \ \text{Int}] \ (fn \ [x] \ x)) \\
\Rightarrow \ [\text{Int} \ :\to \ \text{Int}]
\]

\[
(tc \ ? \ (fn \ [x] \ x)) \\
\Rightarrow \ (\text{Closure} \ {} \ (fn \ [x] \ x))
\]

\[
(tc \ ? \ ((fn \ [x] \ x) \ 1)) \\
\Rightarrow \text{Int}
\]
Prototype Implementation

$$(tc ? 1)$$
$$\Rightarrow \text{Int}$$

$$(tc \ [\text{Int} \rightarrow \text{Int}] \ (fn \ [x] \ x))$$
$$\Rightarrow [\text{Int} \rightarrow \text{Int}]$$

$$(tc \ ? \ (fn \ [x] \ x))$$
$$\Rightarrow (\text{Closure} \ {} \ (fn \ [x] \ x))$$

$$(tc \ ? \ ((fn \ [x] \ x) \ 1))$$
$$\Rightarrow \text{Int}$$
Prototype Implementation

\[(\text{tc} \ ? \ (\text{map} \ (\text{fn} \ [x] \ x) \ [1 \ 2 \ 3])))\]
\[\Rightarrow (\text{Seq} \ \text{Int})\]
Prototype Implementation

\[(\text{tc} \ ? \ (\text{map} \ (\text{fn} \ [x] \ x) \ [1 \ 2 \ 3])) \Rightarrow (\text{Seq} \ \text{Int})\]
Prototype Implementation

```
(tc ? (map (fn [x] x) [1 2 3]))
=> (Seq Int)
```
Prototype Implementation

(tc ? (map (fn [x] x) [1 2 3]))
=> (Seq Int)

(tc ? (map (comp (fn [x] x)
(fn [y] y))
[1 2 3]))
=> (Seq Int)
Prototype Implementation

```
(tc ? (map (fn [x] x) [1 2 3]))
=> (Seq Int)
```

```
(tc ? (map (comp (fn [x] x) (fn [y] y)) [1 2 3]))
=> (Seq Int)
```
Prototype Implementation

(tc ? (map (fn [x] x) [1 2 3]))
=> (Seq Int)

(tc ? (map (comp (fn [x] x) 
             (fn [y] y)))
     [1 2 3]))
=> (Seq Int)
Prototype Implementation

```
(tc ? (map (fn [x] x) [1 2 3]))
=> (Seq Int)
```

```
(tc ? (map (comp (fn [x] x) (fn [y] y))) [1 2 3]))
=> (Seq Int)
```
Prototype Implementation

GR is an *untypable*[^1] strongly normalizing term of System F

[^1]: LICS’88, Giannini & Rocca
Prototype Implementation

GR is an **untypable**\[1\] strongly normalizing term of System F

Evaluating it in plain Clojure, it’s just quirky identity function

\[
\text{GR} (\text{fn} \ [\_] (\text{fn} \ [\_] \ 42)) \quad \Rightarrow \ 42
\]

\[
\text{GR} (\text{fn} \ [\_] (\text{fn} \ [\_] \ \text{“hello”})) \quad \Rightarrow \ \text{“hello”}
\]

\[1\] LICS’88, Giannini & Rocca
Prototype Implementation

GR is an **untypable**[1] strongly normalizing term of System F

Evaluating it in plain Clojure, it’s just quirky identity function

\[
\text{GR (fn [__] (fn [__] 42))) ;=} 42
\]
\[
\text{GR (fn [__] (fn [__] “hello”))) ;=} “hello”
\]

**Challenge**: Type check this quirky identity function

\[
\text{(ann id (All [a] [a -> a]))}
\]
\[
\text{(defn id [x]
  (GR (fn [__] (fn [__] x))))}
\]

[1] LICS’88, Giannini & Rocca
Prototype Implementation

Symbolic closures let us treat GR as a **black box** until it is executed symbolically

```
(let [I (fn [a] a)
      K (fn [b] (fn [c] b))
      D (fn [d] (d d))]
  ((fn [x] (fn [y] ((y (x I))
                      (x K)))))
D)
```
Prototype Implementation

