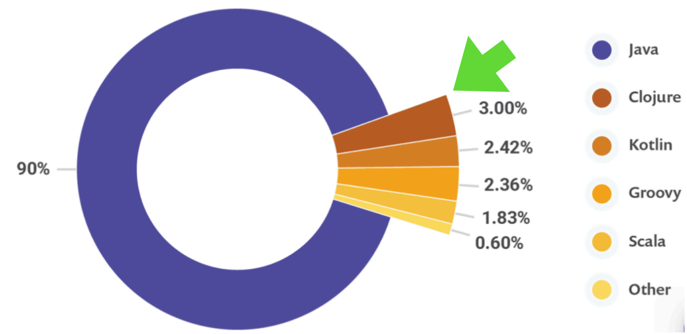
Typed Clojure in Theory and Practice

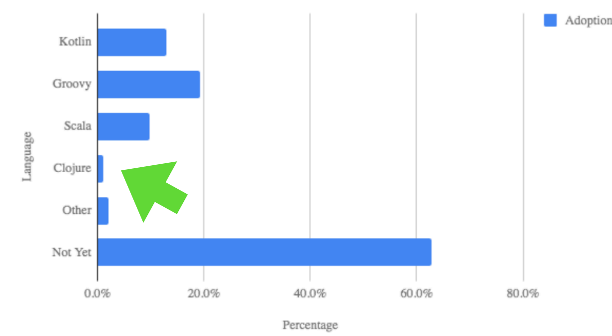
Ambrose Bonnaire-Sergeant

Hi, my name is Ambrose Bonnaire-Sergeant, and welcome to my talk. Today I will be defending my thesis, which is titled “Typed Clojure in Theory and Practice”

What is Clojure?

A programming language
running on the Java Virtual Machine



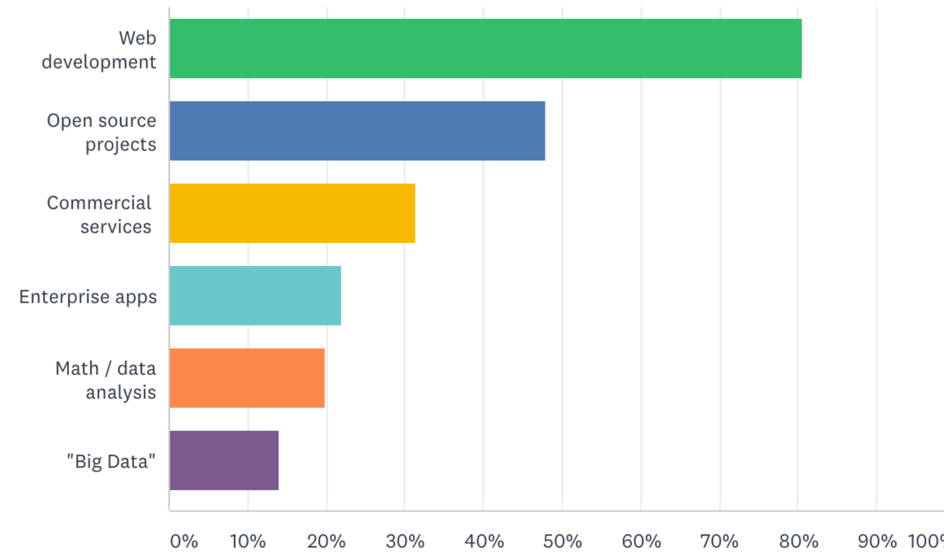


3% of JVM users' primary language is Clojure
- [JVM Ecosystem Report 2018, snyk.io]

