Typed Clojure in Theory and Practice

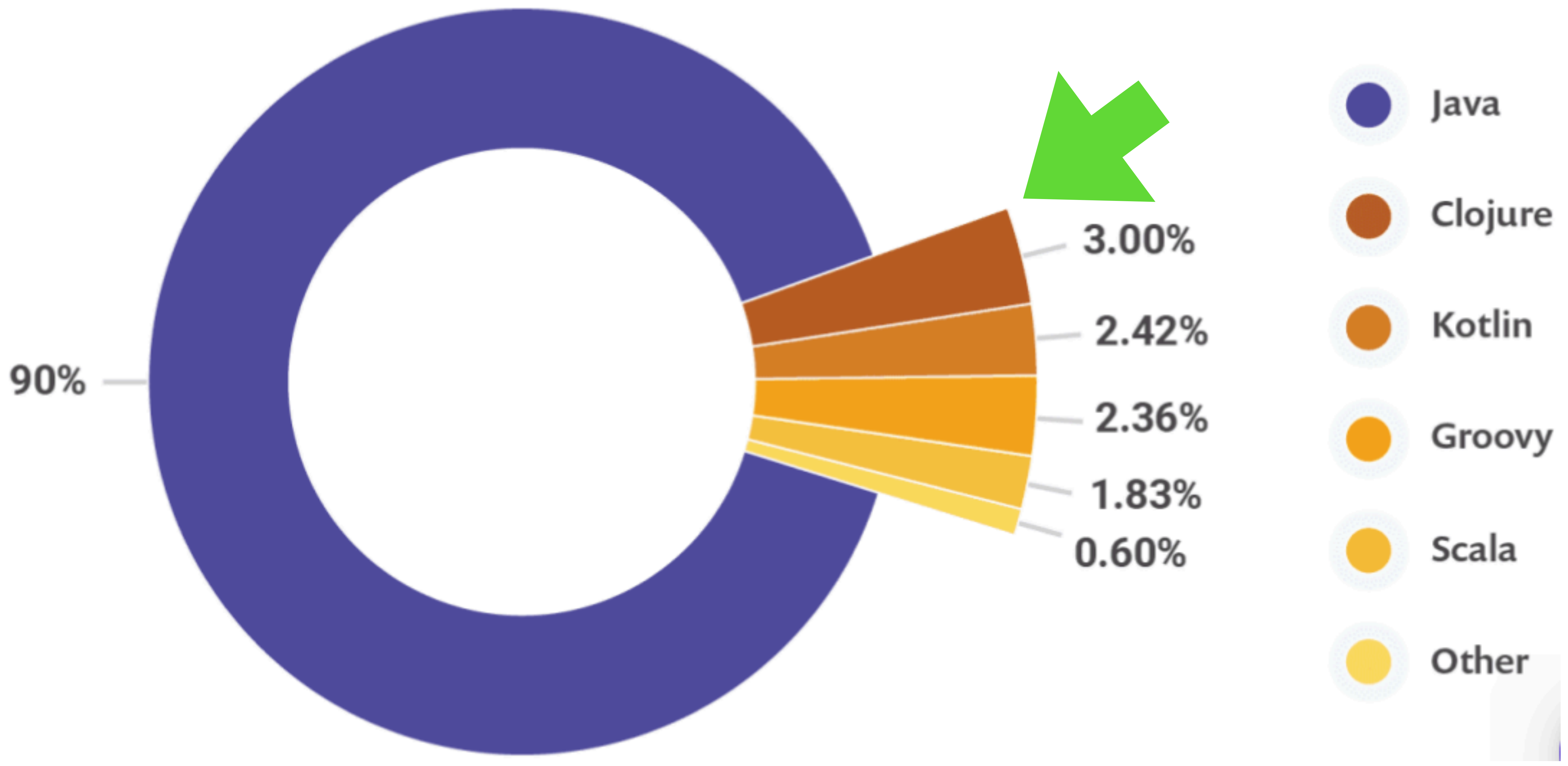
Ambrose Bonnaire-Sergeant

What is Clojure?

A programming language
running on the Java Virtual Machine

What is Clojure?

A programming language
running on the Java Virtual Machine

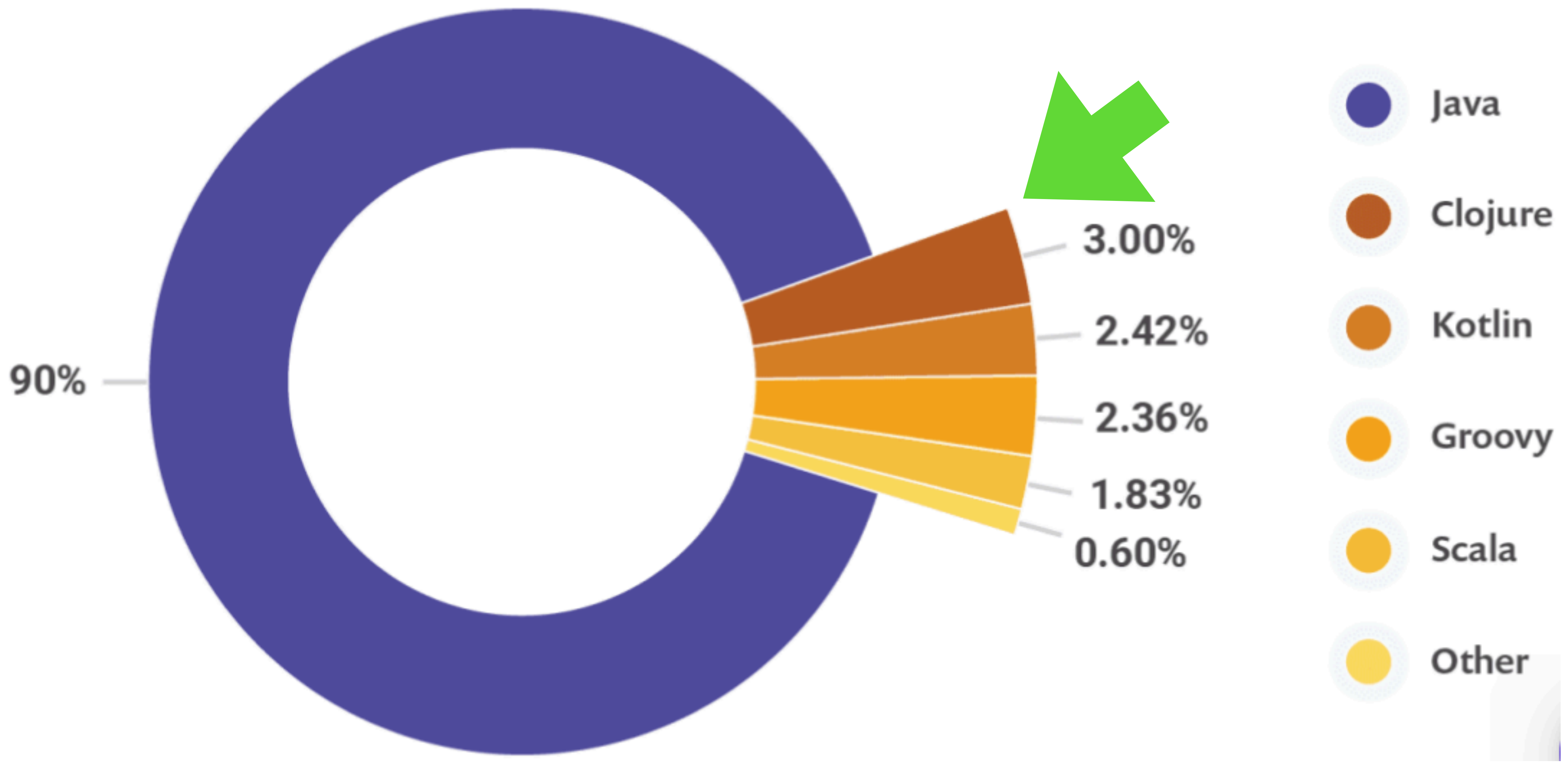


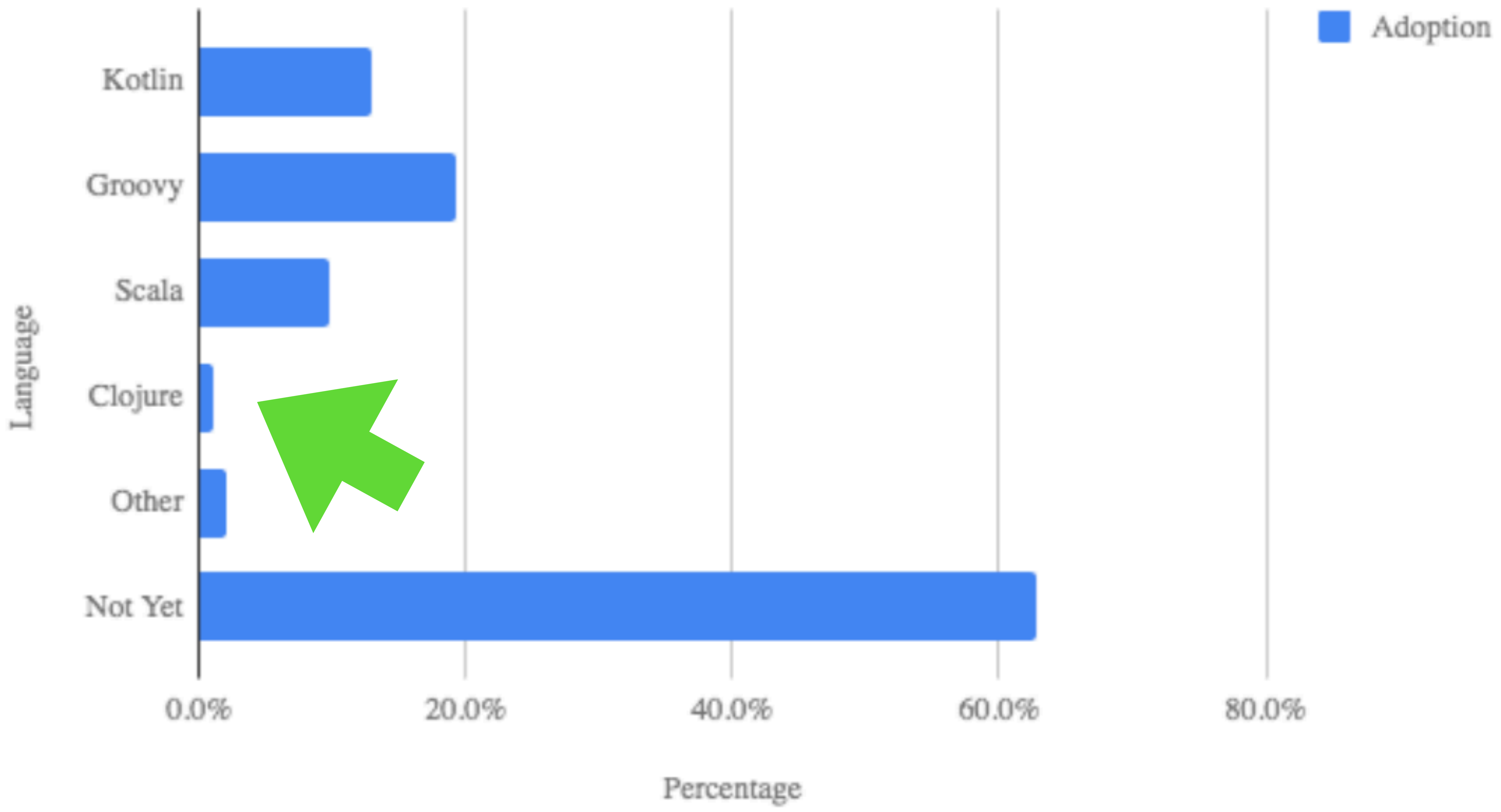
3% of JVM users' primary language is Clojure

- [JVM Ecosystem Report 2018, snyk.io]

What is Clojure?

A programming language
running on the Java Virtual Machine





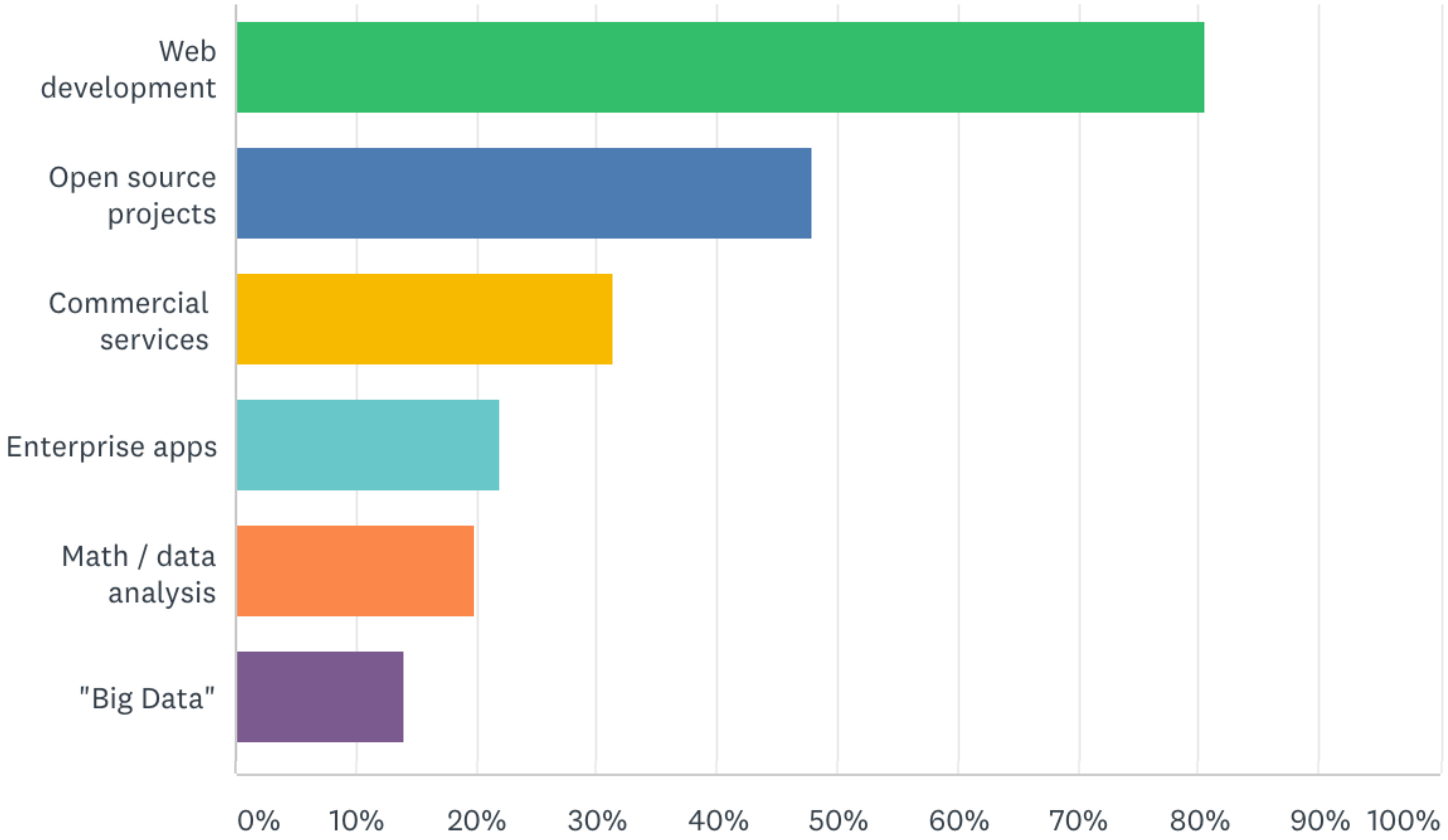
3% of JVM users' primary language is Clojure

- [JVM Ecosystem Report 2018, snyk.io]

1.1% of JVM users have adopted Clojure

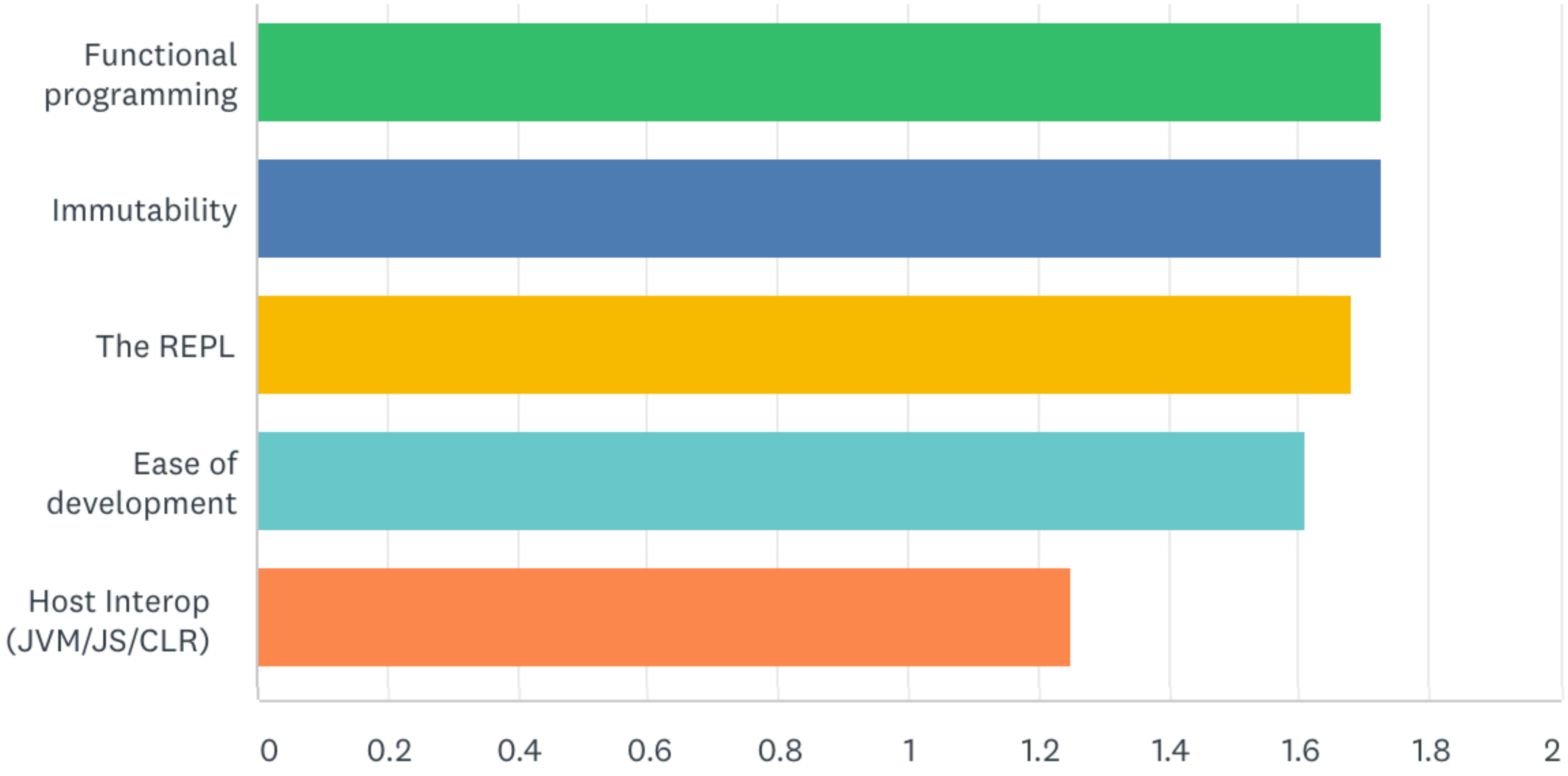
- [The State of Java in 2018, baeldung.com]

General Purpose



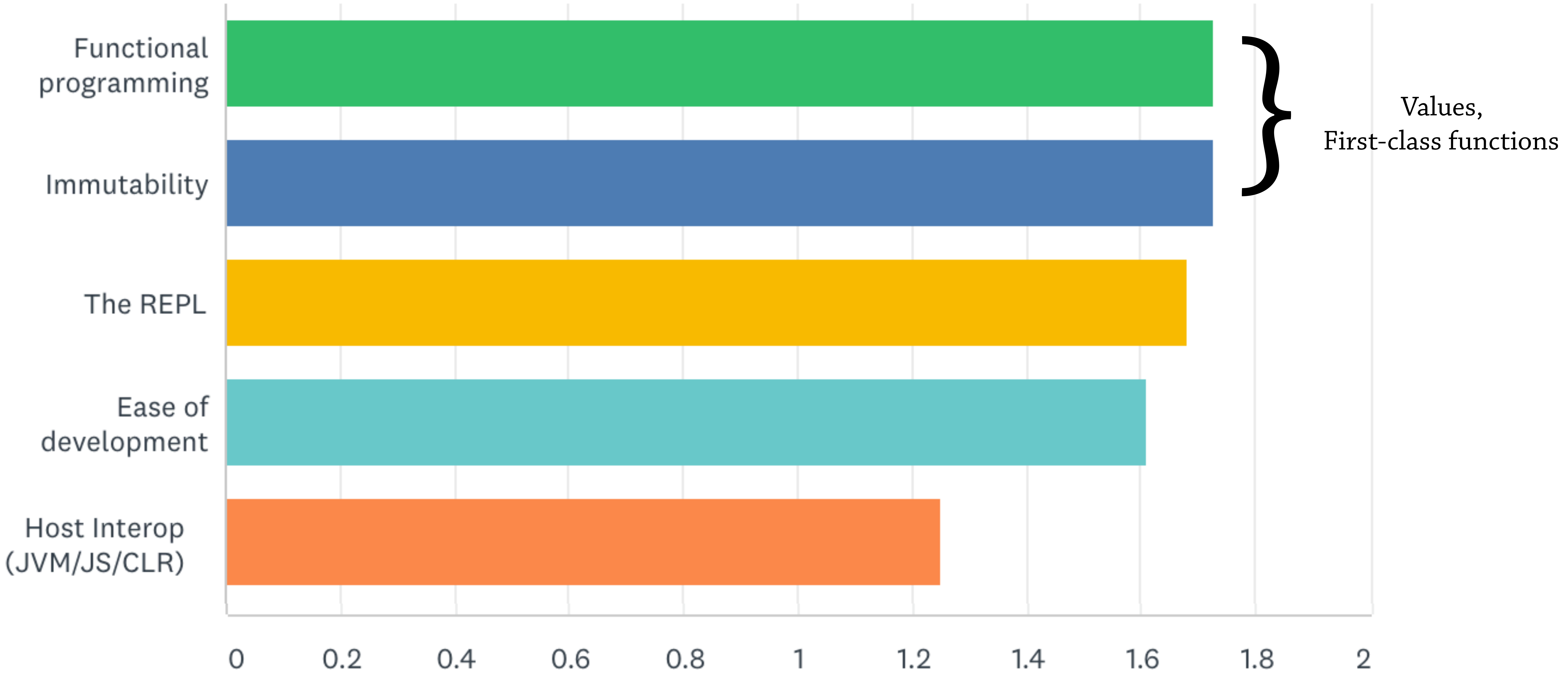
[State of Clojure 2019 Survey]

Survey: Why Clojure?