Symbolic closures let us treat GR as a **black box** until it is executed symbolically.
Prototype Implementation

Symbolic closures let us treat GR as a black box until it is executed symbolically

\[(\text{tc } (\text{All } [a] \ [a \rightarrow a]))\]

\[(\text{fn } [x])\]
\[(\text{GR } (\text{fn } [_] (\text{fn } [_] x))))\]

\[\Rightarrow (\text{All } [a] \ [a \rightarrow a])\]
Prototype Implementation

Symbolic closures let us treat GR as a black box until it is executed symbolically

\[
\text{tc (All [a] [a -> a])}
\]

\[
\text{(fn [x] (GR (fn [_] (fn [_] x)))))}
\]

\[
=> (All [a] [a -> a])
\]
Prototype Implementation

Symbolic closures let us treat GR as a **black box** until it is executed symbolically

\[
(tc \ (All \ \text{[a]} \ \text{[a -> a]}))
\]

\[
(fn \ [x]
   \ (GR \ (fn \ [\_] \ (fn \ [\_] \ x)))))
\]

\[
=\ (All \ [a] \ [a -> a])
\]
Prototype Implementation

Symbolic closures let us treat GR as a black box until it is executed symbolically

\[(tc (All [a] [a -> a]))\]
\[(fn [x] (GR (fn [_] (fn [_] x))))\]
\[=> (All [a] [a -> a])\]
Prototype Implementation

Symbolic closures let us treat GR as a black box until it is executed symbolically

\[(\text{tc (All } [a] [a \rightarrow a])\]
\[(\text{fn } [x] \ (\text{GR (fn } [\_] (\text{fn } [\_] x))))\)

\[=\ (\text{All } [a] [a \rightarrow a])\]
Prototype Implementation

Symbolic closures let us treat GR as a **black box** until it is executed symbolically

\[
\text{GR} \downarrow
\]

\[
\text{(tc (All [a] [a -> a]))}
\]

\[
\text{(fn [x] (GR (fn [_] (fn [_] x)))))}
\]

\[
=> \text{(All [a] [a -> a])}
\]
Prototype Implementation

Symbolic closures let us treat GR as a **black box** until it is executed symbolically

\[
(tc (All [a] [a -> a]))
\]

\[
(fn [x]
  (GR (fn [_] (fn [_] x))))
\]

\[
=> (All [a] [a -> a])
\]

**Symbolic Closures make the most of top-level annotations**
Scorecard

- Functional programming: ✓ ✗
- Immutability: ✓
- The REPL: { ✗
- Ease of development: { ✗
- Host Interop: ✓

“Check more programs!”
Scorecard

Symbolic closure prototype:
Checks more programs

- Functional programming: ✔️
- Immutability: ✔️
- The REPL: ✗
- Ease of development: ✗
- Host Interop: ✔️

“Check more programs!”
Conclusion
Typed Clojure is a **sound** and **practical** optional type system for Clojure
Typed Clojure is a **sound** and **practical** optional type system for Clojure.

I present the **design** of Typed Clojure, **formalize** the core type system, and prove it **sound**.
Typed Clojure is a **sound** and **practical** optional type system for Clojure.

I **empirically** show Typed Clojure’s features correspond to **real-world** programs.
Typed Clojure is a **sound** and **practical** optional type system for Clojure.

I present a tool to **automatically generate annotations** and use it to port **real-world** Clojure programs.
Typed Clojure is a sound and practical optional type system for Clojure.

I identify and prototype several extensions to improve errors and type check more programs.
Thanks
Extra slides
Type soundness Proof

1. Extend calculus with Java-style throwable errors
2. Make explicit assumptions about Java
3. Add “stuck”, “wrong”, and “error” rules to semantics
4. Shown: Well-typed programs reduce to correct values or errors
   • By induction on the reduction derivation, then cases on final red.
     rule and final (non-subsump.) typing rule
5. Corollary: Well-typed programs don’t “go wrong”
6. Corollary: Well-typed programs don’t throw null-ptr exceptions