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

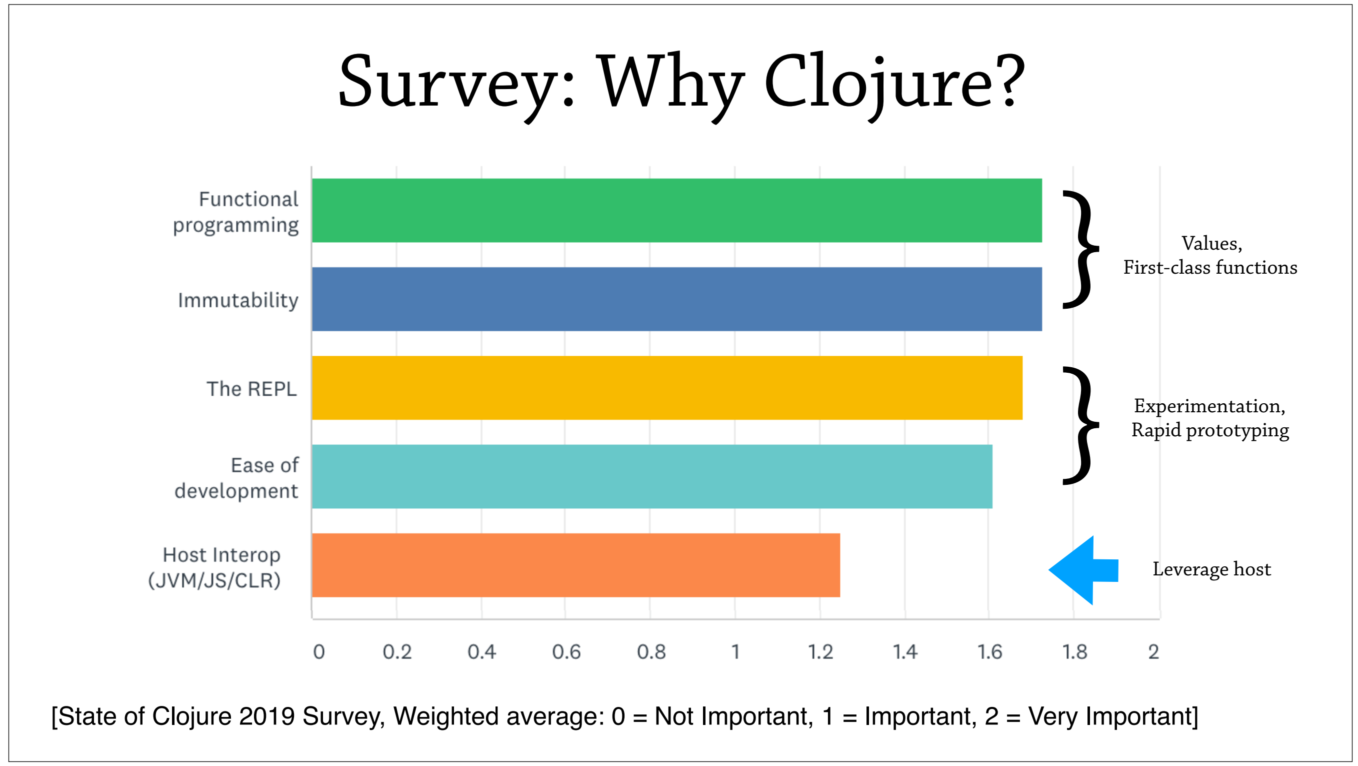
So, what is Clojure? Clojure is a programming language running on the Java Virtual Machine, so it runs wherever Java does. According to recent JVM surveys, Clojure has around 3%-1% market share of JVM users, so it's probably in the top 5 most popular languages on the JVM.