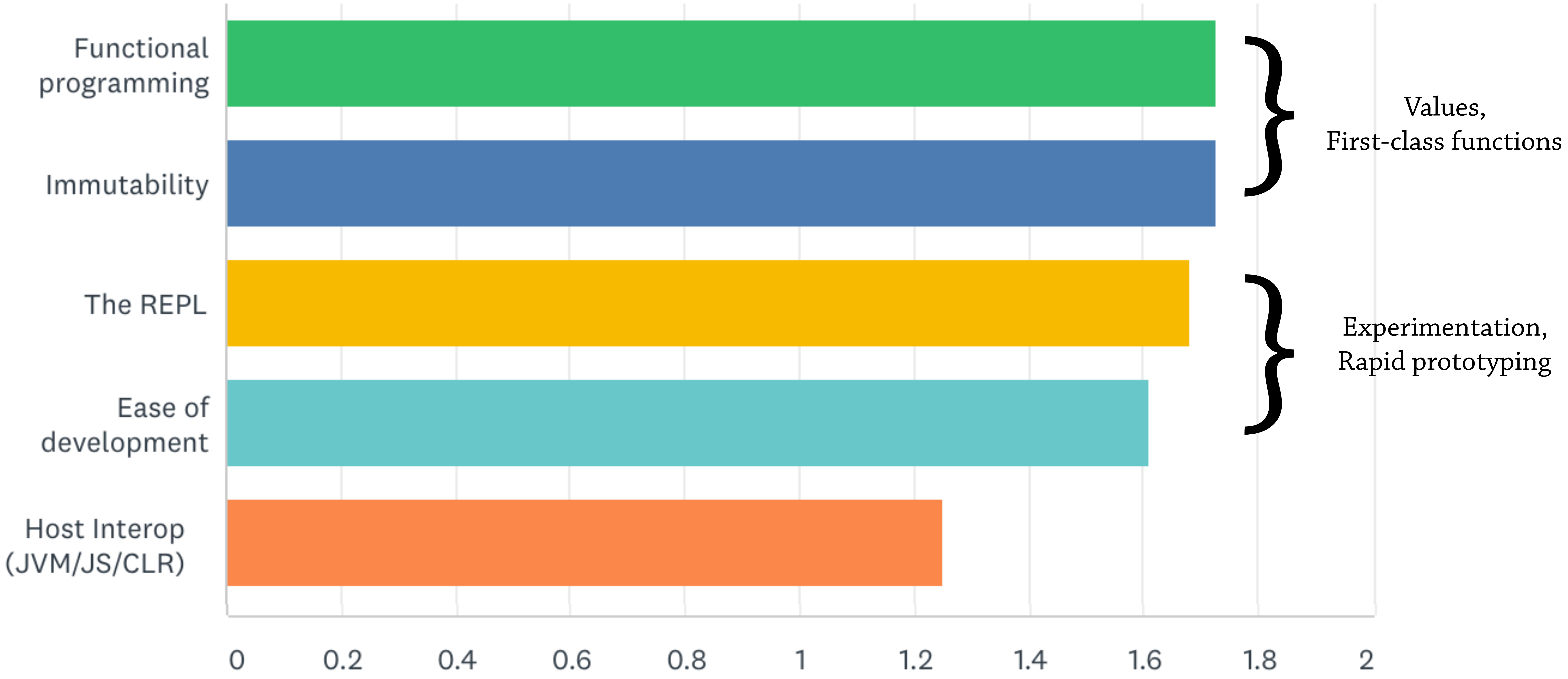
[State of Clojure 2019 Survey, Weighted average: 0 = Not Important, 1 = Important, 2 = Very Important]

Survey: Why Clojure?



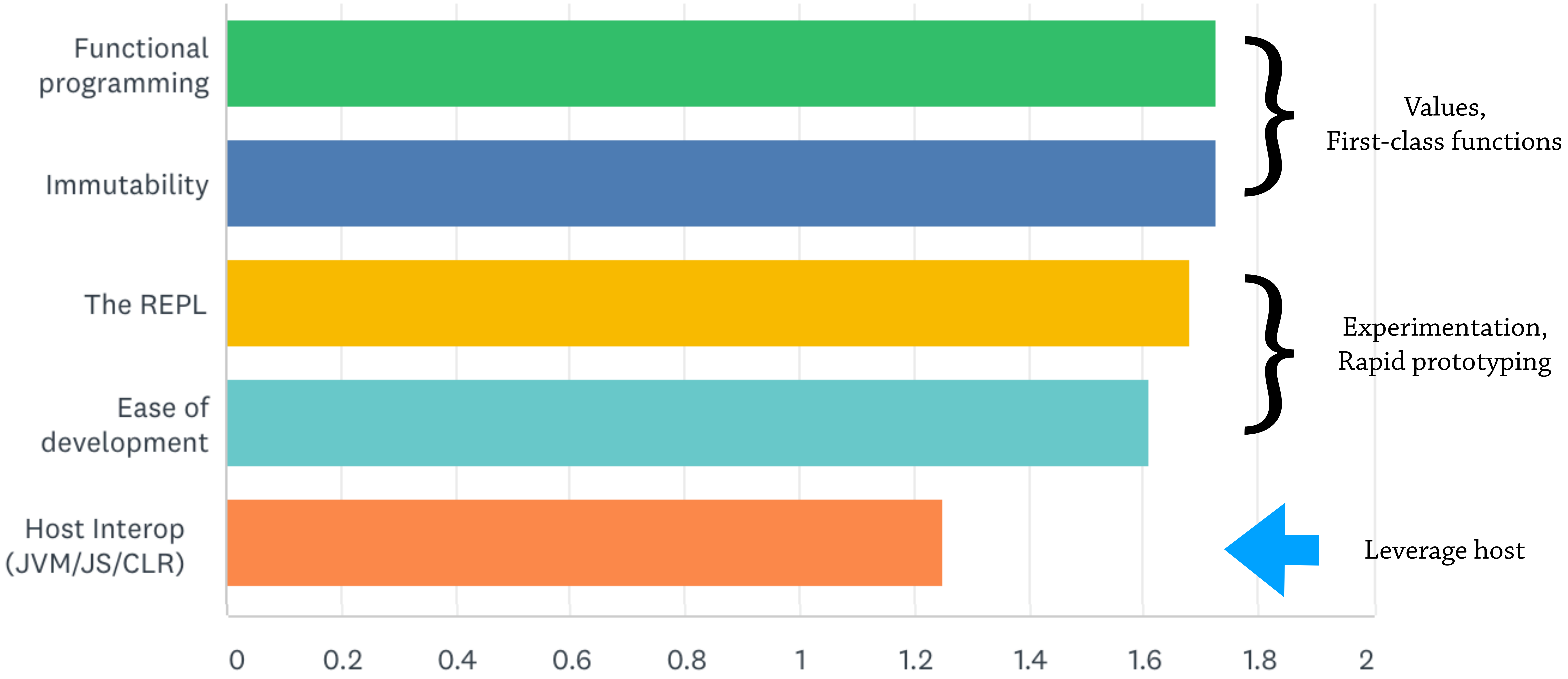
[State of Clojure 2019 Survey, Weighted average: 0 = Not Important, 1 = Important, 2 = Very Important]

Survey: Why Clojure?



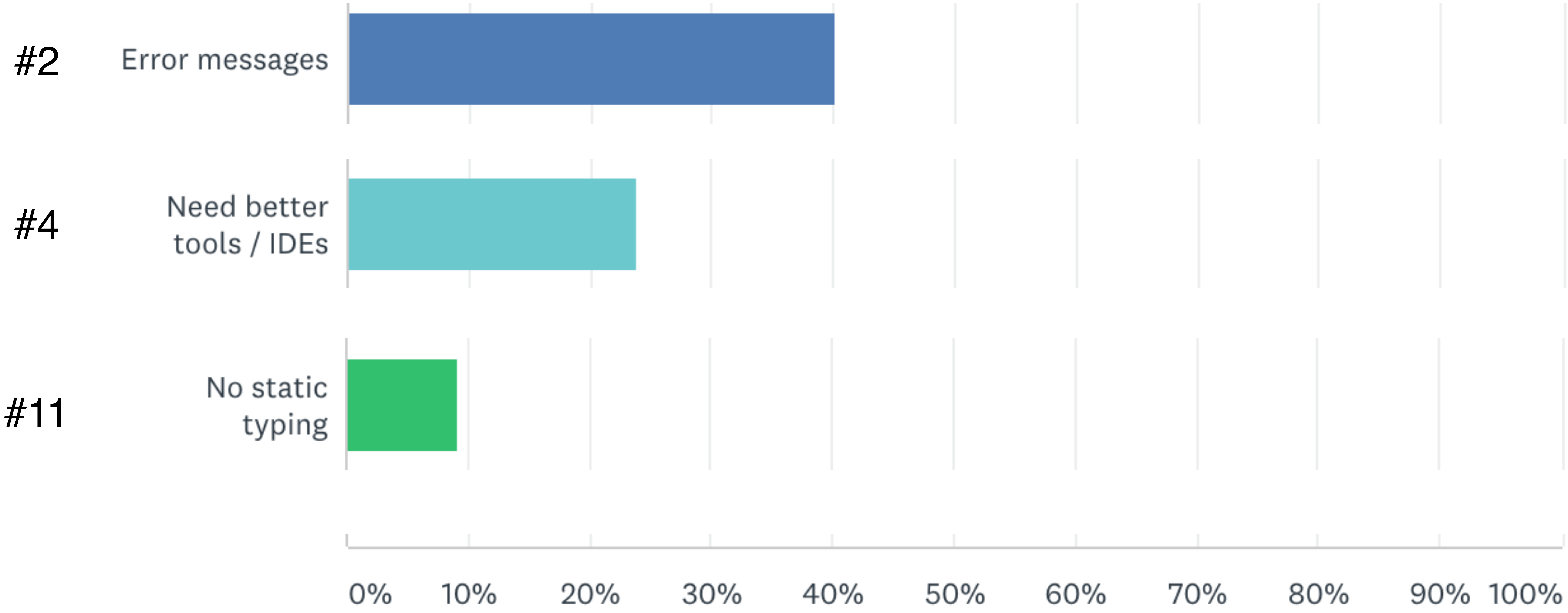
[State of Clojure 2019 Survey, Weighted average: 0 = Not Important, 1 = Important, 2 = Very Important]

Survey: Why Clojure?



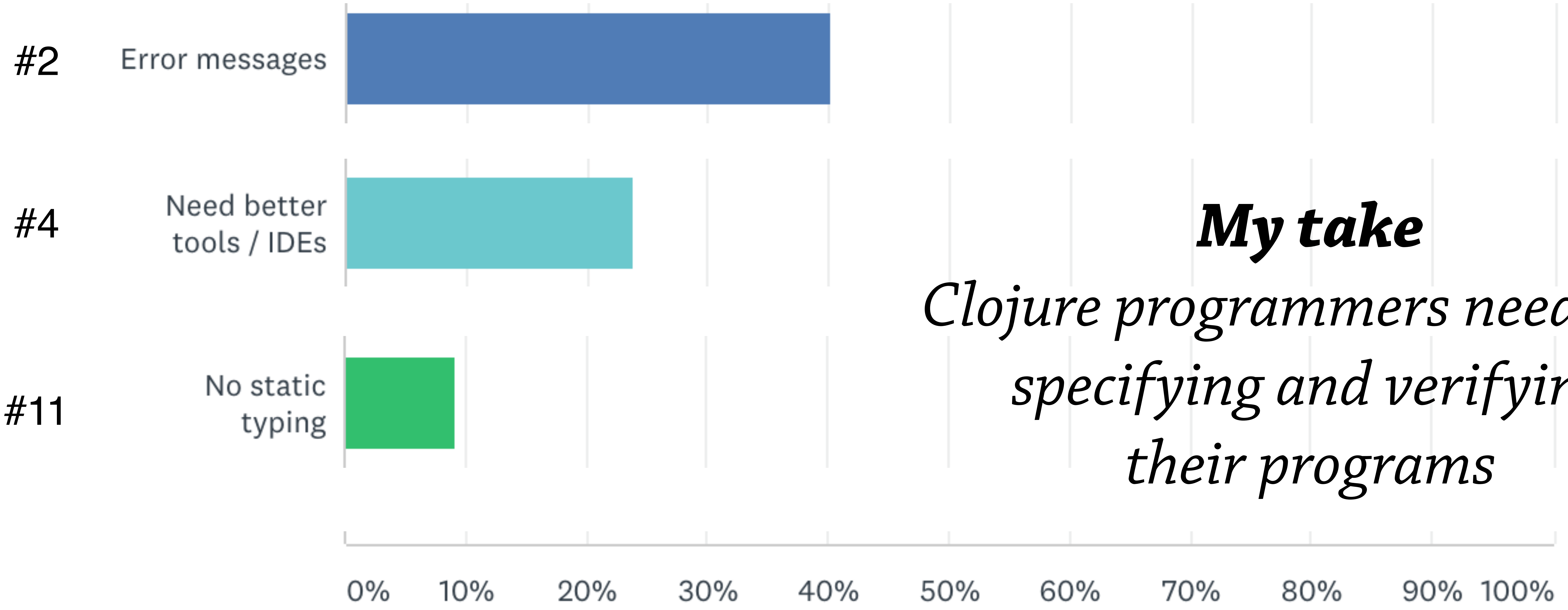
[State of Clojure 2019 Survey, Weighted average: 0 = Not Important, 1 = Important, 2 = Very Important]

Frustrations with Clojure



[State of Clojure 2019 Survey]

Frustrations with Clojure



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 2013 2014 2015 2016 2017

2012
Clojure conj
Google Summer of Code

2013
INDIEGOGO
$35,254 USD
728 backers
Google Summer of Code

2014
strangeloop
Google Summer of Code
Google Summer of Code
Google Summer of Code
EMERGING TECHNOLOGIES FOR THE ENTERPRISE CONFERENCE
CODE MESH

2015
strangeloop
Google Summer of Code
Google Summer of Code
INDIEGOGO
$11,695 USD by 199 backers

2016
INDIEGOGO
$8,621 USD
73 backers
ETAPS
EUROPEAN JOINT CONFERENCES ON THEORY & PRACTICE OF SOFTWARE

2017
Clojure conj

clojure / core.typed | Unwatch 92 | Unstar 1,076 | Fork 68

Typed Clojure @TypedClojure | Followers 1,574

How Typed Clojure works

How Typed Clojure works

1. Take an existing Clojure program

```
(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)

```
(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:

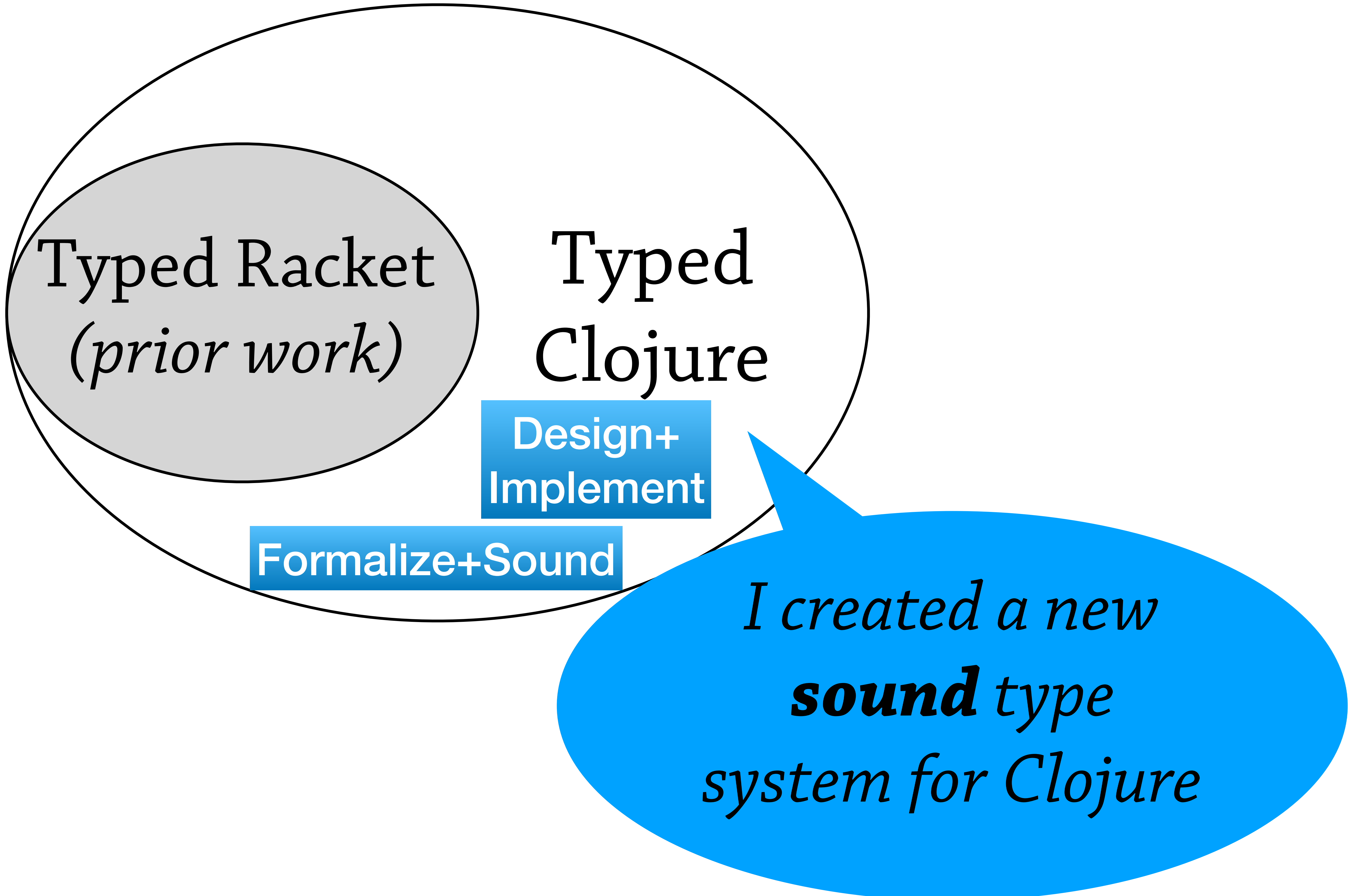
Typed Clojure is a
sound and **practical**
optional type system for Clojure

Typed Racket
(prior work)

My starting point for Typed Clojure

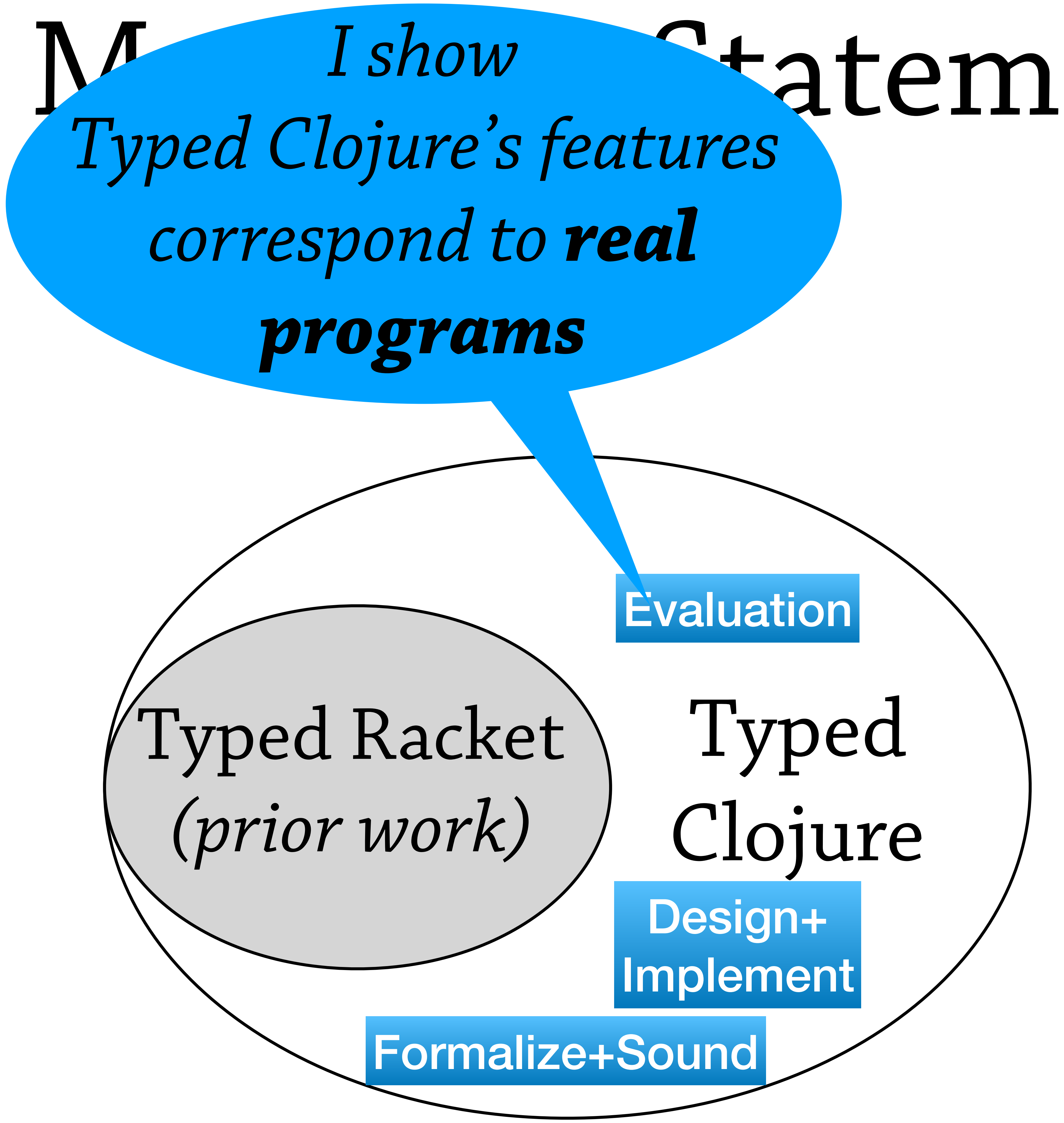
My Thesis Statement:

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



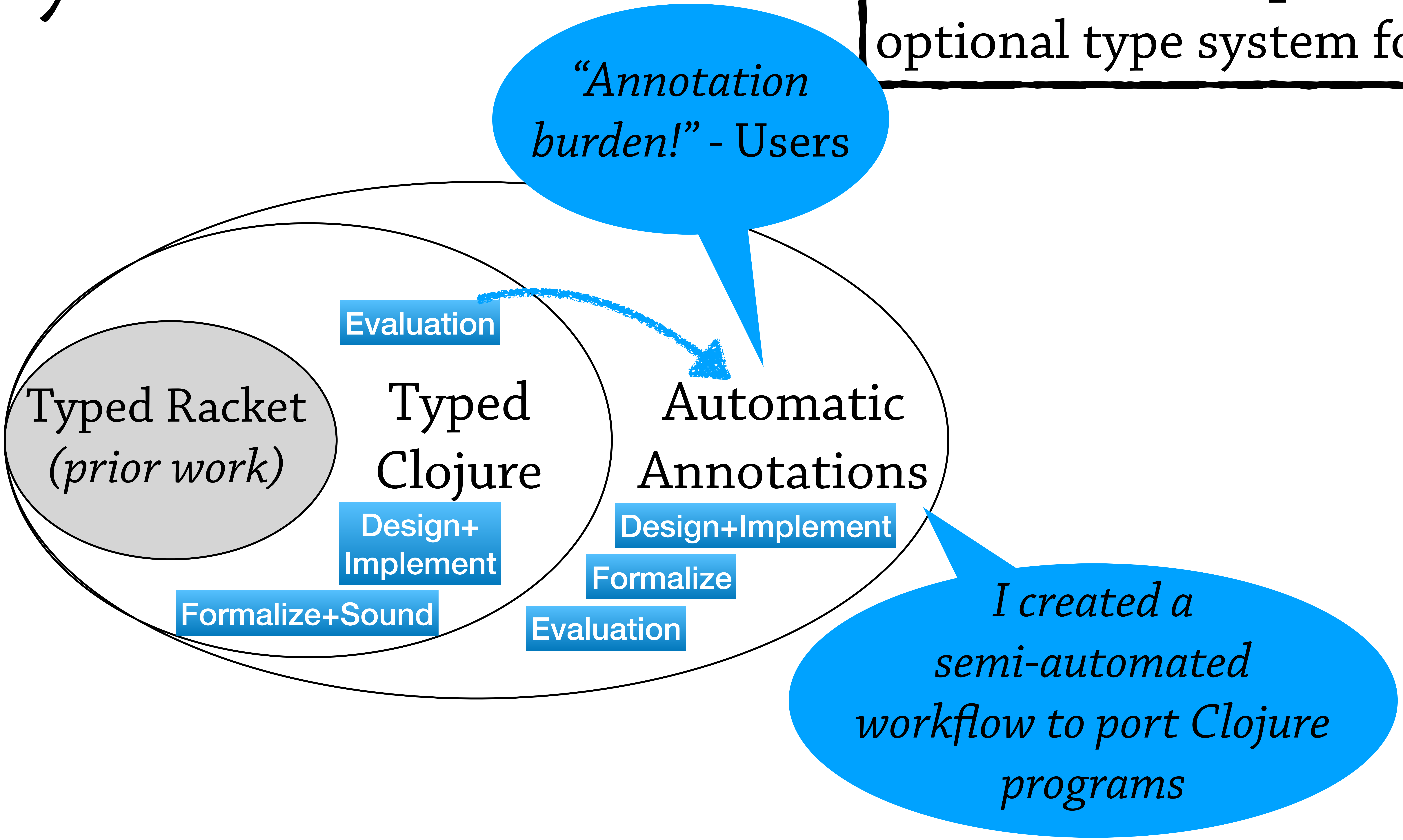
M Statement:

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



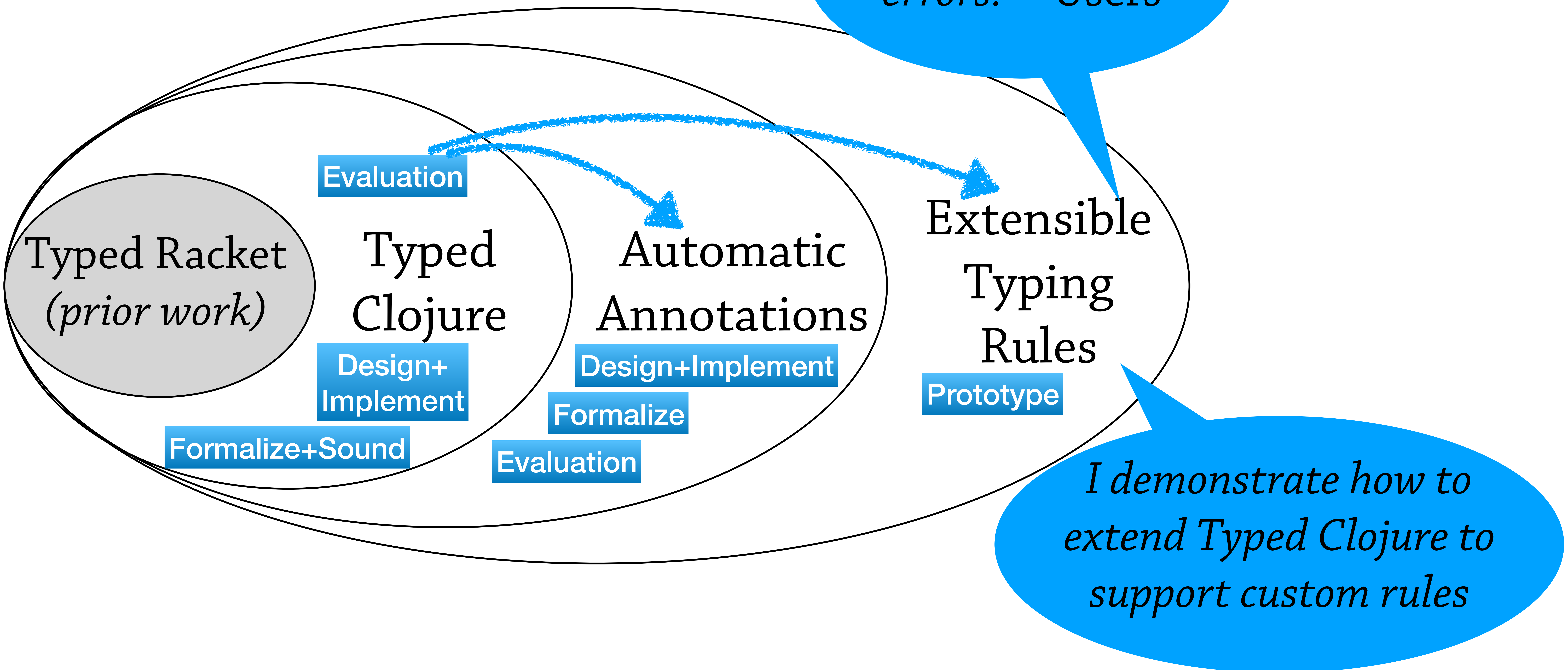
My Thesis Statement:

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



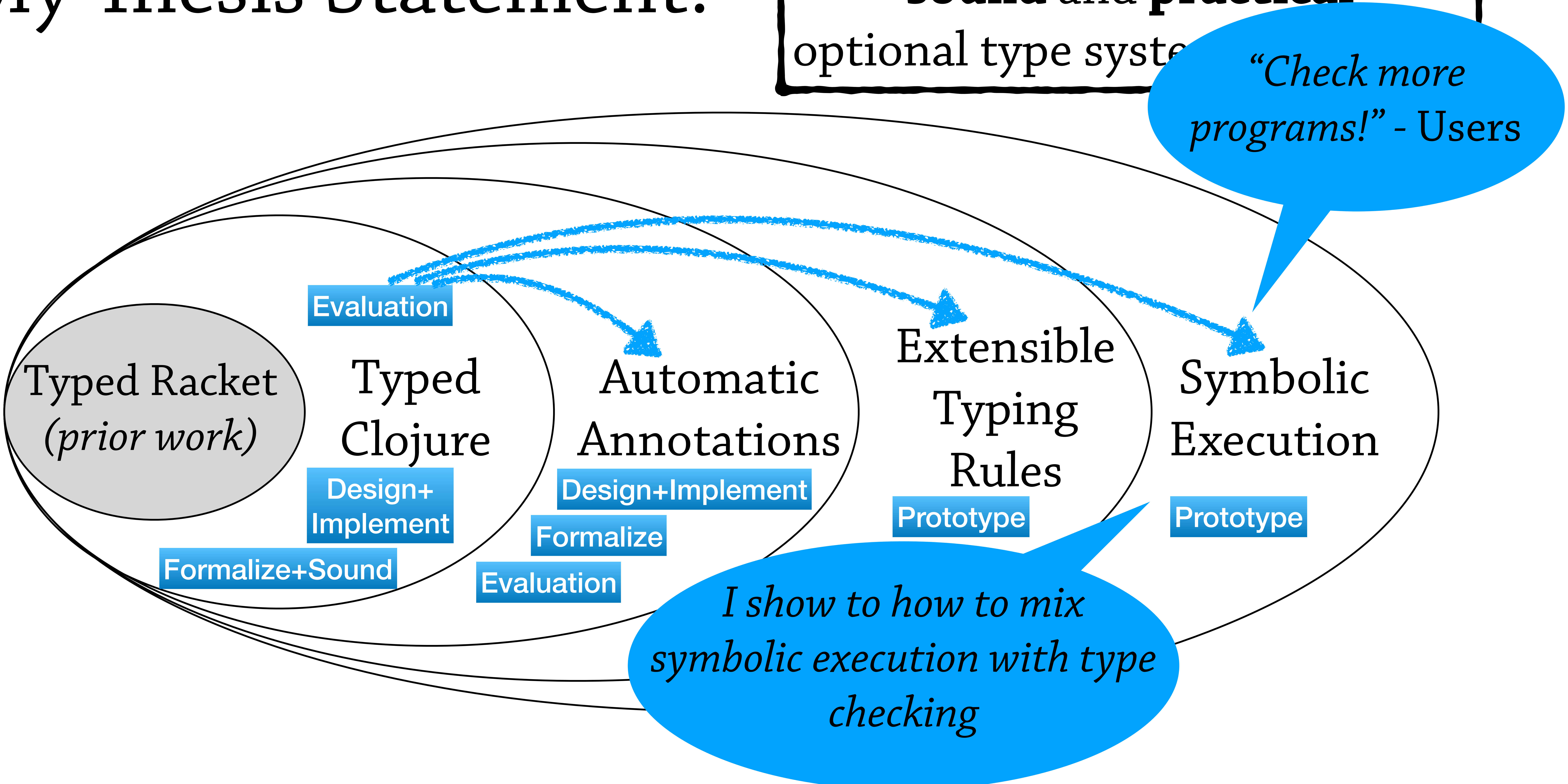
My Thesis Statement:

Typed Clojure is a
sound and **practical**
o… …tem for Clojure

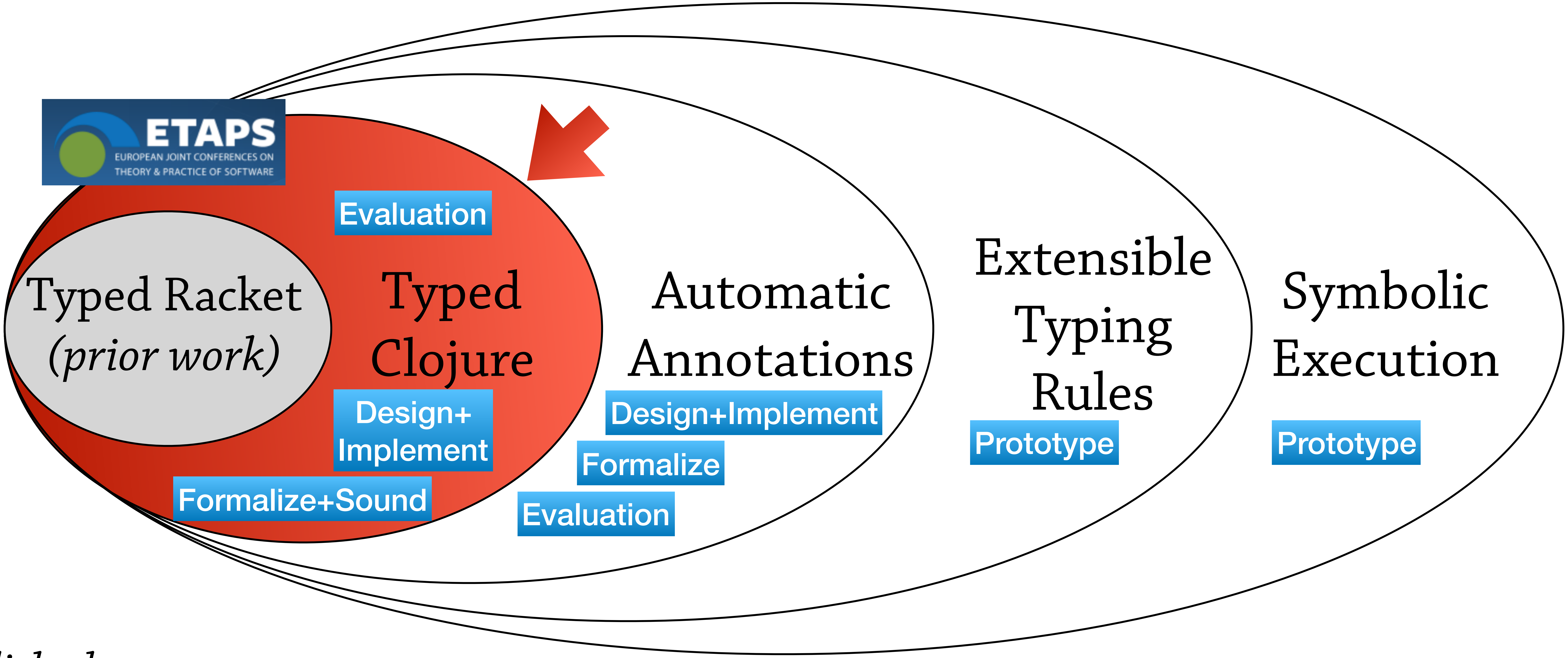


My Thesis Statement:

Typed Clojure is a **sound** and **practical** optional type syste

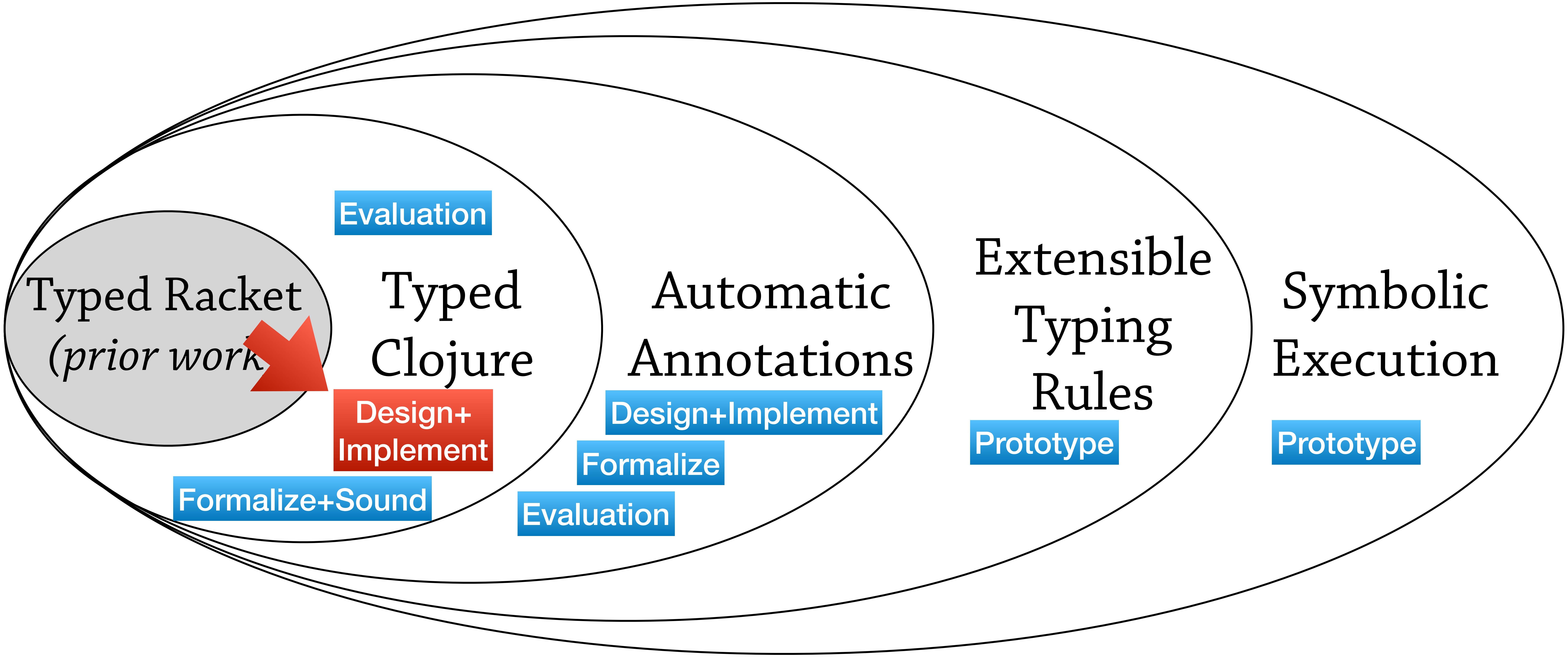


Part I
Design and Evaluation of Typed Clojure

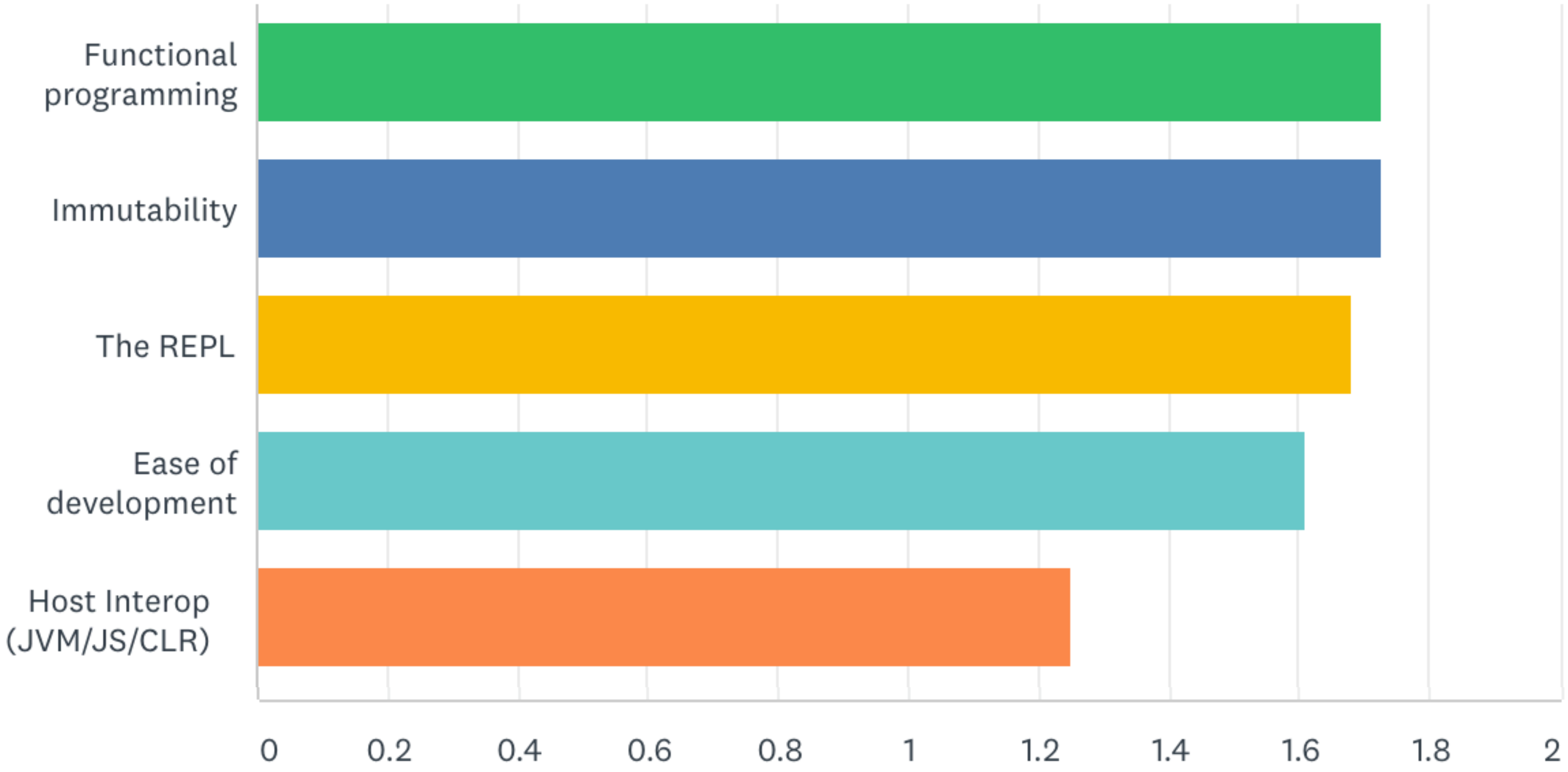


Published:

"Practical Optional Types for Clojure", **Ambrose Bonnaire-Sergeant**, Rowan Davies, Sam Tobin-Hochstadt; **ESOP 2016**


Evaluation
Typed Racket
(prior work)
Typed
Clojure
Design+
Implement
Formalize+Sound
Automatic
Annotations
Design+Implement
Formalize
Evaluation
Extensible
Typing
Rules
Prototype
Symbolic
Execution
Prototype

Check with Typed Clojure



Simple Functions

Scorecard

Functional programming

Immutability

The REPL

Ease of development

Host Interop

```
(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
```

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

```
(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

```
(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
```

Str

Int

Type-Based Control flow

Scorecard

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

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

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

(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

```clojure
(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

```
(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))
(defmethod to-int-mm Number [m] m)

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

Str

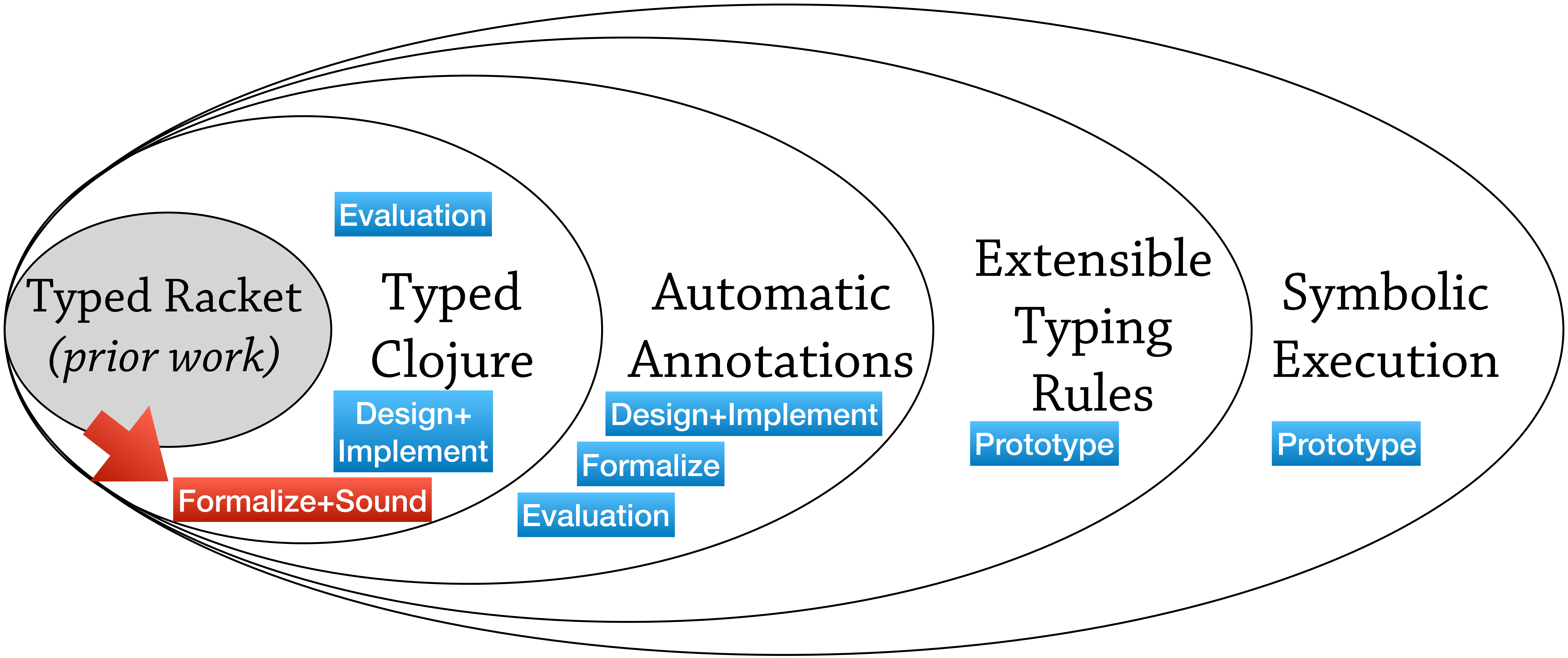
Int

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          Int
(to-int-mm "2") ;=> 2
```



Evaluation
Typed Racket
(prior work)
Typed
Clojure
Design+
Implement
Formalize+Sound
Automatic
Annotations
Design+Implement
Formalize
Evaluation
Extensible
Typing
Rules
Prototype
Symbolic
Execution
Prototype