General Purpose

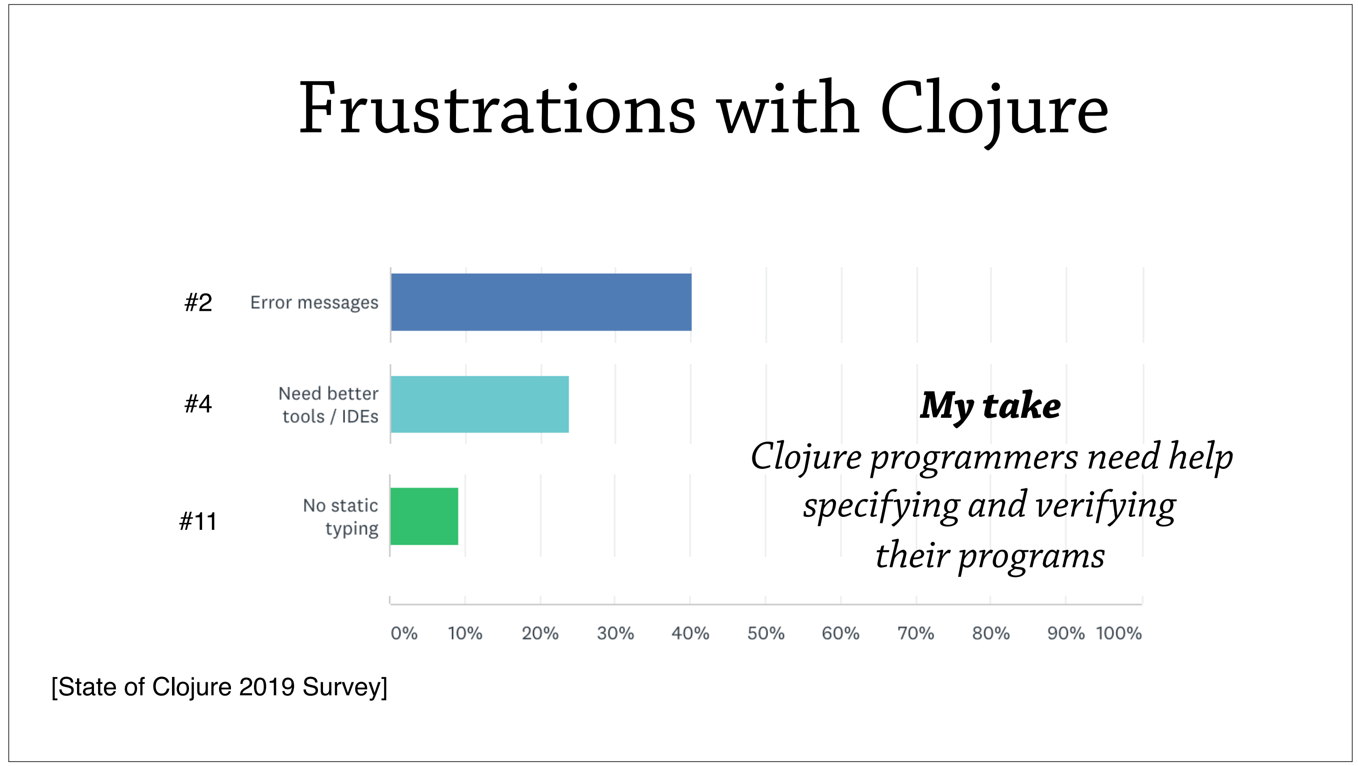


[State of Clojure 2019 Survey]

Clojure is designed to be a general purpose programming language, and is used in a wide variety of areas. A survey of around 2500 Clojure programmers earlier this year showed Clojure is mostly used to build Web applications, open source projects, and provide commercial services.



What makes Clojure worth choosing over other languages? The same survey asked this question, and had participants rate their favourite Clojure features from 0 (not important) to 2 (very important). The top-5 features are in three main groups. First, functional programming and immutability emphasise programming with values and first-class functions. Second, it is easy to experiment and prototype in Clojure using the REPL and other features. Third, Clojure can leverage all the JVM ecosystem with host interoperability.