λ_{TC}

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…

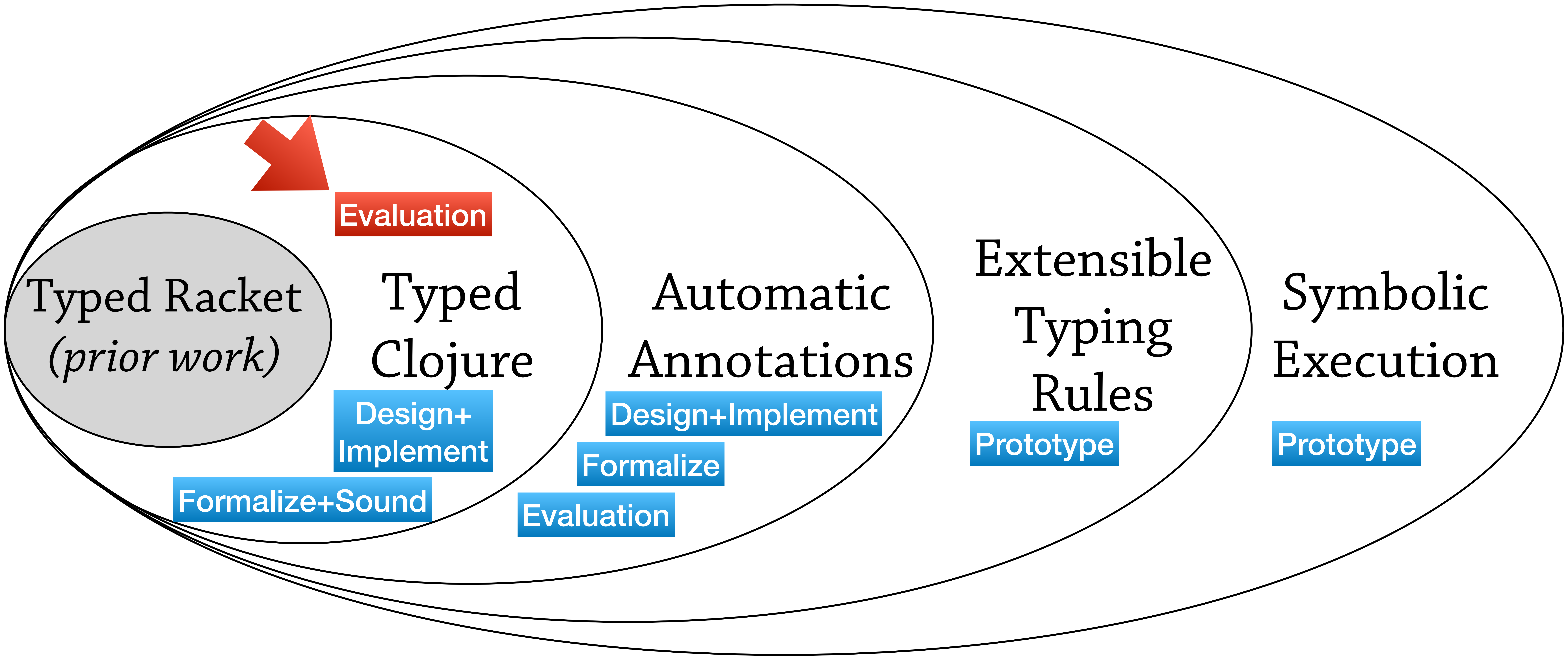
[1] ICFP '10 - Tobin-Hochstadt, Felleisen

λ_{TC}

Type soundness

Theorem Well-typed programs don't "go wrong"

Corollary Well-typed programs **don't throw null-pointer exceptions**


Evaluation
Typed Racket
(prior work)
Typed
Clojure
Design+
Implement
Formalize+Sound
Automatic
Annotations
Design+Implement
Formalize
Evaluation
Extensible
Typing
Rules
Prototype
Symbolic
Execution
Prototype

Empirical Evaluation of Typed Clojure

circleci

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

Required!

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

Required!

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

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

```
(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! ✗

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

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
```

✗ *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
```

✗ *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))
```

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	✓ ✗
Immutability	✓
The REPL Ease of development	} ✗
Host Interop	✓

Scorecard: Typed Clojure's initial design

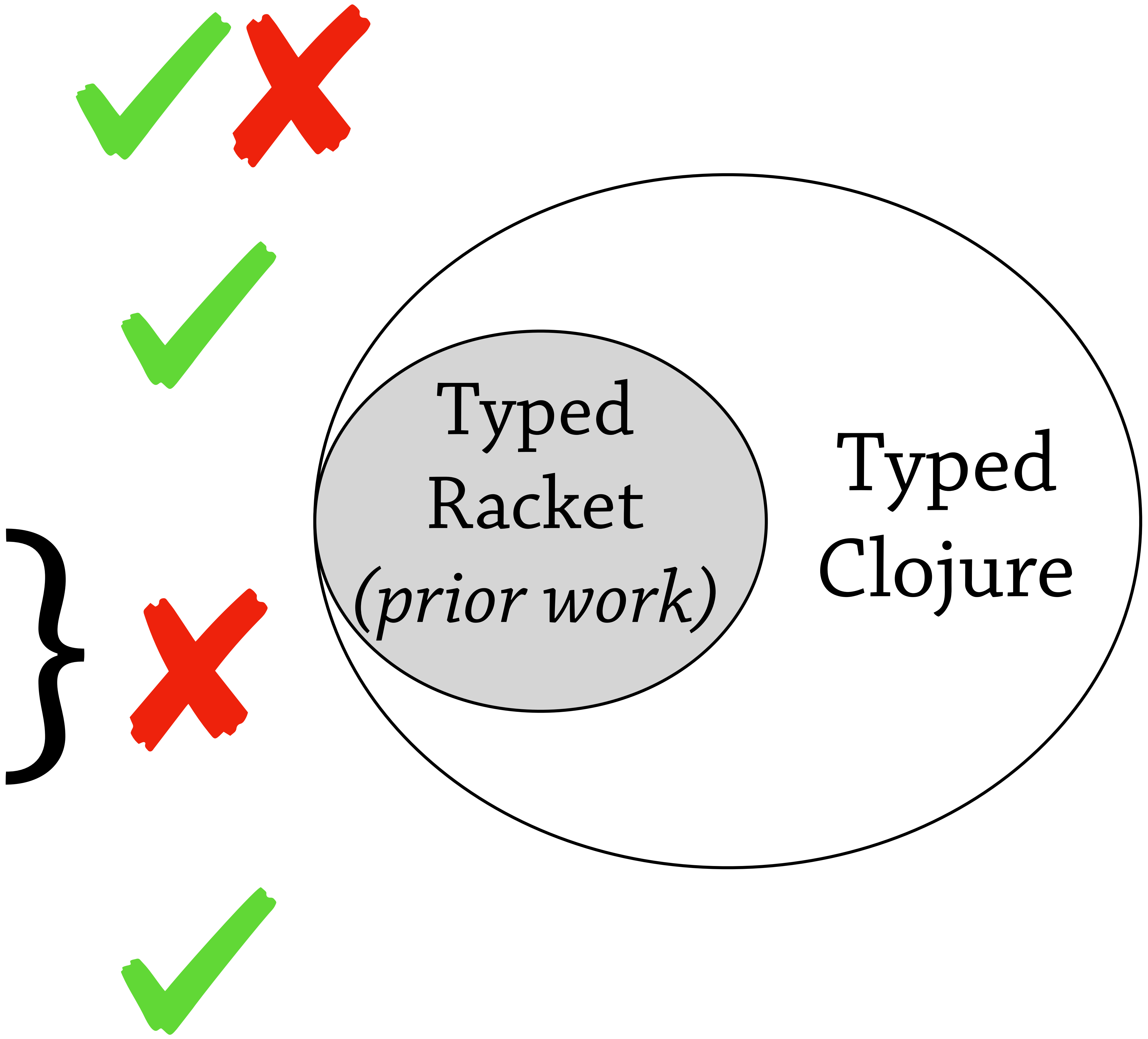
Functional programming

Immutability

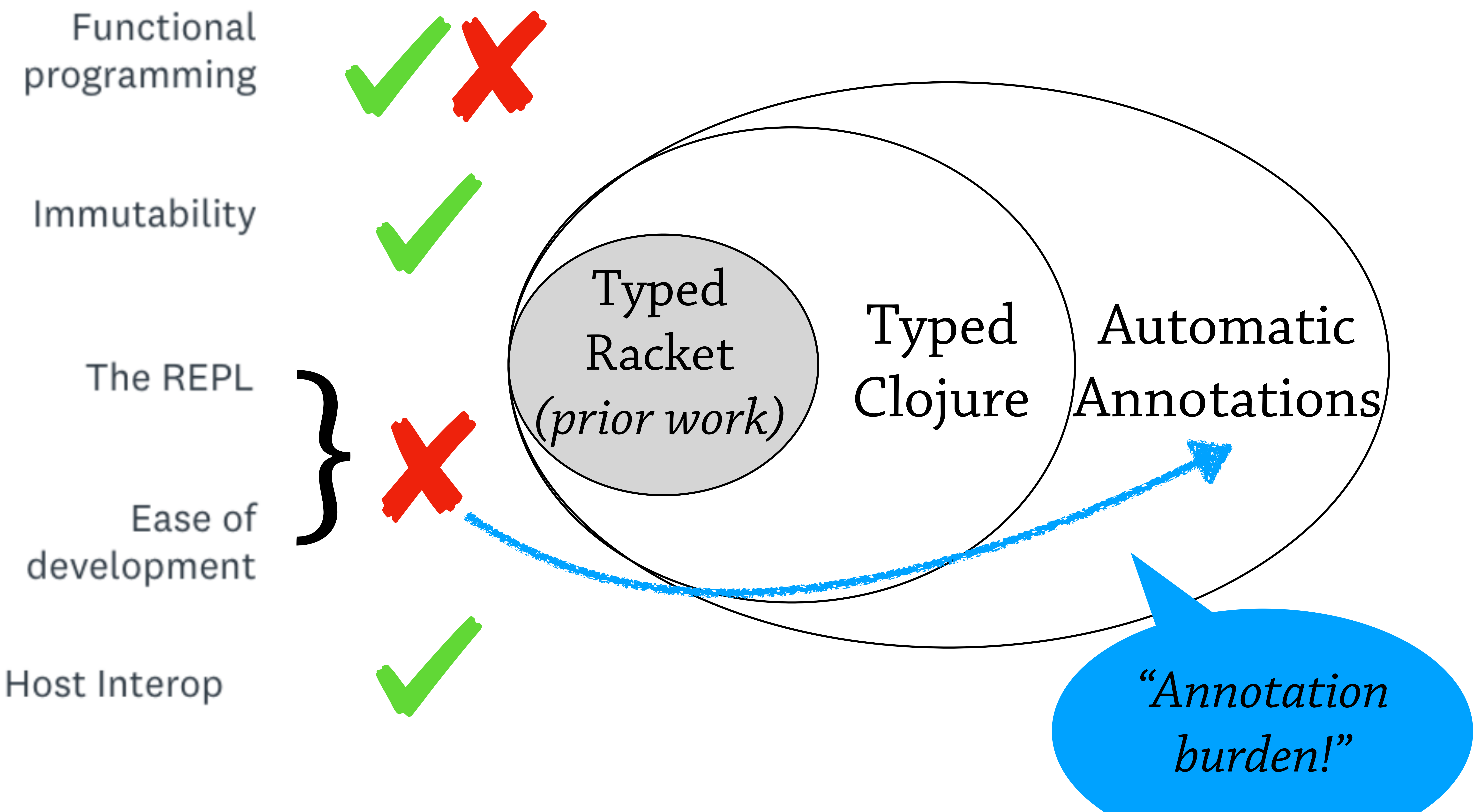
The REPL

Ease of development

Host Interop



Scorecard: Typed Clojure's initial design



Scorecard: Typed Clojure's initial design

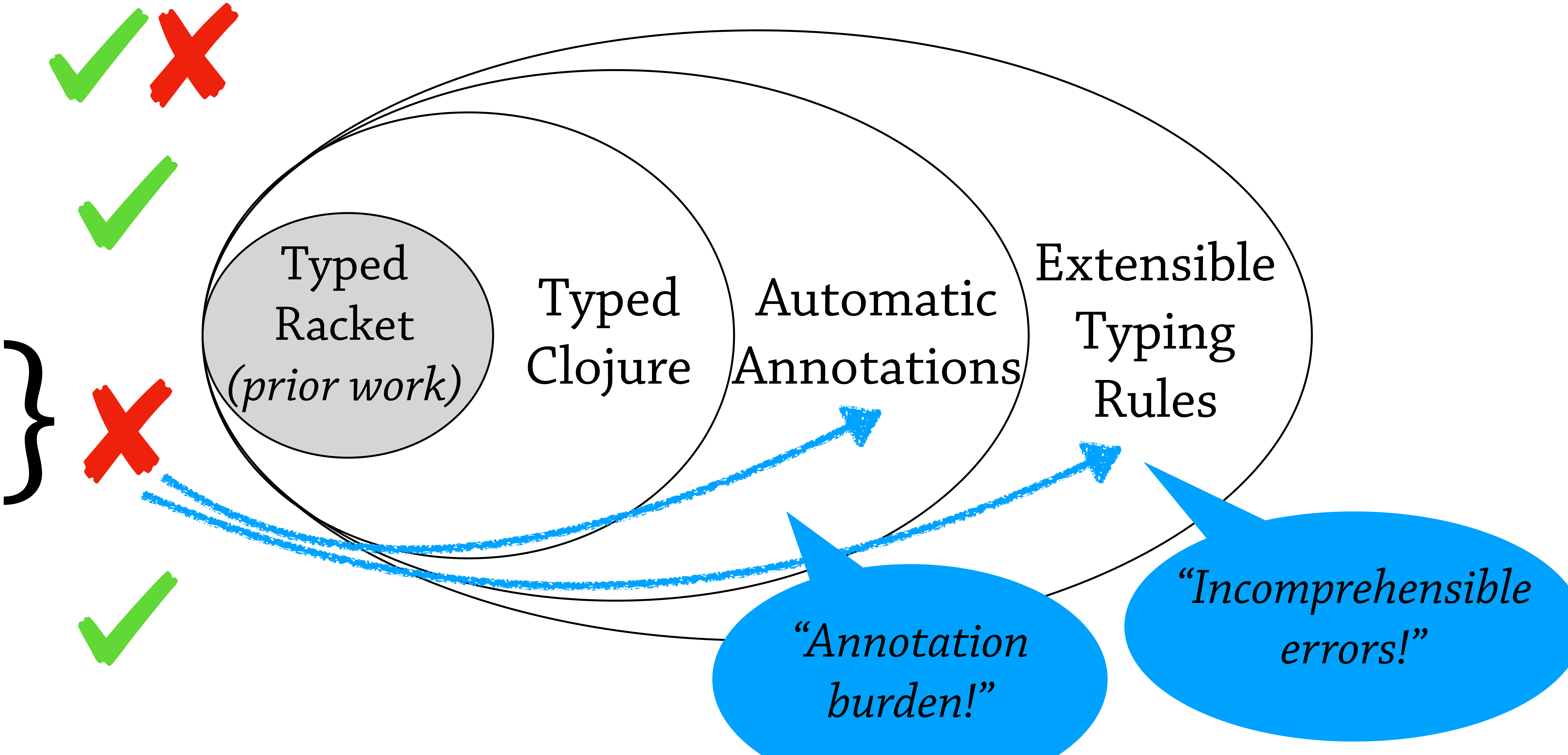
Functional programming

Immutability

The REPL

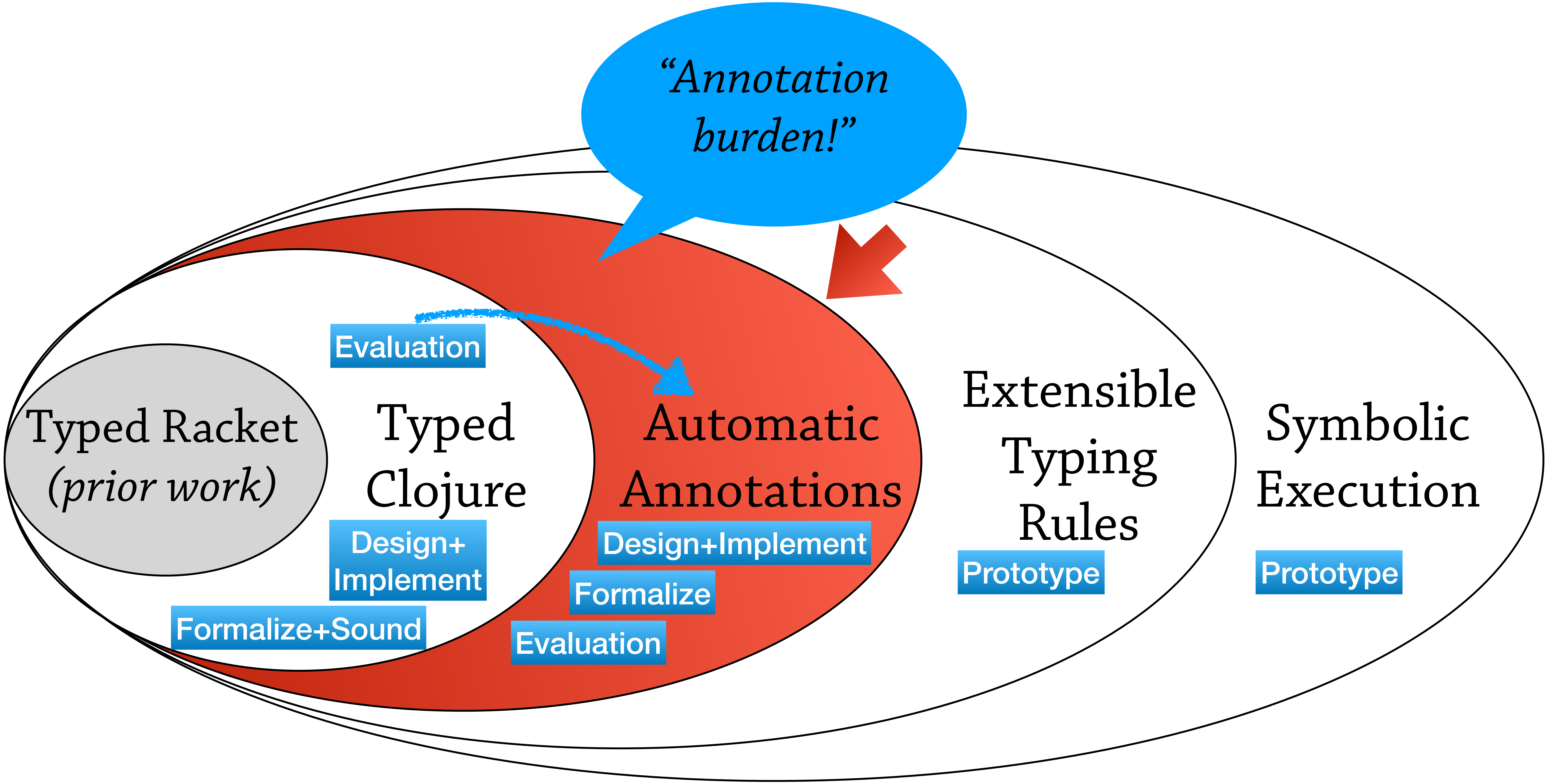
Ease of development

Host Interop



Scorecard: Typed Clojure's initial design

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 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])
```

Annotation burden

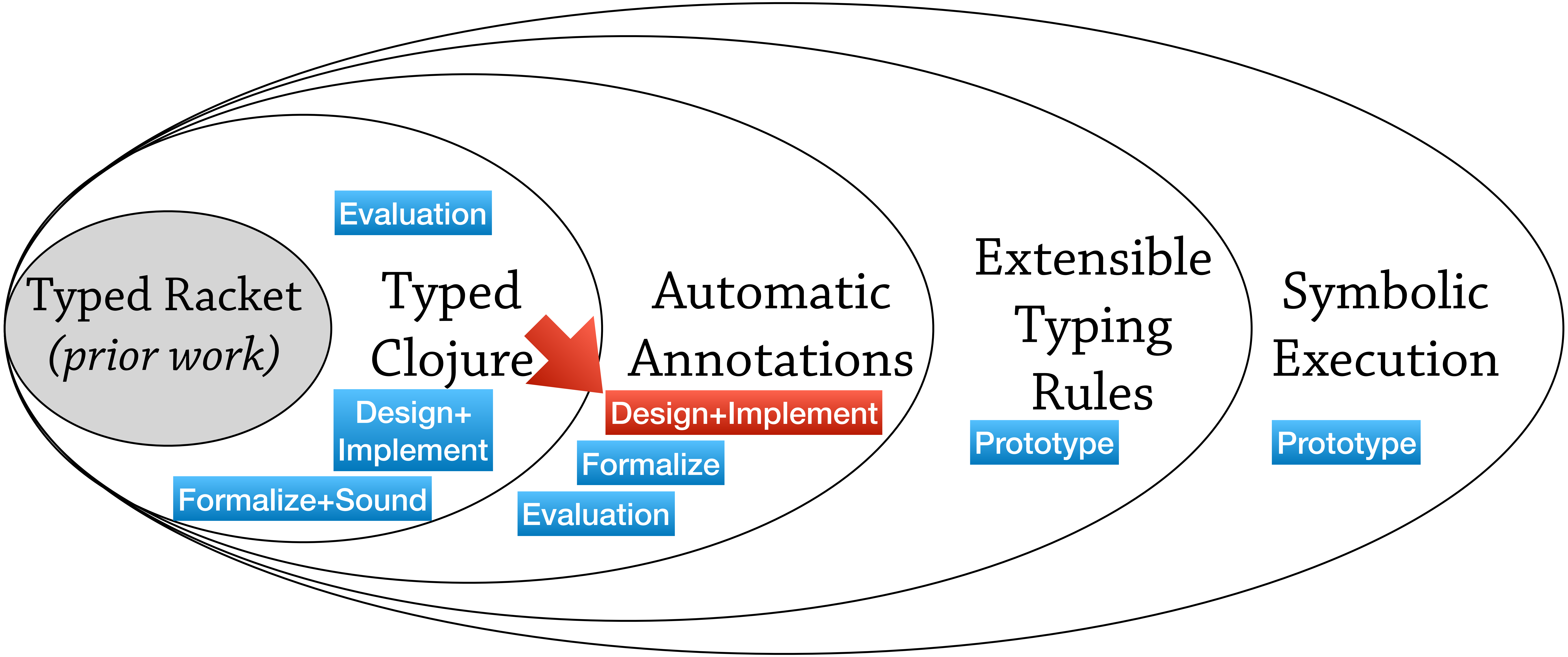
```
(defalias Point
  '{:x Int :y Int})

(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])
```

Goal: Automatically generate


Typed Racket
(prior work)
Evaluation
Typed
Clojure
Design+
Implement
Formalize+Sound
Automatic
Annotations
Design+Implement
Formalize
Evaluation
Extensible
Typing
Rules
Prototype
Symbolic
Execution
Prototype

`(def forty-two 42)`

Tool design

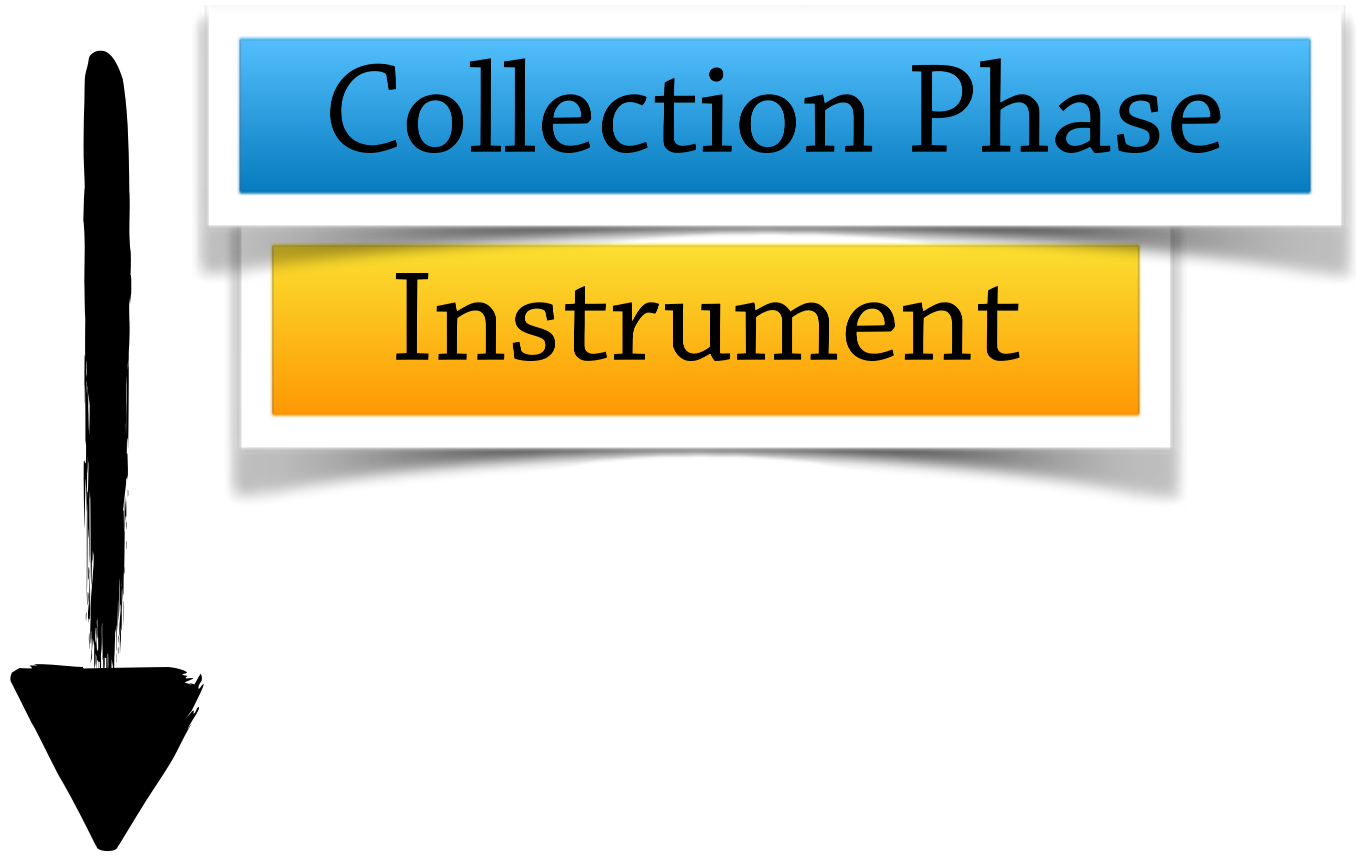
Tool design

```
(def forty-two 42)
```

$$\Gamma = \{\texttt{forty-two} : \texttt{Long}\}$$

Tool design

```
(def forty-two 42)
```




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

$$\Gamma = \{\texttt{forty-two} : \texttt{Long}\}$$

Tool design

```
(def forty-two 42)
```

Collection Phase
Instrument

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

Collection Phase
Track

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

$$\Gamma = \{\texttt{forty-two} : \texttt{Long}\}$$

Tool design

```
(def forty-two 42)
```

Collection Phase

Instrument

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

Collection Phase

Track

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

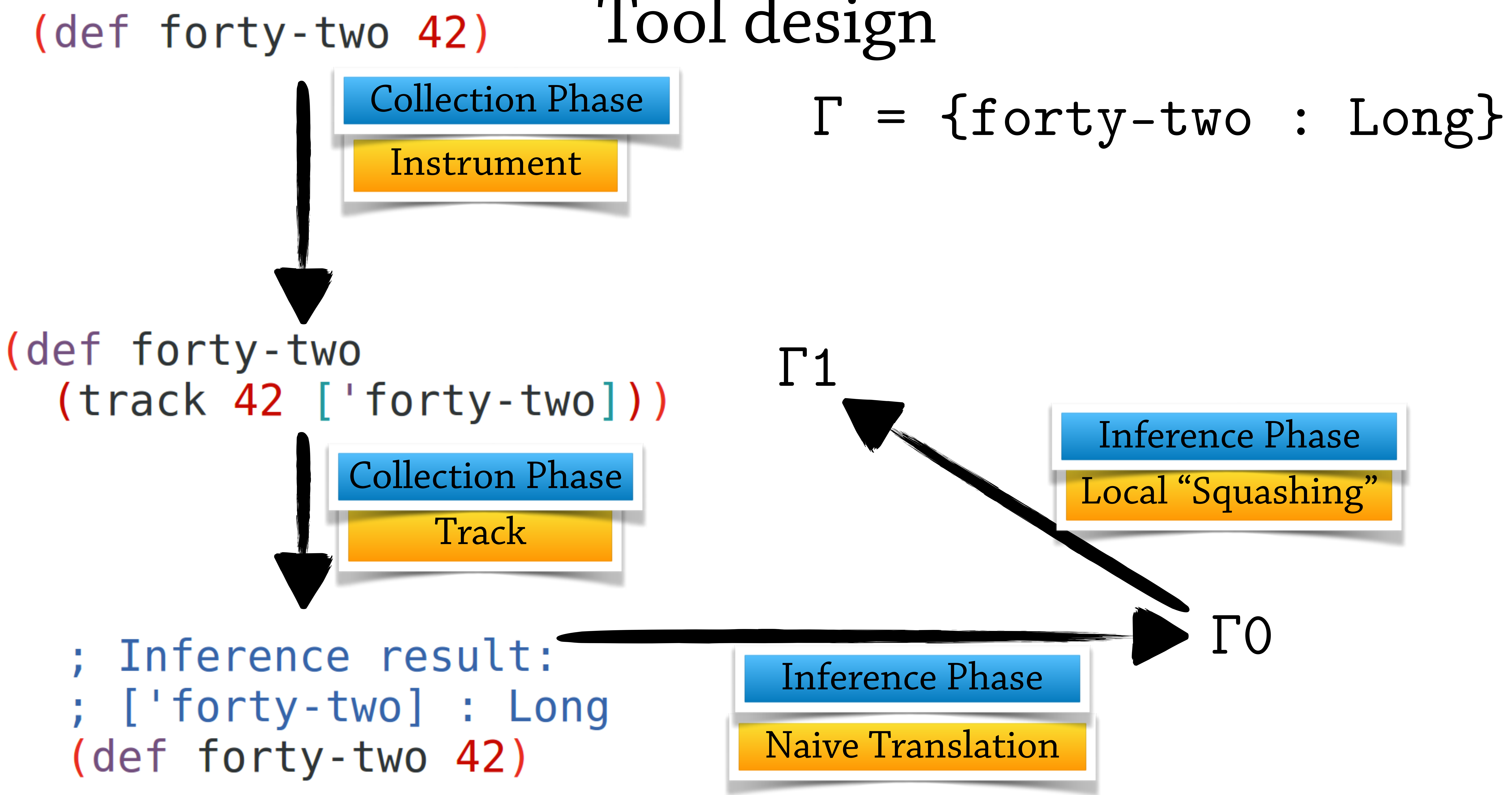
Inference Phase

Naive Translation

$\Gamma 0$

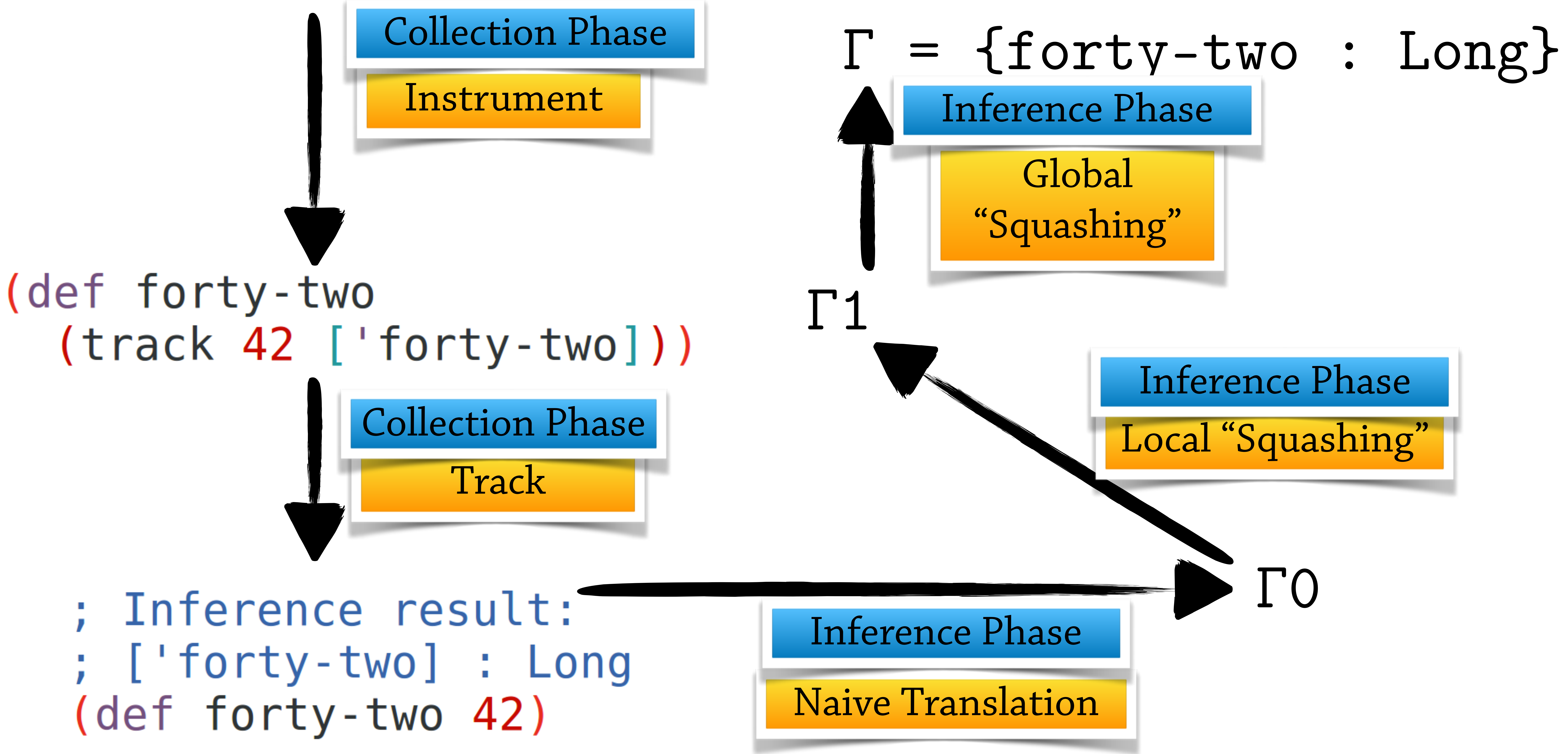
Γ = {forty-two : Long}

Tool design

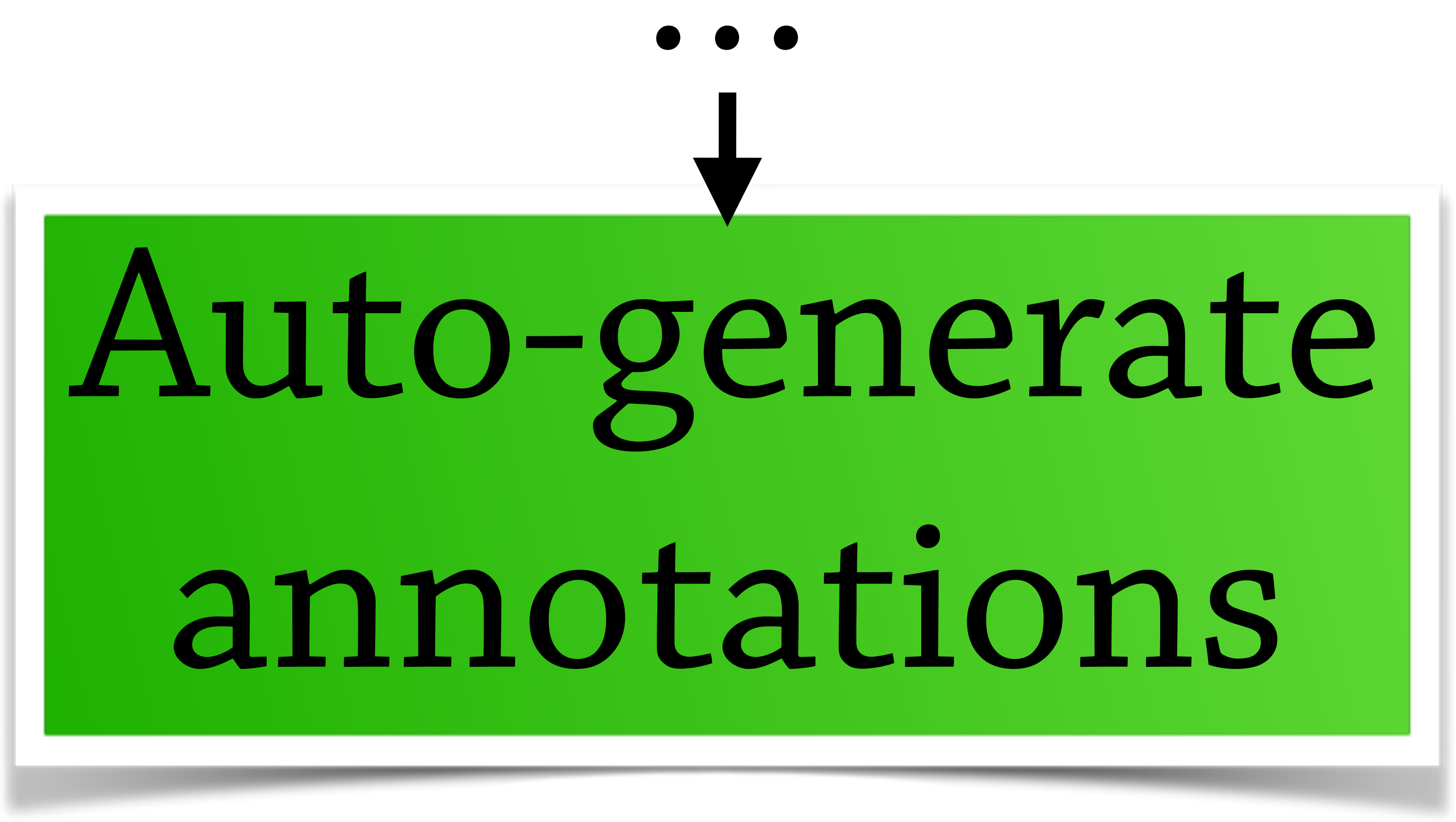


(def forty-two 42)
Collection Phase
Instrument
Γ = {forty-two : Long}
(def forty-two
(track 42 ['forty-two]))
Γ1
Inference Phase
Local "Squashing"
Collection Phase
Track
Γ0
; Inference result:
; ['forty-two] : Long
(def forty-two 42)
Inference Phase
Naive Translation

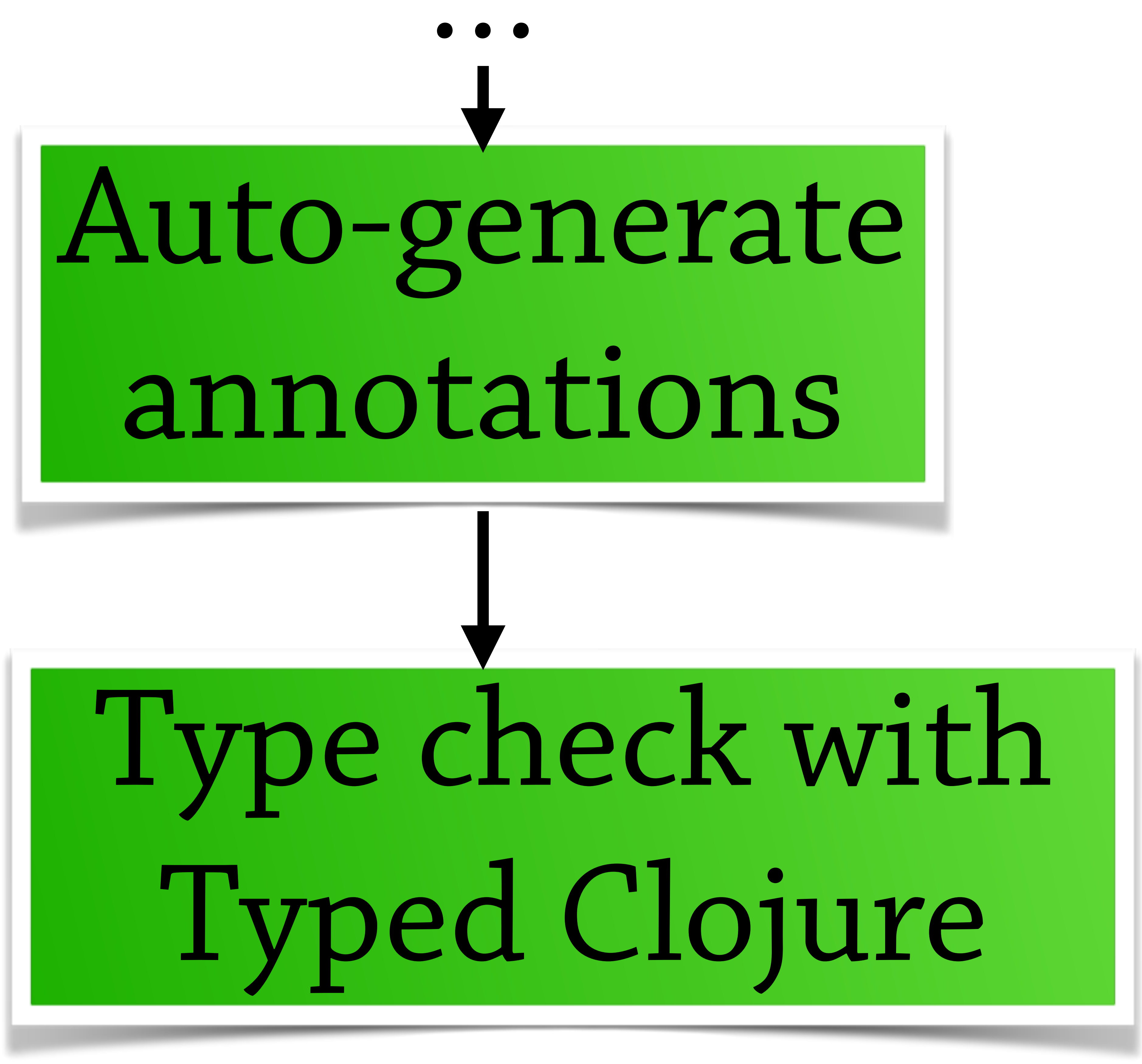
Tool design


(def forty-two 42)
Collection Phase
Instrument
(def forty-two
(track 42 ['forty-two]))
Collection Phase
Track
; Inference result:
; ['forty-two] : Long
(def forty-two 42)
Inference Phase
Naive Translation
Γ0
Inference Phase
Local "Squashing"
Γ1
Inference Phase
Global "Squashing"
Γ = {forty-two : Long}

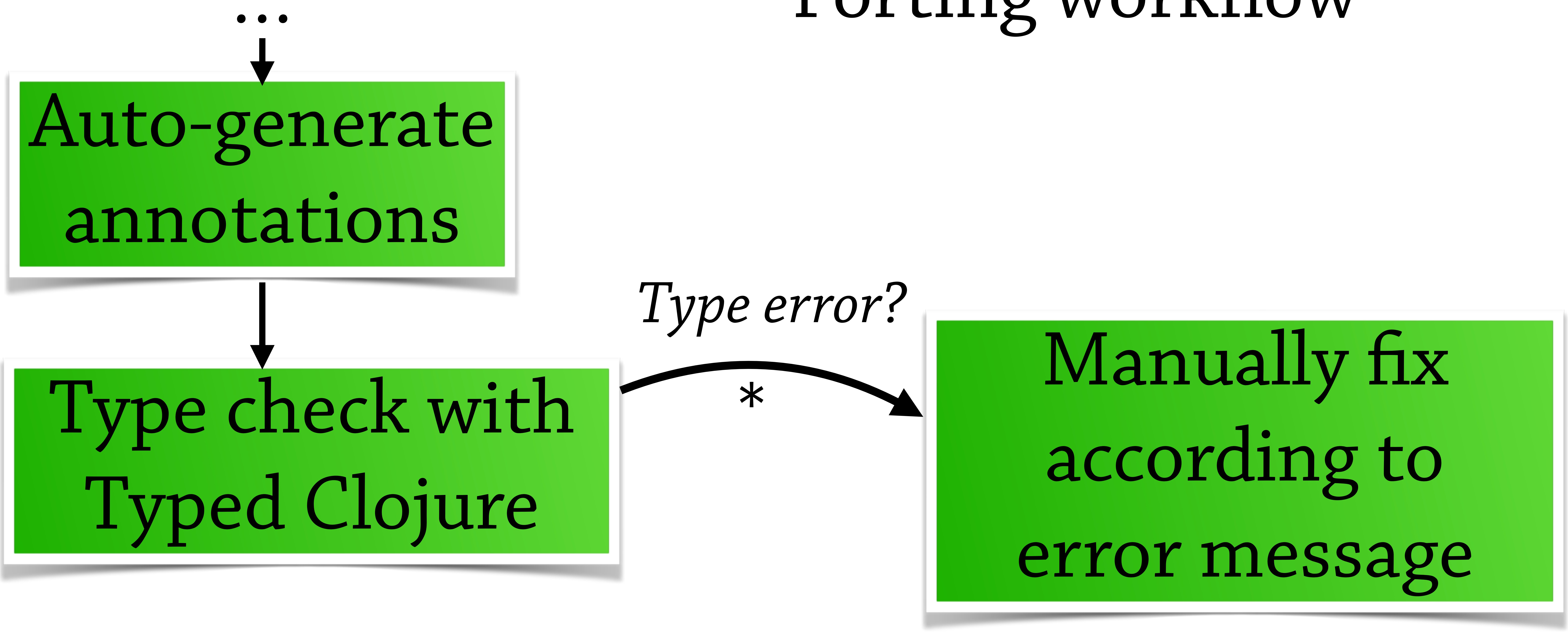
Porting workflow



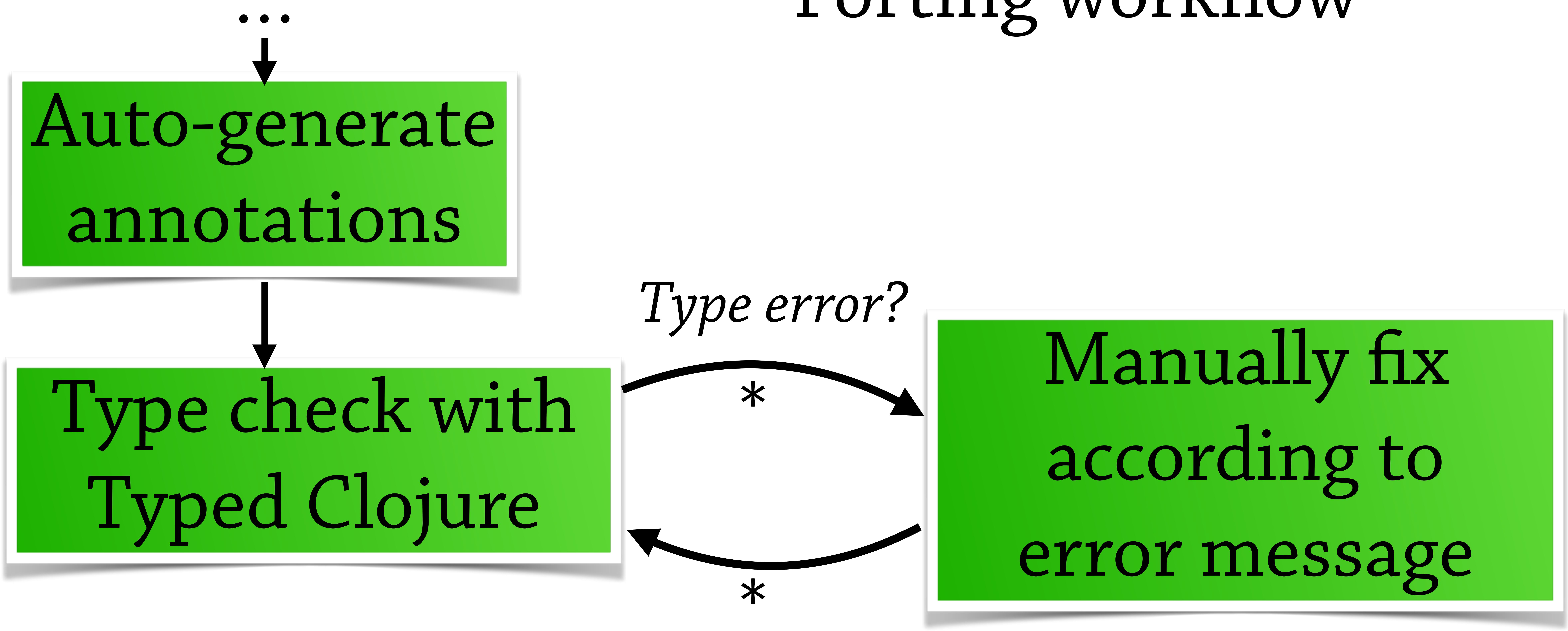
Porting workflow



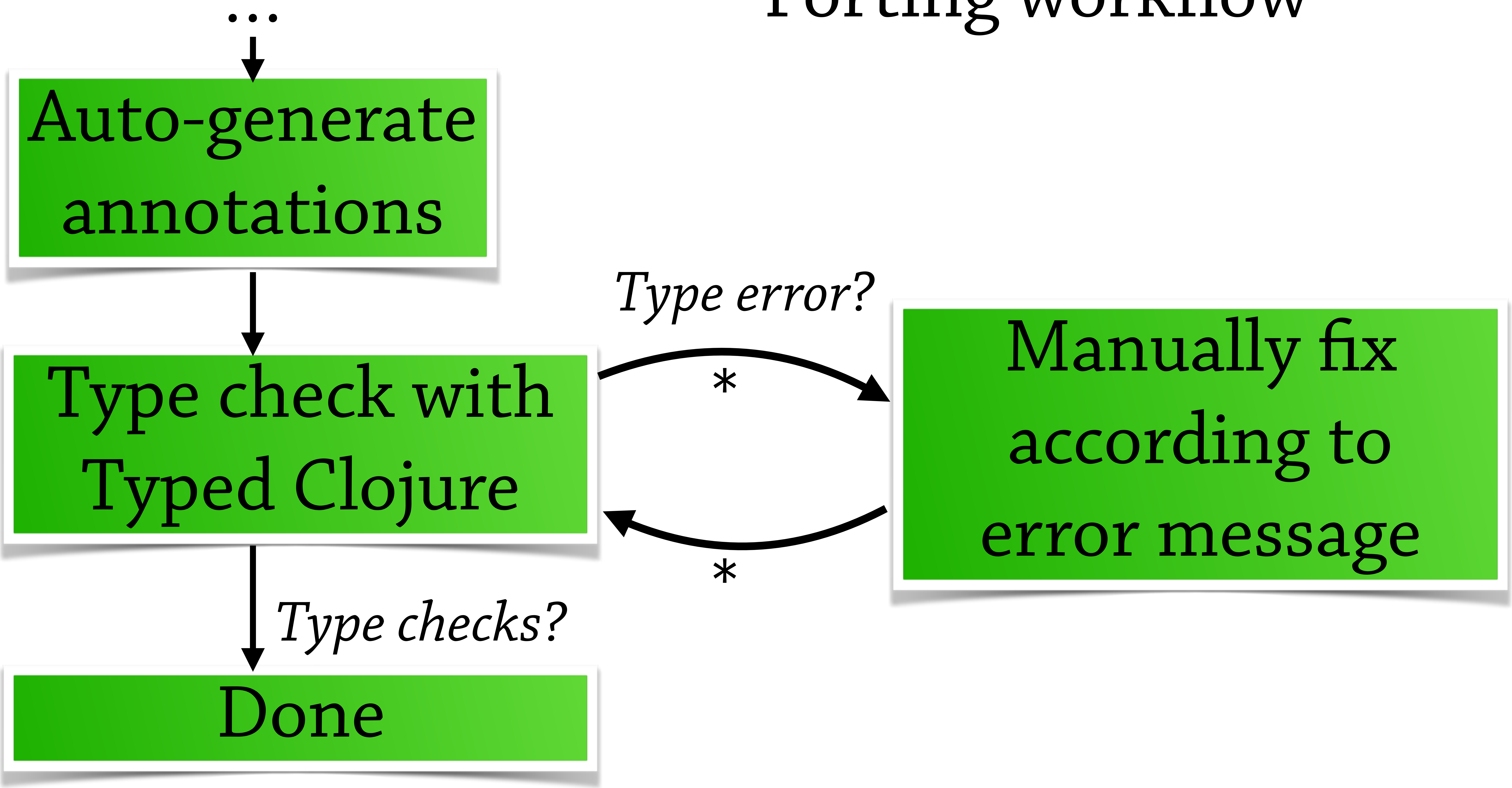
Porting workflow

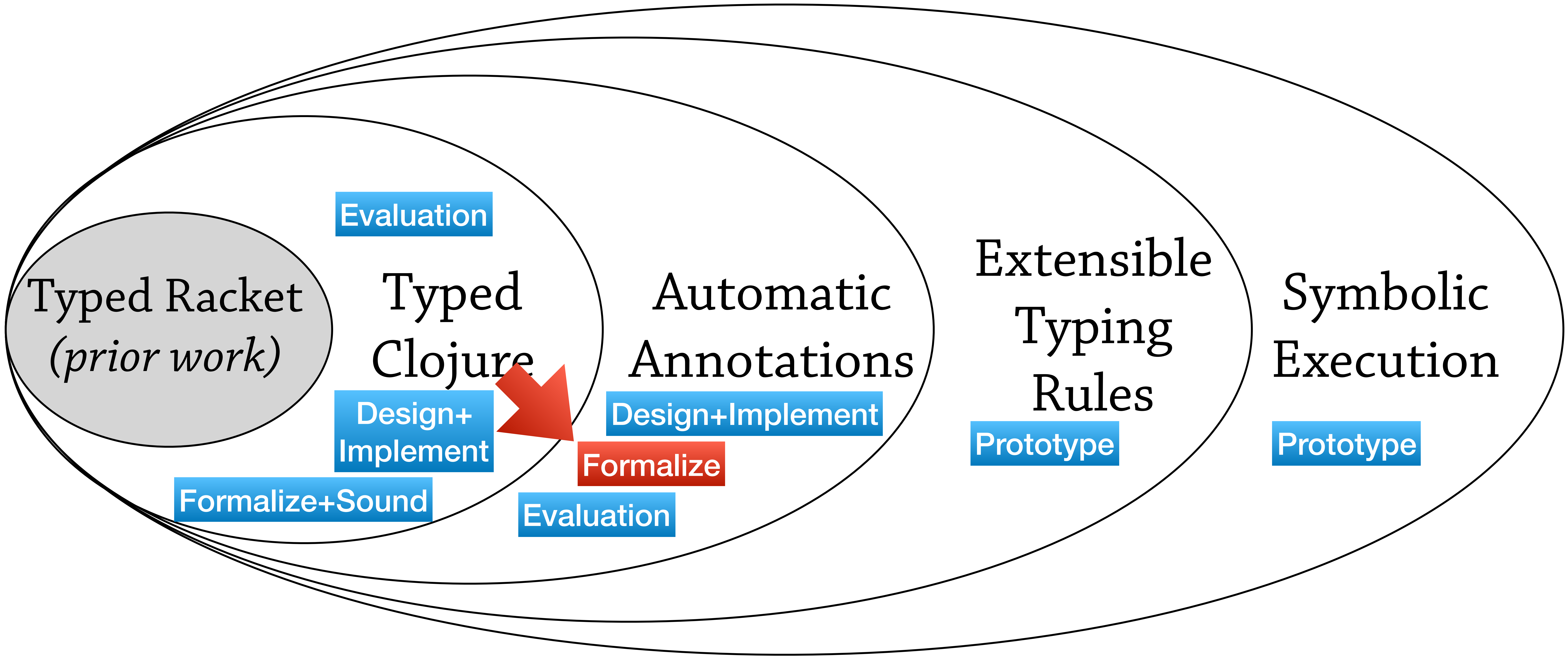


Porting workflow



Porting workflow




Typed Racket
(prior work)
Evaluation
Typed
Clojure
Design+
Implement
Formalize+Sound
Automatic
Annotations
Design+Implement
Formalize
Evaluation
Extensible
Typing
Rules
Prototype
Symbolic
Execution
Prototype

λtrack

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

annotate $=$ infer $\circ$ collect

λtrack

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

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

"Track and annotate x's in program e"

λ_{track}

λtrack

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

Definition

λtrack

$\texttt{define } f = \lambda m.(\text{get } m \text{ :a})$ **Definition**

$(f\ \{\text{:a}\ 42\})\ \texttt{=>}\ 42$ **Test**

λtrack

define f = $\lambda m.$(get m :a)

Definition

(f {:a 42}) => 42

Test

annotate((f {:a 42}), [f]) = {f : [{:a N} → N]}

λtrack

define f = $\lambda m.$(get m :a)

Definition

(f {:a 42}) => 42

Test

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

Test

λtrack

define f = $\lambda m.$(get m :a)

Definition

(f {:a 42}) => 42

Test

annotate((f {:a 42}), [f]) = {f : [{:a N} → N]}

Test

Track-me

λtrack

$\texttt{define}\ f\ \texttt{=}\ \lambda m.(\mathsf{get}\ \ m\ \ \mathsf{:a})$ Definition

$(f\ \ \{\mathsf{:a}\ \ 42\})\ \ \texttt{=>}\ \ 42$ Test

$\mathsf{annotate}((f\ \ \{\mathsf{:a}\ 42\}), [f]) = \{f : [\{\mathsf{:a}\ \mathsf{N}\} \to \mathsf{N}]\}$

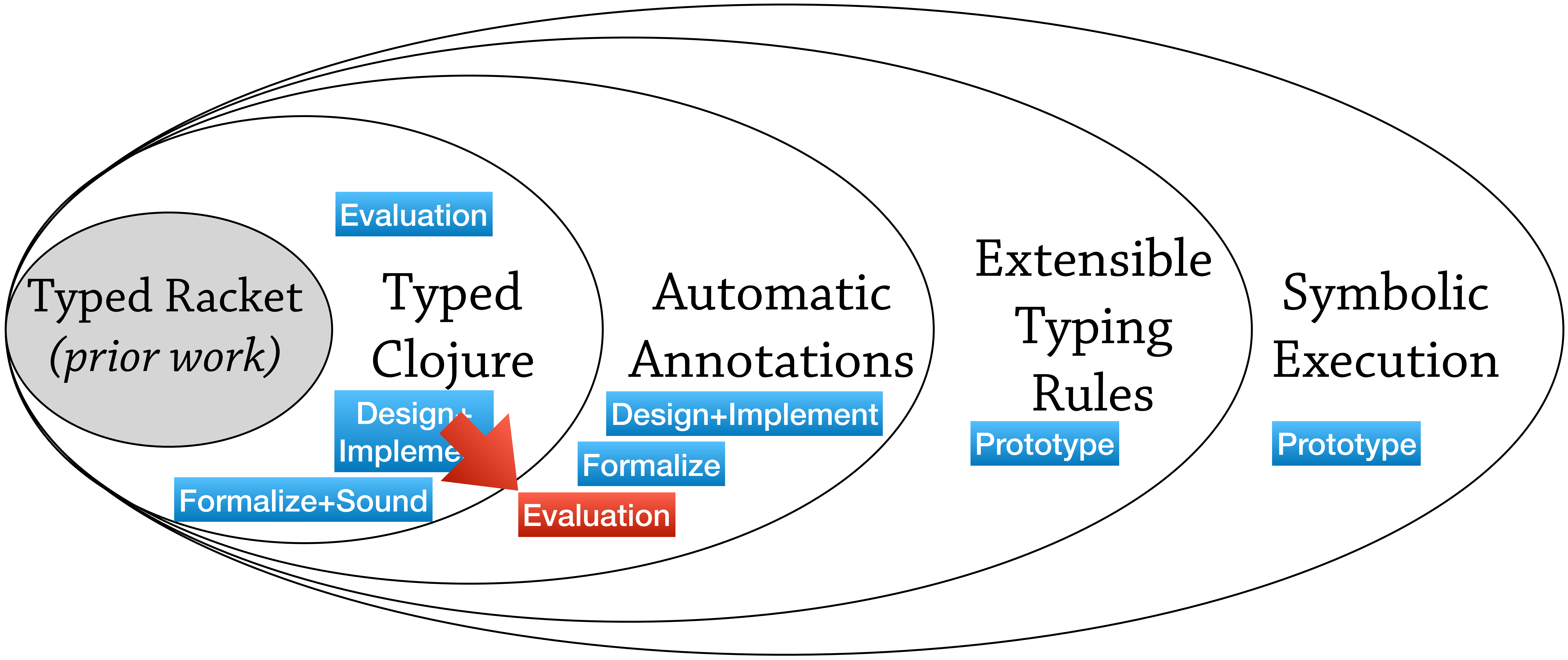
Test Track-me Derived type

λtrack

Intentionally unsound

Aggressively combines
types to create compact aliases
and recursive types

Tailored for the workflow


Evaluation
Typed Racket
(prior work)
Typed
Clojure
Design+
Impleme
Formalize+Sound
Automatic
Annotations
Design+Implement
Formalize
Evaluation
Extensible
Typing
Rules
Prototype
Symbolic
Execution
Prototype

Evaluation

Ported 5 open-source programs (~1500 LOC)

Measured the kinds of manual changes needed

Auto-generated types

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

Auto-generated types

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

Manual changes

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

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)])
```