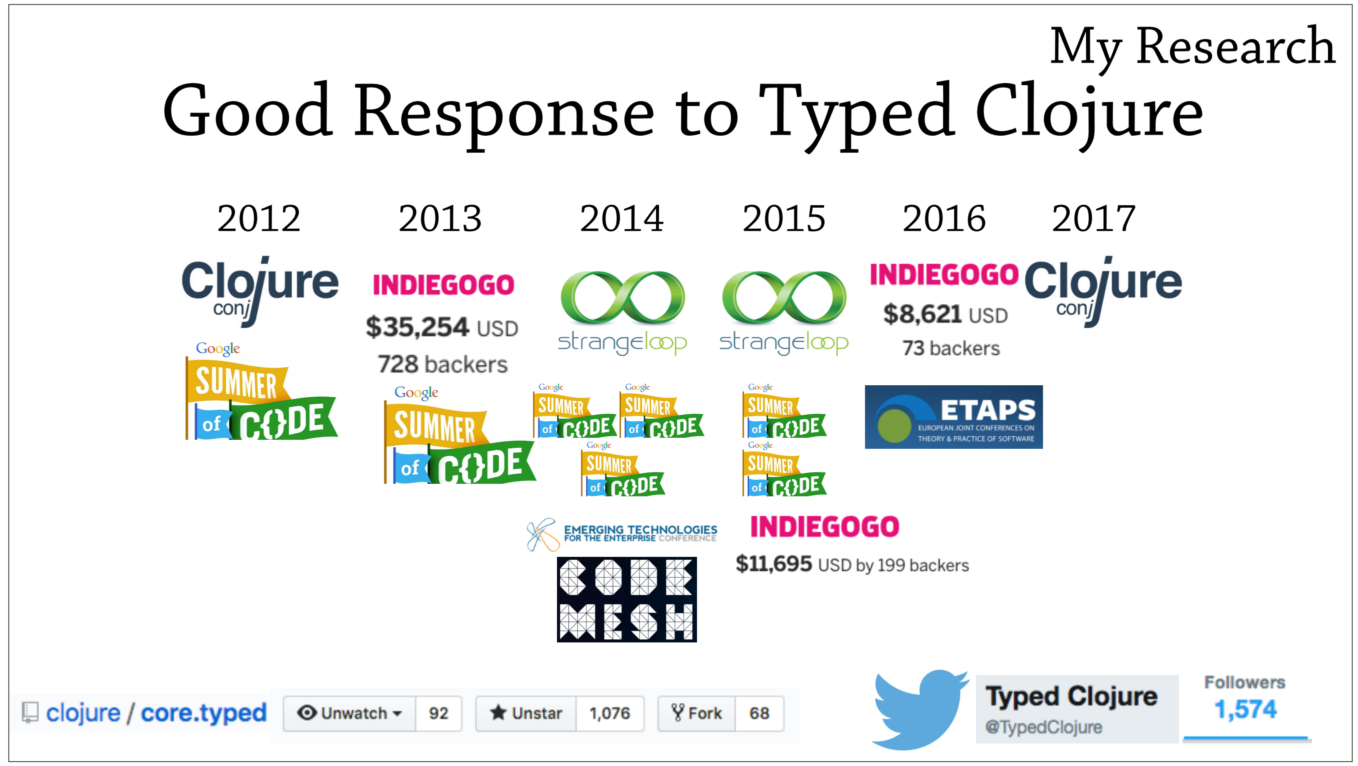


However, Clojure programmers have their frustrations with Clojure. Of the technical complaints, my take is that Clojure programmers need help specifying and verifying their programs. The number 2 complaint was the quality of error messages, with suggestions of creating better language tools perhaps via static typing.

Typed Clojure

Typed Clojure is an *optional type system* for Clojure

This leads to my work. I create Typed Clojure, an optional type system for Clojure.



There has been a good response to Typed Clojure since I started it in 2012. I have spoken a several major industry conferences, raised money to fund its development, and mentored students through GSoC.

How Typed Clojure works

Here's how TC works.

How Typed Clojure works

1. Take an existing Clojure program

```
(defn say-hello [to]
  (str ‘‘Hello, ’’ to))

(say-hello ‘‘world!’’)
;=> ‘‘Hello, world!’’
```

First, you take and existing Clojure program. This particular one creates a Hello World string.

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!’’
```

Then you add type annotations to each top-level function.

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!’’
```

This says “say-hello” accepts any value and returns a string.

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!’’
```

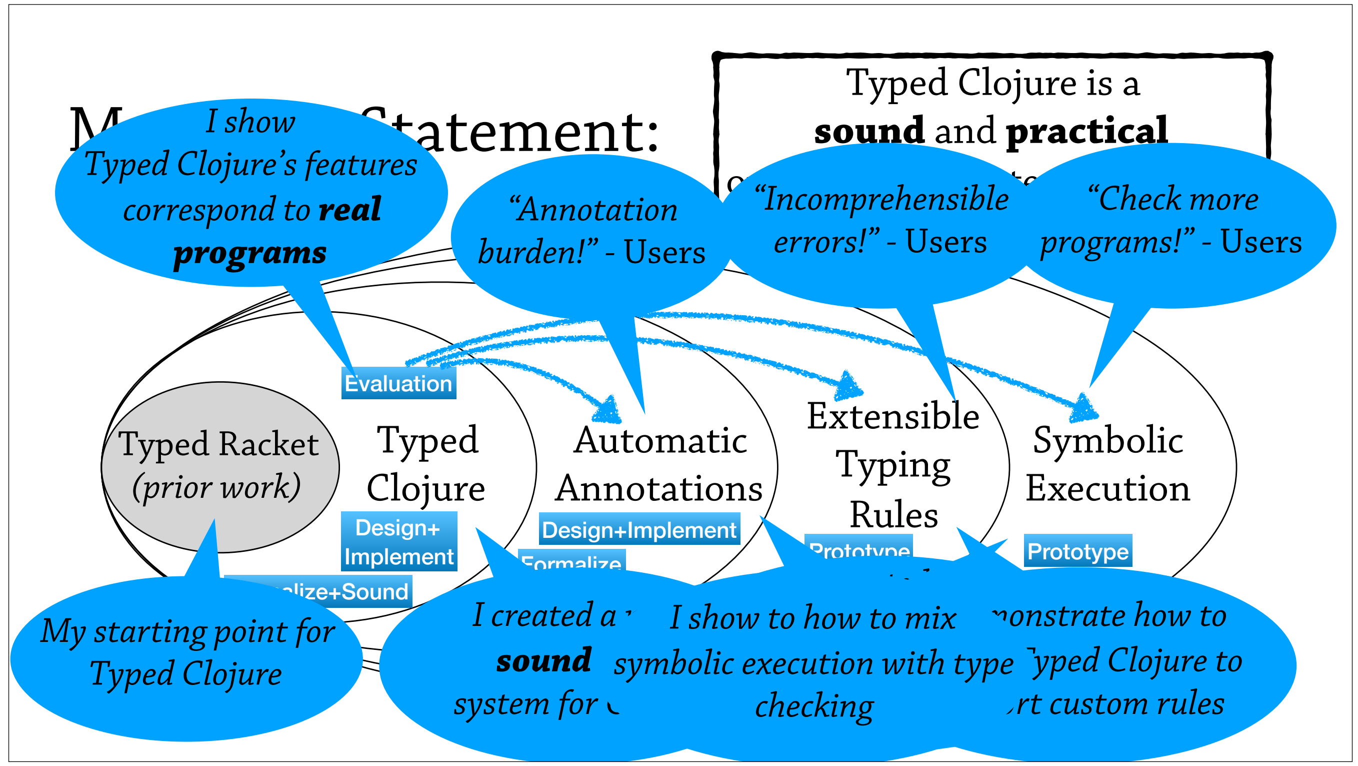
Finally, you use the provided type checker to verify the Clojure program conforms to the type.

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
```