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

Manual changes

Has an interesting type

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

```
(defalias 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}))

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

Auto-generated types

Has an interesting type

Manual changes

Auto-generated types

Has an interesting type

```
(defalias 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

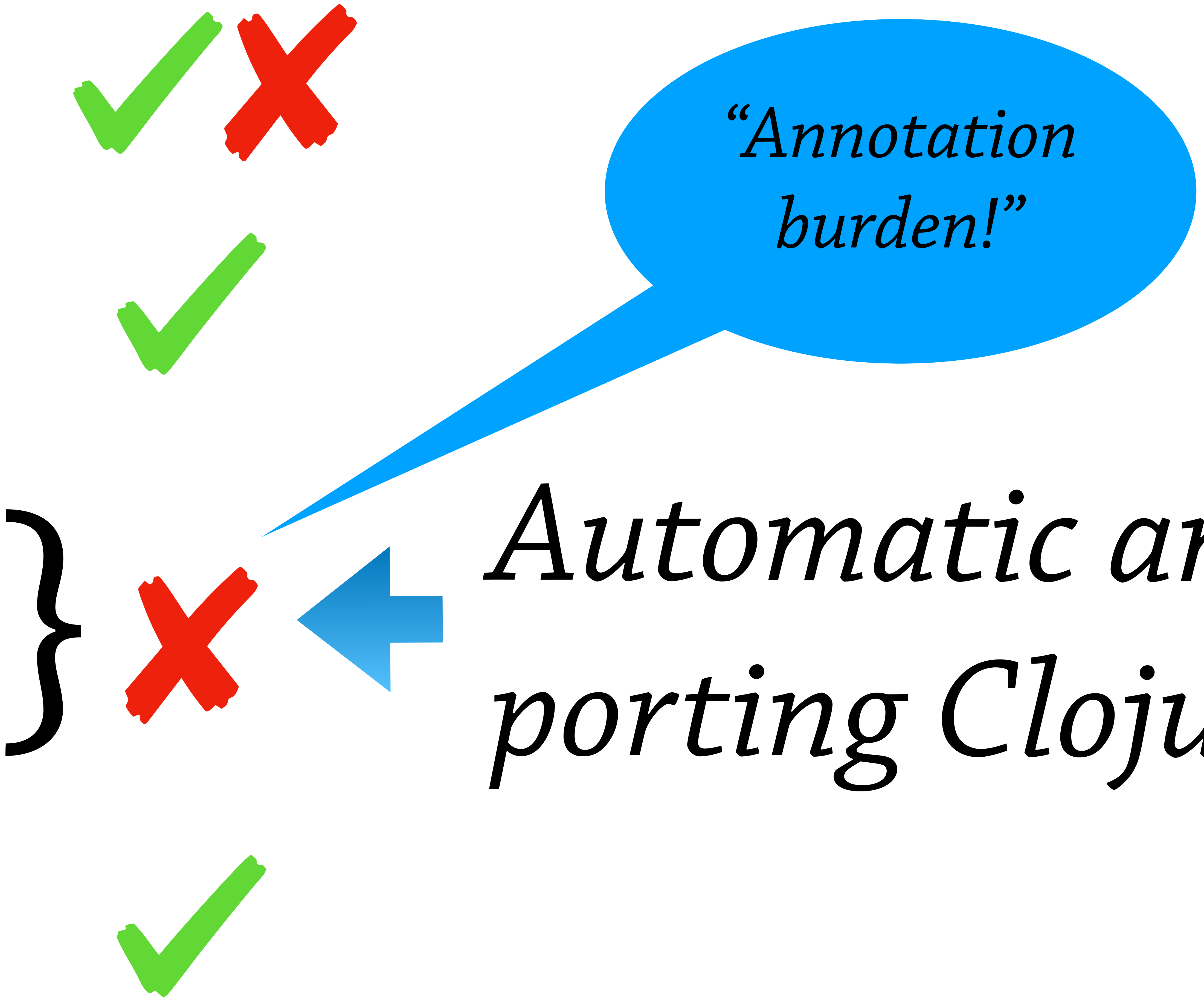
Functional programming

Immutability

The REPL

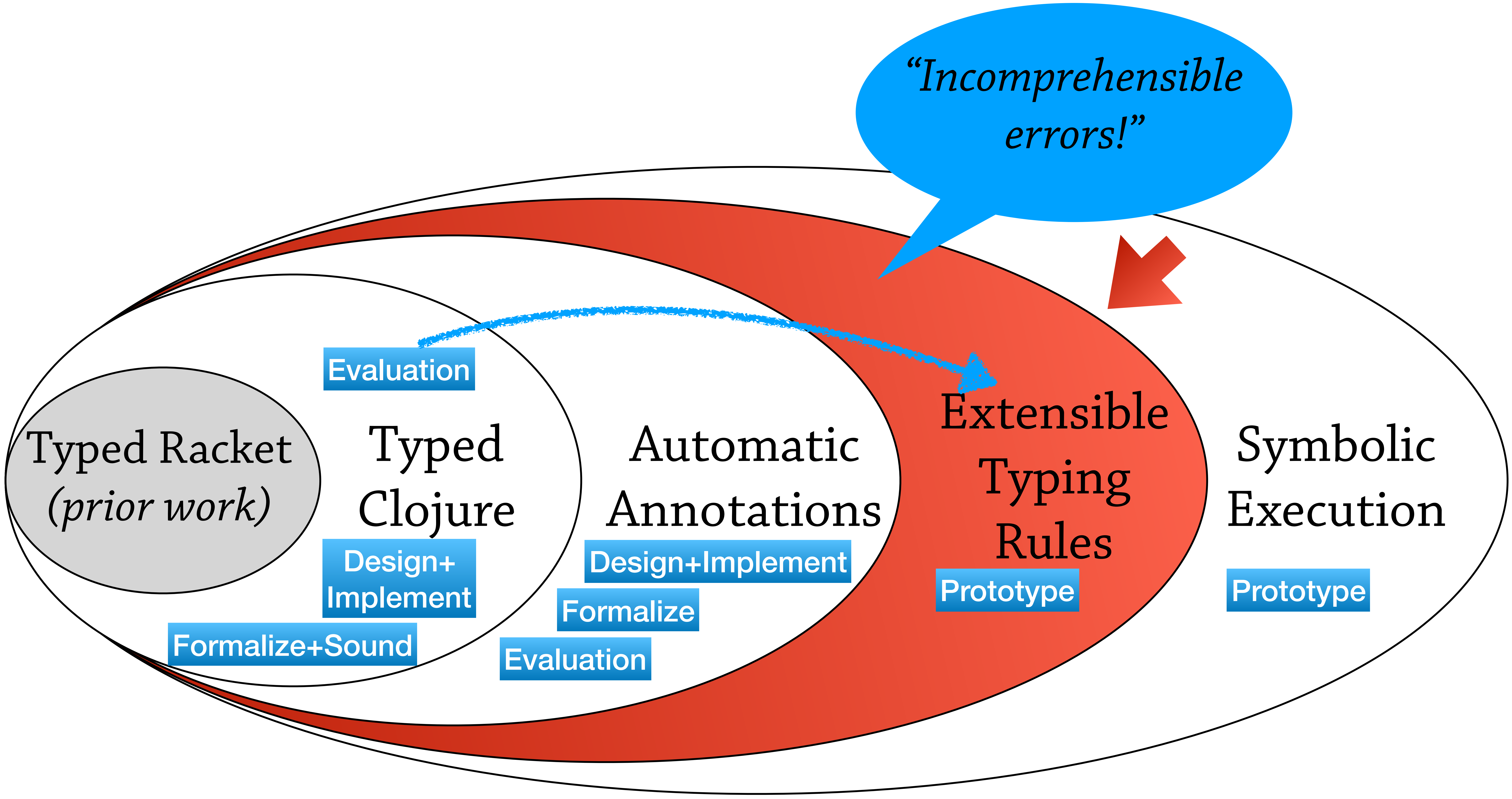
Ease of development

Host Interop



Automatic annotations makes porting Clojure programs easier

Part III
Extensible Typing Rules


"Incomprehensible errors!"
Evaluation
Typed Racket (prior work)
Typed Clojure
Design+ Implement
Formalize+Sound
Automatic Annotations
Design+Implement
Formalize
Evaluation
Extensible Typing Rules
Prototype
Symbolic Execution
Prototype

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

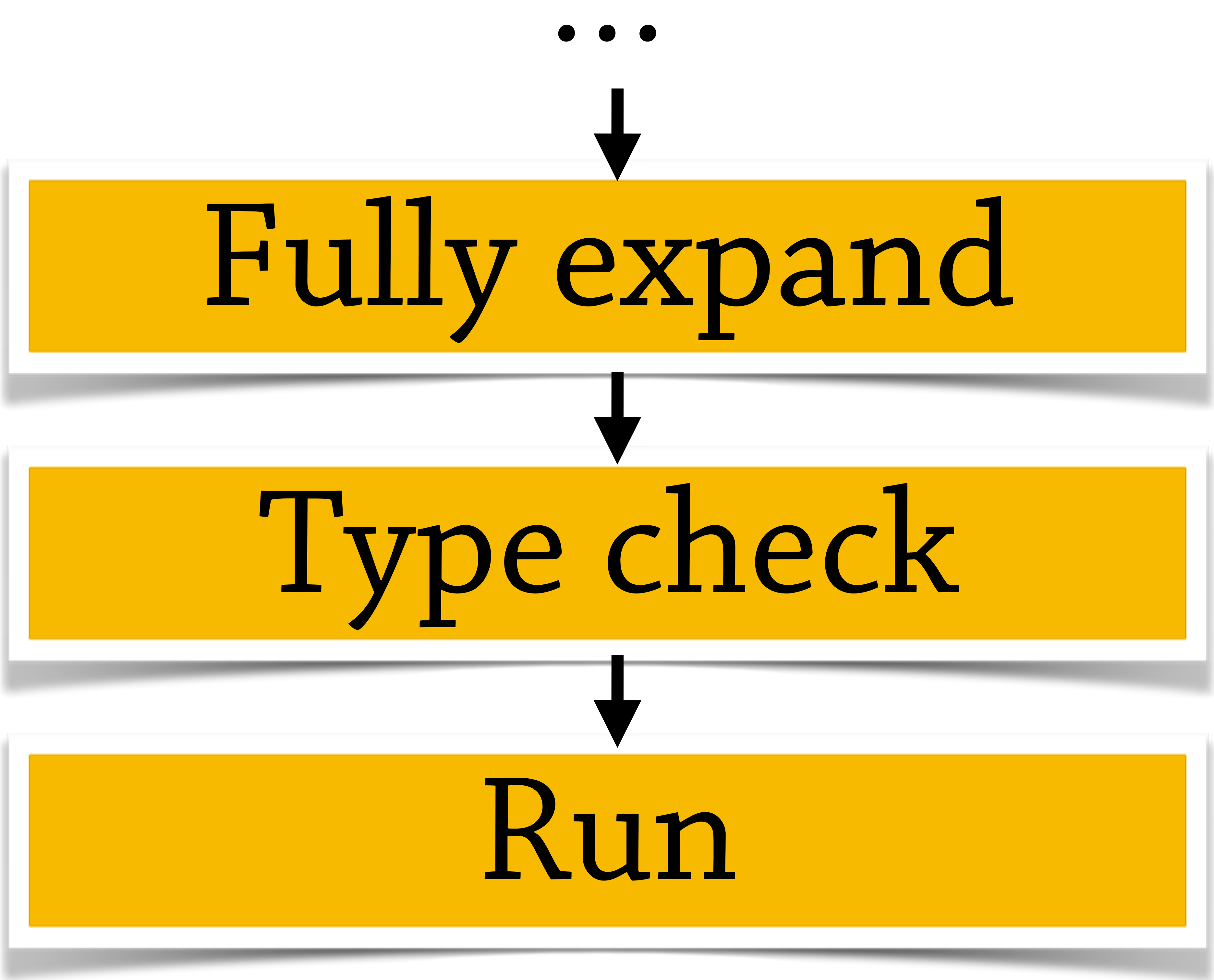
```
(for [a [1 2 3]]
  (inc a))
```

Idea

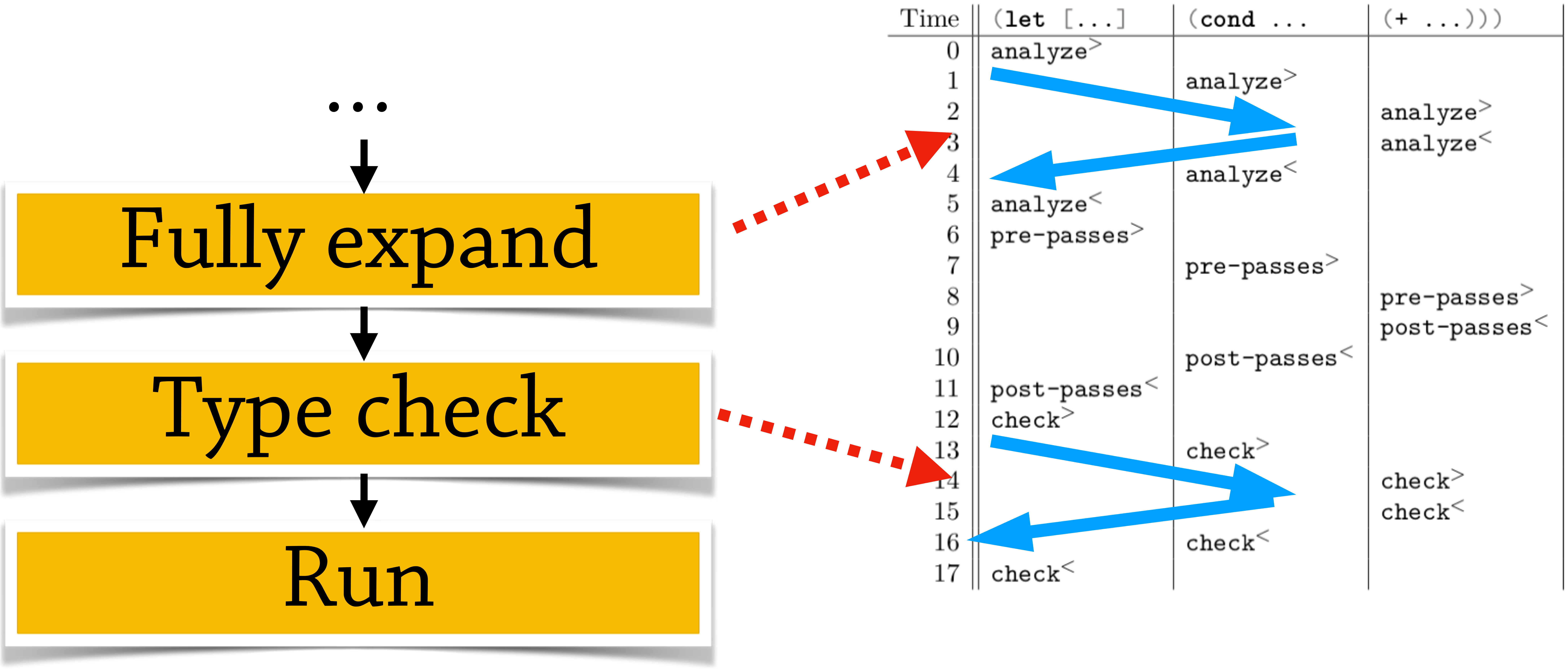
```
(for [a [1 2 3]]
  (inc a))
```

Allow the user to define custom typing rules for macros

Roadblock:
Expansion comes *before* check

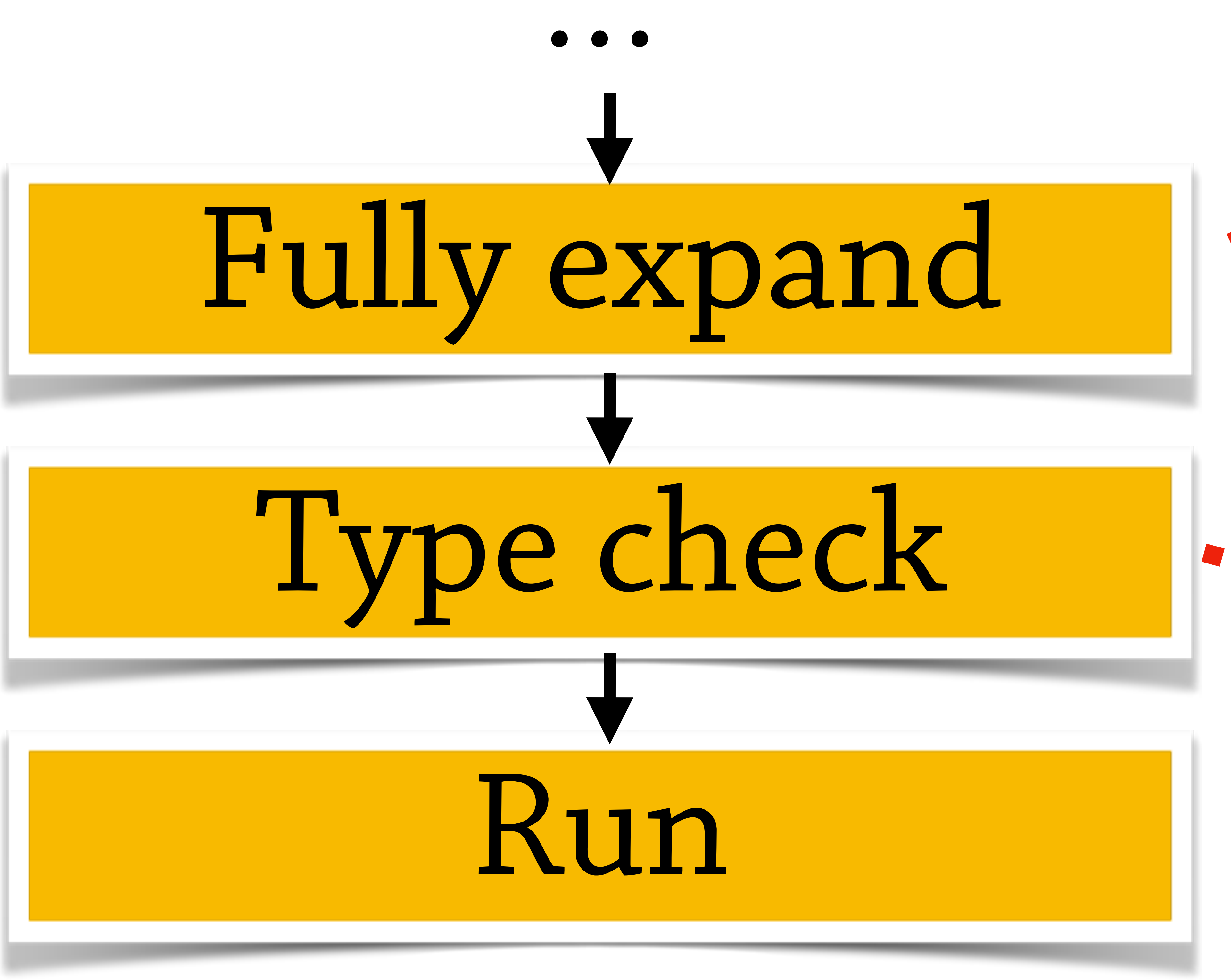


Roadblock:
Expansion comes *before* check



Time	(let [...]	(cond ...	(+ ...)))
0	analyze$^{>}$		
1		analyze$^{>}$	
2			analyze$^{>}$
3			analyze$^{<}$
4		analyze$^{<}$	
5	analyze$^{<}$		
6	pre-passes$^{>}$		
7		pre-passes$^{>}$	
8			pre-passes$^{>}$
9			post-passes$^{<}$
10		post-passes$^{<}$	
11	post-passes$^{<}$		
12	check$^{>}$		
13		check$^{>}$	
14			check$^{>}$
15			check$^{<}$
16		check$^{<}$	
17	check$^{<}$		

Roadblock:
Expansion comes *before* check

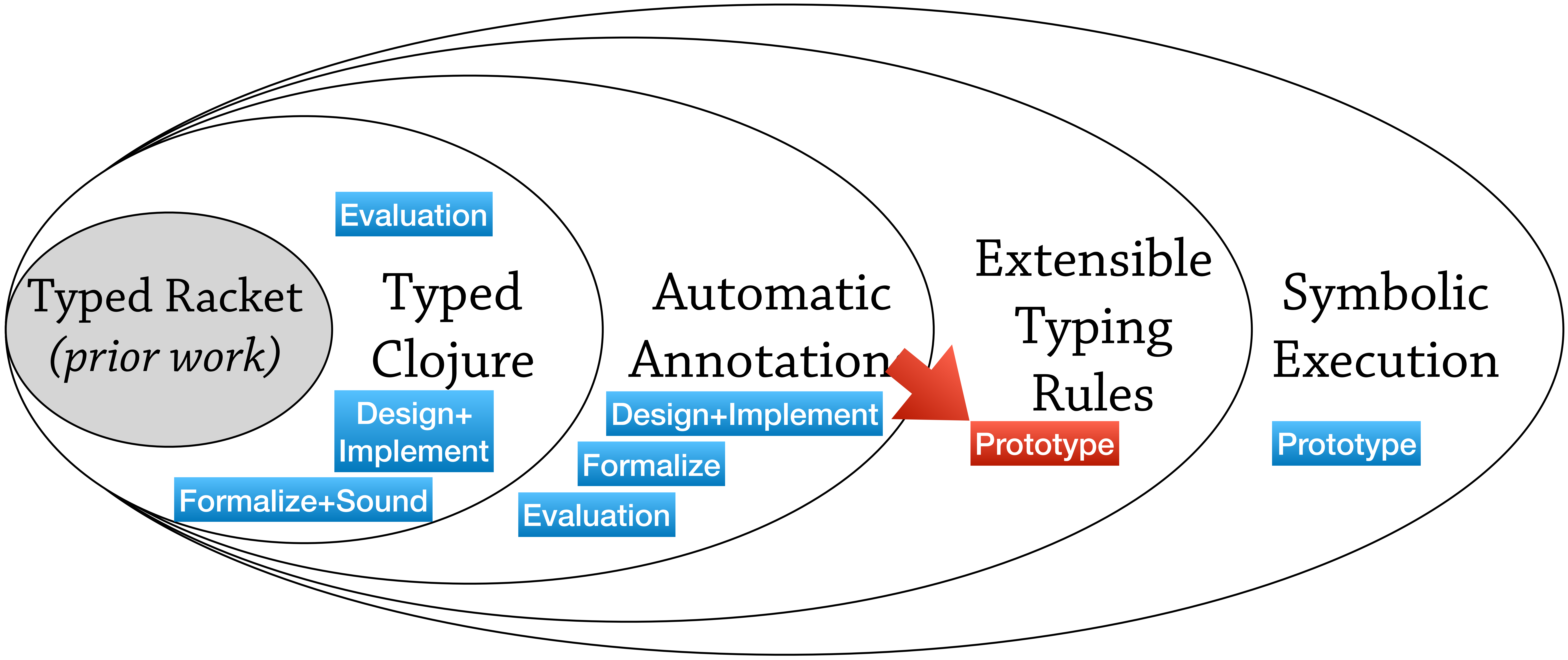


Time	(let [...]	(cond ...	(+ ...)))
0	analyze$^{>}$		
1		analyze$^{>}$	
2			analyze$^{>}$
3			analyze$^{<}$
4		analyze$^{<}$	
5	analyze$^{<}$		
6	pre-passes$^{>}$		
7		pre-passes$^{>}$	
8			pre-passes$^{>}$
9			post-passes$^{<}$
10		post-passes$^{<}$	
11	post-passes$^{<}$		
12	check$^{>}$		
13		check$^{>}$	
14			check$^{>}$
15			check$^{<}$
16		check$^{<}$	
17	check$^{<}$		

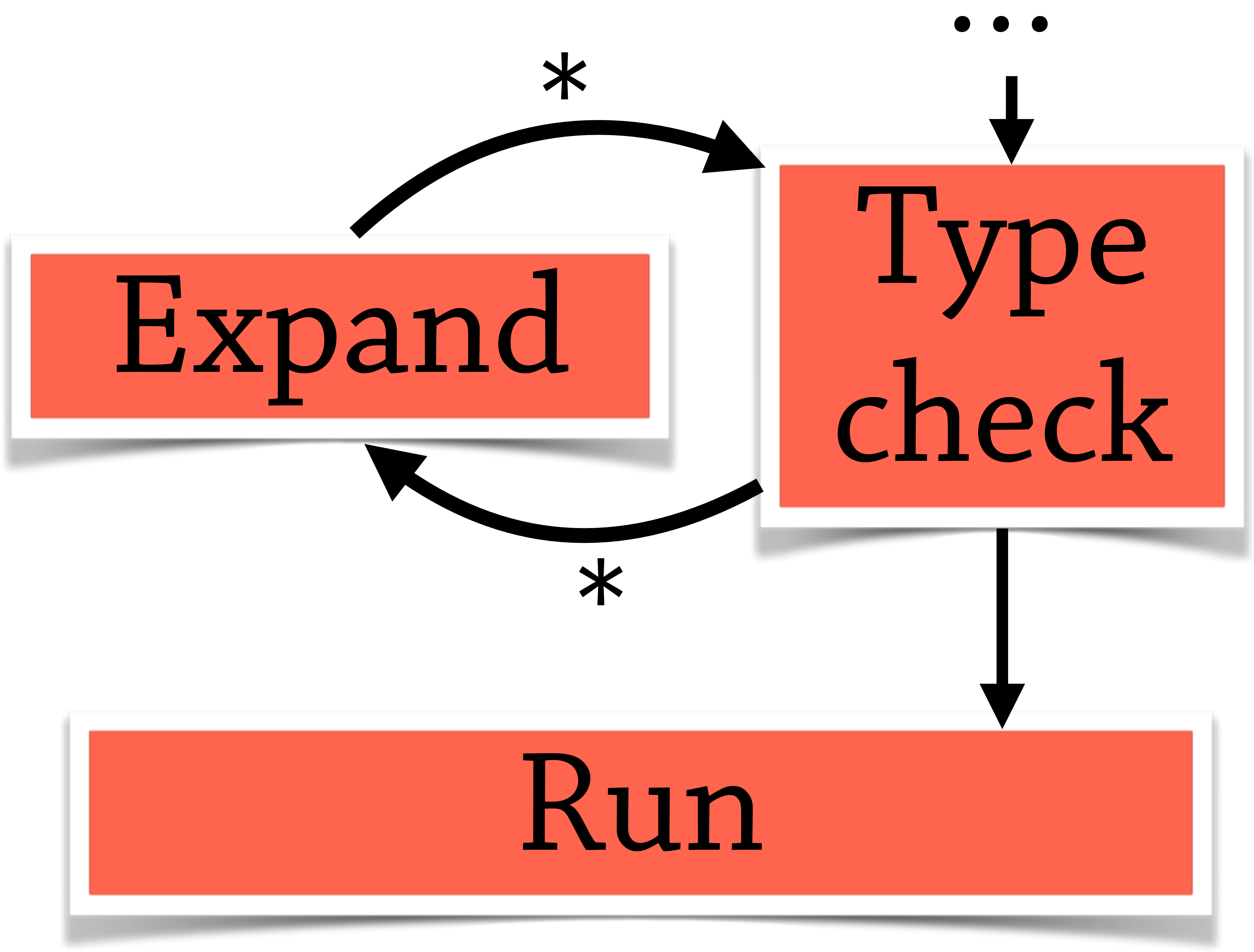
Already expanded!

Solution

Allow Typed Clojure to interleave macroexpansion and type checking


Evaluation
Typed Racket
(prior work)
Typed
Clojure
Design+
Implement
Formalize+Sound
Automatic
Annotation
Design+Implement
Formalize
Evaluation
Extensible
Typing
Rules
Prototype
Symbolic
Execution
Prototype

Checker controls expansion



I wrote a new Clojure code analyzer

Time	(let [...]	(cond ...	(+ ...)))
0	unanalyzed$^{>}$		
1	analyze-outer*		
2	run-pre-passes$^{>}$		
3	check$^{>}$		
4		analyze-outer*	
5		run-pre-passes$^{>}$	
6		check$^{>}$	
7			analyze-outer*
8			run-pre-passes$^{>}$
9			check$^{>}$
10			run-post-passes$^{<}$
11			check$^{<}$
12		run-post-passes$^{<}$	
13		check$^{<}$	
14	run-post-passes$^{<}$		
15	check$^{<}$		

Expand as needed

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

If partially expanded…

```
(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

} ✗

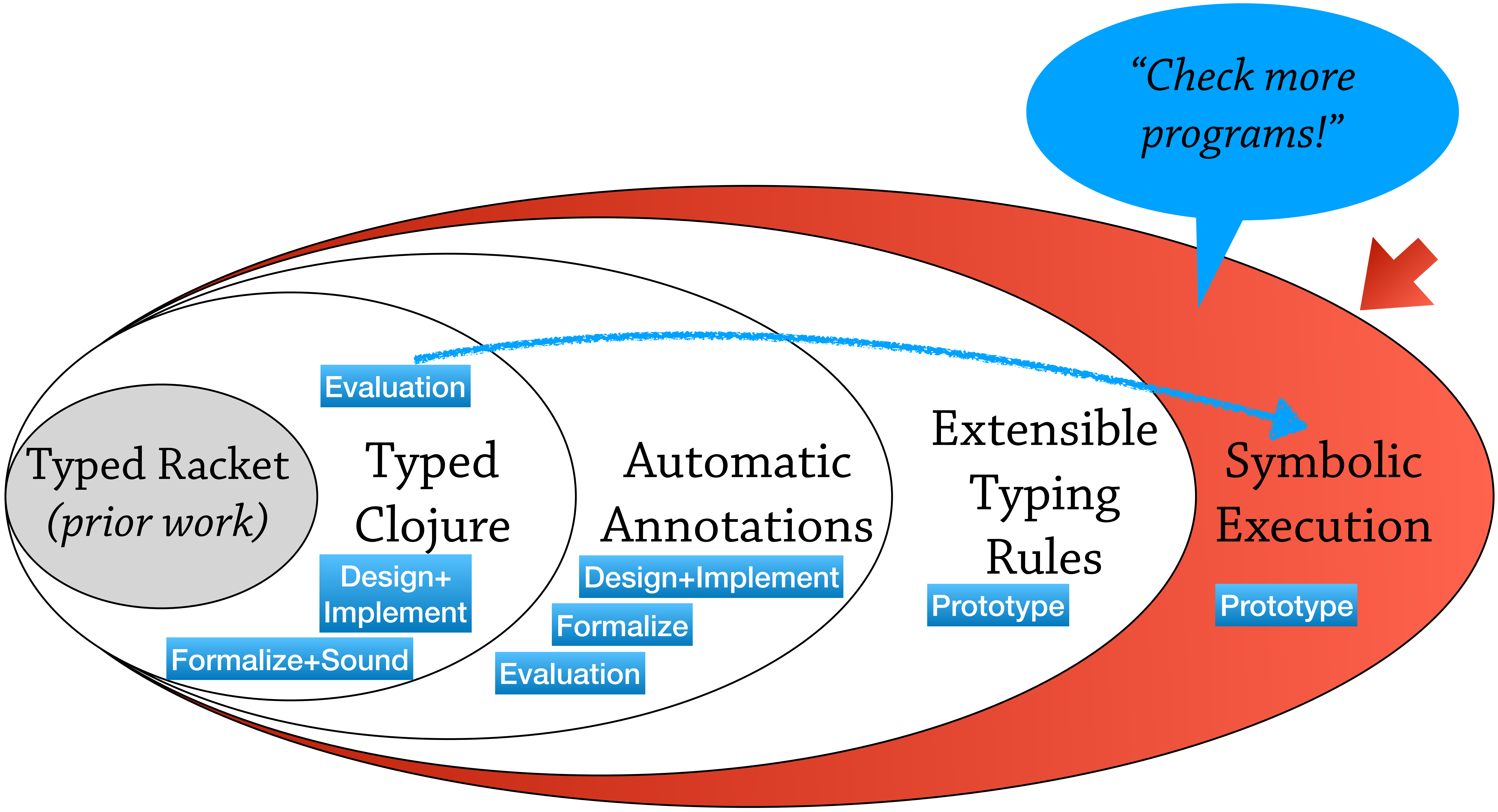
Host Interop ✓

"Incomprehensible errors!"

Extensible rules Prototype:

Improve errors, check more programs

Part VI
Symbolic Execution


"Check more programs!"
Typed Racket (prior work)
Typed Clojure
Evaluation
Design+ Implement
Formalize+Sound
Automatic Annotations
Design+Implement
Formalize
Evaluation
Extensible Typing Rules
Prototype
Symbolic Execution
Prototype

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)]
  (f 1))
```

```
(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))
```

```
(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

```
(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)])
```

Must have type of p here

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)]
  (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)])
```

Approach

New type rule for checking (unannotated) functions:

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

Approach

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

Approach

New type rule for checking (unannotated) functions:

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

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 : Γ@(fn [x] x)
  (f 1))
```

Application rule?

Approach

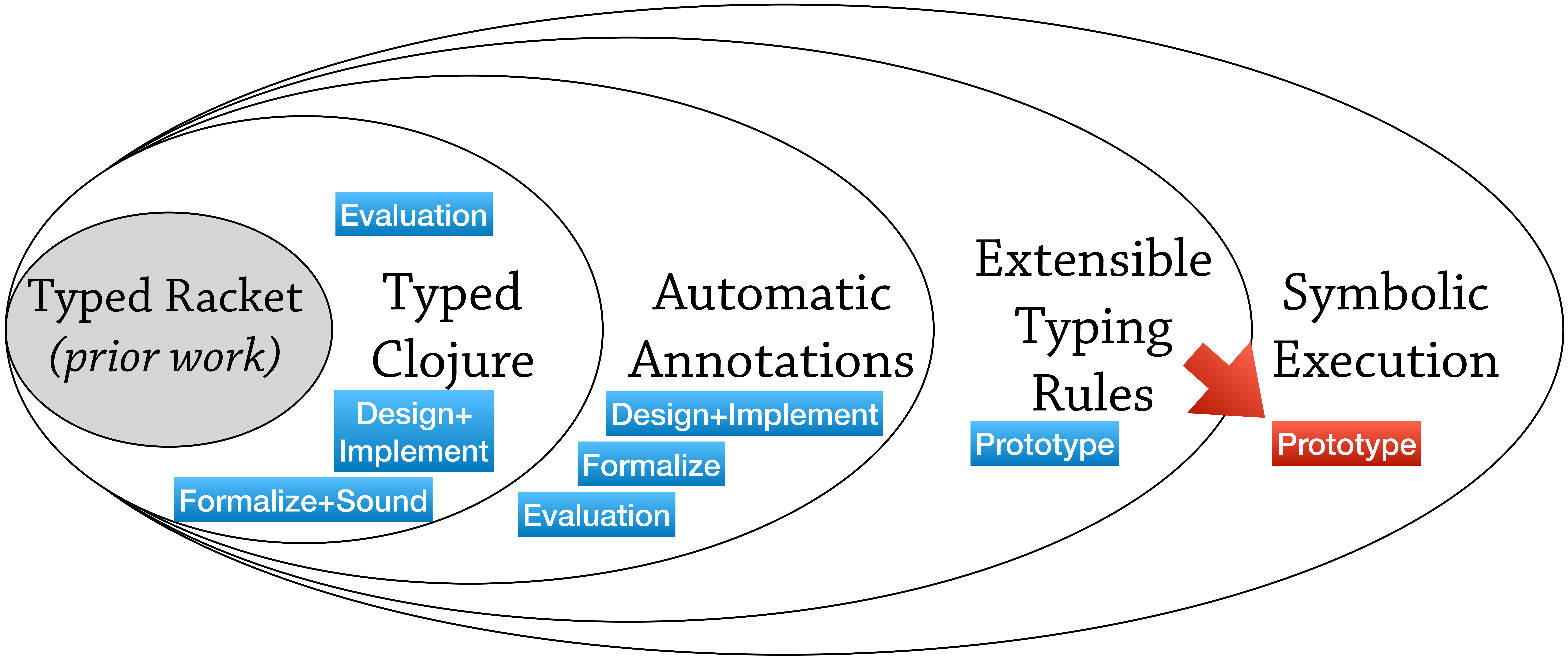
```
(let [f (fn [x] x)]
  ; f : Γ@(fn [x] x)
  (f 1))
```

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


Evaluation
Typed Racket
(prior work)
Typed
Clojure
Design+
Implement
Formalize+Sound
Automatic
Annotations
Design+Implement
Formalize
Evaluation
Extensible
Typing
Rules
Prototype
Symbolic
Execution
Prototype

Naive formalism

UAbs

$$\frac{}{\Gamma \vdash \lambda(x)f : \Gamma@\lambda(x)f}$$

UApp

$$\frac{\Gamma' \vdash e_1 : \Gamma@\lambda(x)f \qquad \Gamma' \vdash e_2 : \sigma \qquad \Gamma, x{:}\sigma \vdash f : \tau}{\Gamma' \vdash e_1(e_2) : \tau}$$

Naive formalism

$$\textsc{UAbs}\quad \frac{}{\Gamma \vdash \lambda(x)f : \Gamma@\lambda(x)f}$$

$$\textsc{UApp}\quad \frac{\Gamma' \vdash e_1 : \Gamma@\lambda(x)f \qquad \Gamma' \vdash e_2 : \sigma \qquad \Gamma, x{:}\sigma \vdash f : \tau}{\Gamma' \vdash e_1(e_2) : \tau}$$

Naive formalism

UAbs

$$\frac{}{\Gamma \vdash \lambda(x)f : \Gamma@\lambda(x)f}$$

UApp

$$\frac{\Gamma' \vdash e_1 : \Gamma@\lambda(x)f \qquad \Gamma' \vdash e_2 : \sigma \qquad \Gamma, x{:}\sigma \vdash f : \tau}{\Gamma' \vdash e_1(e_2) : \tau}$$

Naive formalism

$$\textsc{UAbs}\quad \frac{}{\Gamma \vdash \lambda(x)f : \Gamma@\lambda(x)f}$$

$$\textsc{UApp}\quad \frac{\Gamma' \vdash e_1 : \Gamma@\lambda(x)f \qquad \Gamma' \vdash e_2 : \sigma \qquad \Gamma, x{:}\sigma \vdash f : \tau}{\Gamma' \vdash e_1(e_2) : \tau}$$

Naive formalism

UAbs

$$\frac{}{\Gamma \vdash \lambda(x)f : \Gamma@\lambda(x)f}$$

UApp

$$\frac{\Gamma' \vdash e_1 : \Gamma@\lambda(x)f \qquad \Gamma' \vdash e_2 : \sigma \qquad \Gamma, x{:}\sigma \vdash f : \tau}{\Gamma' \vdash e_1(e_2) : \tau}$$

Naive formalism

UAbs

$$\frac{}{\Gamma \vdash \lambda(x)f : \Gamma@\lambda(x)f}$$

UApp

$$\frac{\Gamma' \vdash e_1 : \Gamma@\lambda(x)f \qquad \Gamma' \vdash e_2 : \sigma \qquad \Gamma, x{:}\sigma \vdash f : \tau}{\Gamma' \vdash e_1(e_2) : \tau}$$

Prototype Implementation

Prototype Implementation

```
(tc ? 1)
=> Int
```

Prototype Implementation

```
(tc ? 1)
=> Int

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

Prototype Implementation

```
(tc ? 1)
=> Int

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

Prototype Implementation

```
(tc ? 1)
=> Int

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

Prototype Implementation

```
(tc ? 1)
=> Int

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

(tc ? (fn [x] x))
=> (Closure {} (fn [x] x))
```

Prototype Implementation

```
(tc ? 1)
=> Int

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

(tc ? (fn [x] x))
=> (Closure {} (fn [x] x))


(tc ? ((fn [x] x) 1))
=> Int
```

Prototype Implementation

```
(tc ? 1)
=> Int

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

(tc ? (fn [x] x))
=> (Closure {} (fn [x] x))

(tc ? ((fn [x] x) 1))
=> Int
```

Prototype Implementation

```
(tc ? 1)
=> Int

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

(tc ? (fn [x] x))
=> (Closure {} (fn [x] x))

(tc ? ((fn [x] x) 1))
=> 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)
```

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

GR

```
(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))
```

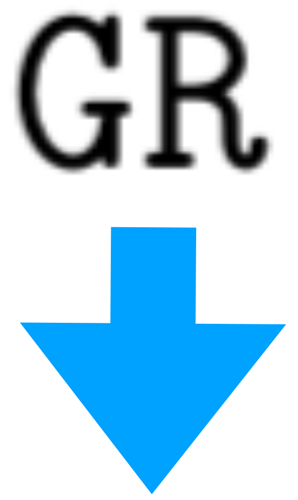
[1] LICS'88, Giannini & Rocca

Prototype Implementation

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

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

```
(GR (fn [_] (fn [_] 42)))          ;=> 42
(GR (fn [_] (fn [_] “hello”))) ;=> “hello”
```




```
(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))
```

[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

GR

```
(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))
```

```
(GR (fn [_] (fn [_] 42)))        ;=> 42
(GR (fn [_] (fn [_] “hello”))) ;=> “hello”
```

Challenge: Type check this quirky identity function

```
(ann id (All [a] [a -> a]))
(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

GR

```
(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


```
(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


```
(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


```
(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


```
(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


```
(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

```
(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

```
(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

Functional programming

Immutability

The REPL

Ease of development

Host Interop

Symbolic closure prototype:

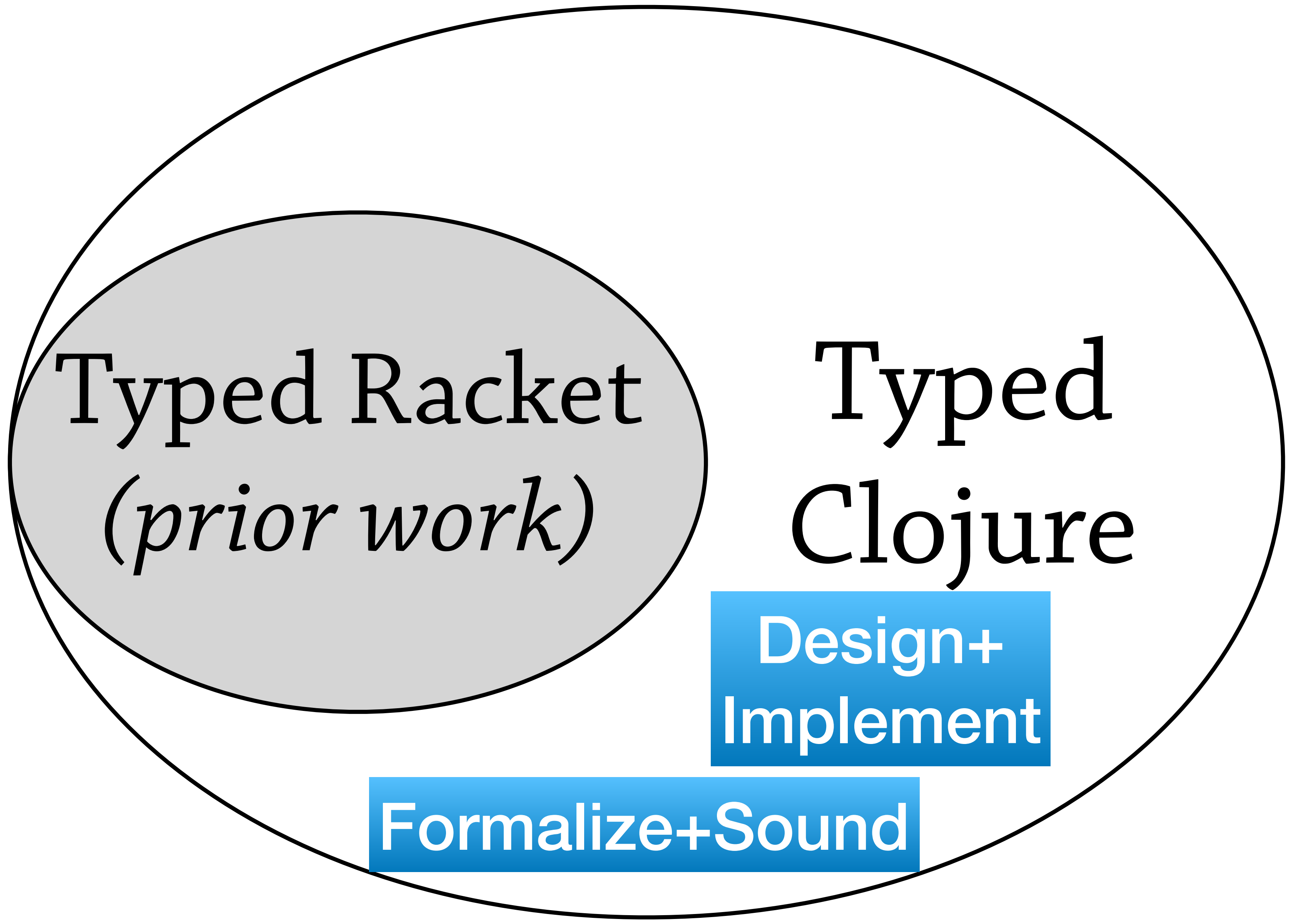
Checks more programs

"Check more programs!"

Conclusion

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

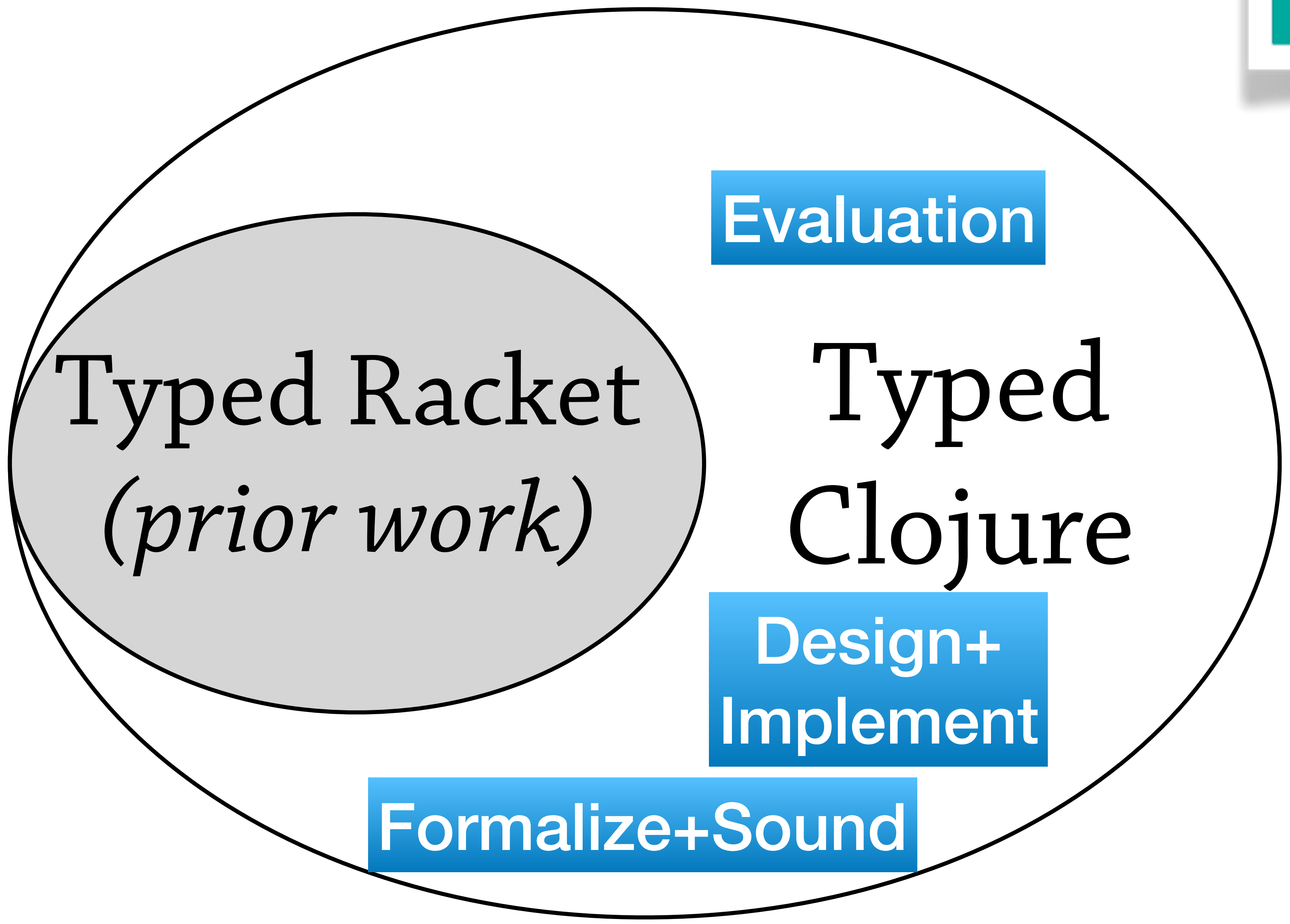
Typed Clojure is a
sound and **practical**
optional type system for Clojure



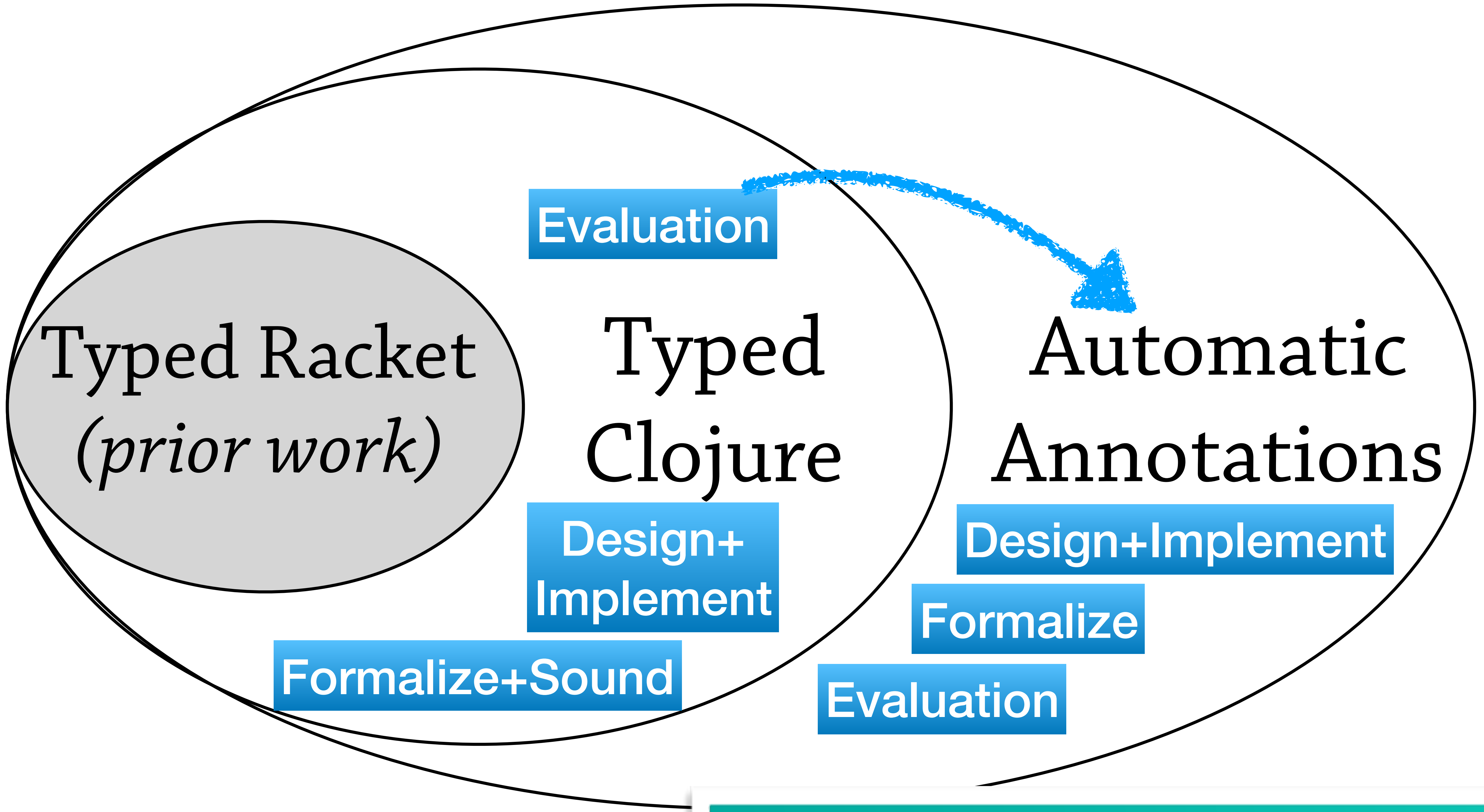
I present the **design** of Typed Clojure,
formalize the core type system, and prove it **sound**

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

I **empirically** show Typed Clojure's features correspond to **real-world** programs

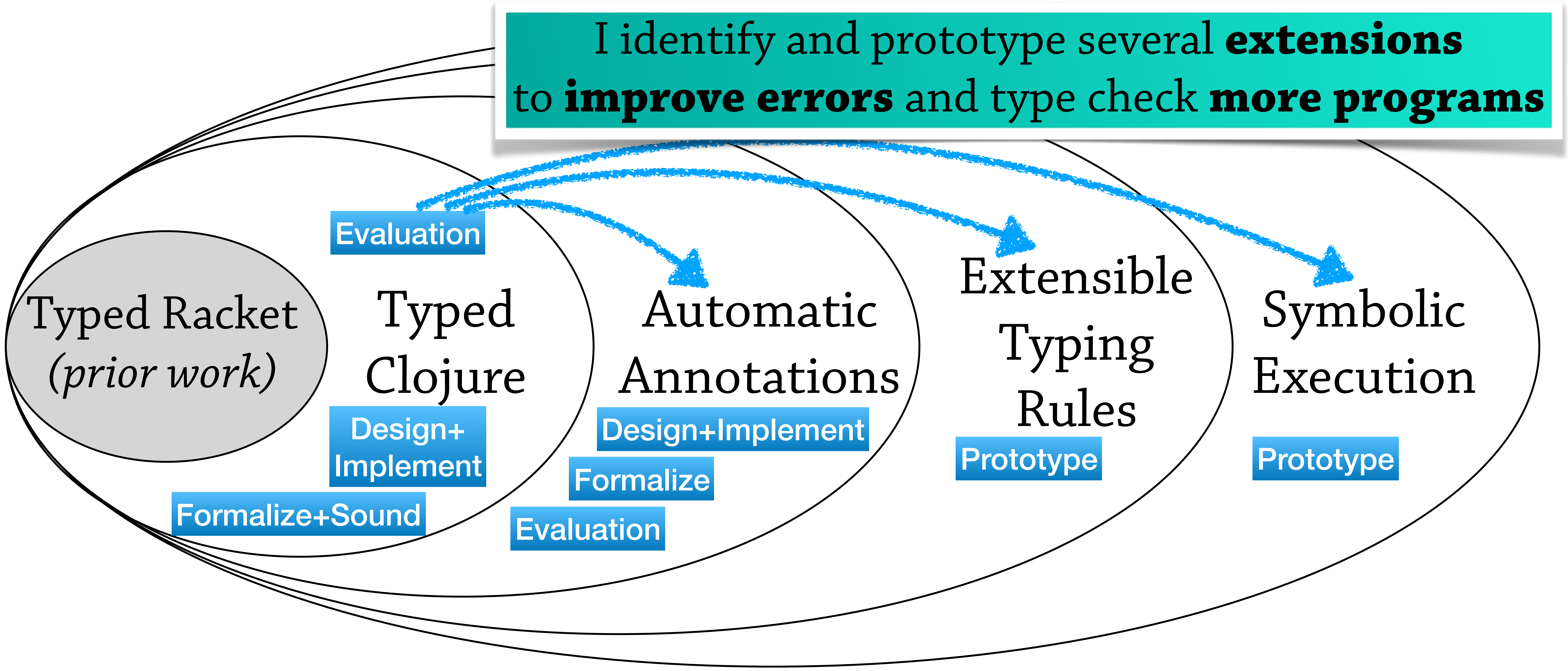


Typed Clojure is a
sound and **practical**
optional type system for Clojure



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



Thanks

Extra slides

λ_{TC} 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**