This happens a compile-time, so this is a static analysis. The return type of String is calculated without running the program.



Today I am here to present my thesis on TC, summarized by my thesis statement: "TC is a sound and practical optional type system for Clojure". First, I identified TR as a good starting point for a Clojure type system, and repurposed its ideas and implementation. I present the design of TC, formalize its core and prove it sound.

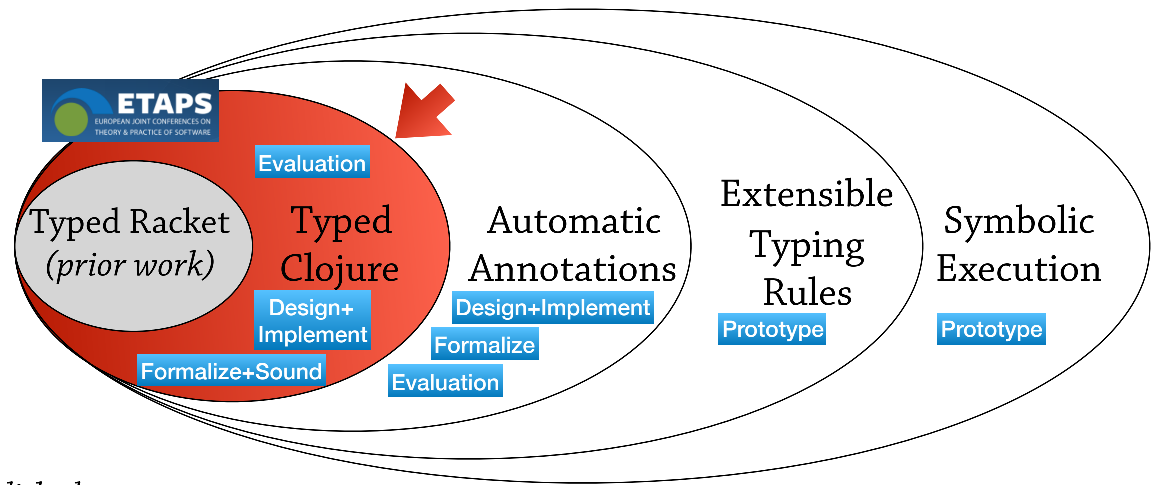
Then I show TC's features correspond to real-world programs by evaluating over 19k LOC in a production installation of TC.

This evaluation revealed several shortcomings. First, users encountered a high annotation burden, which I created a tool and workflow to help users write annotations. Second, type errors in expanded macros were difficult to understand, so I demonstrate how to extend Typed Clojure with custom typing rules for macros. Third, I show how to mix symbolic execution with type checking to type check more programs.

Part I

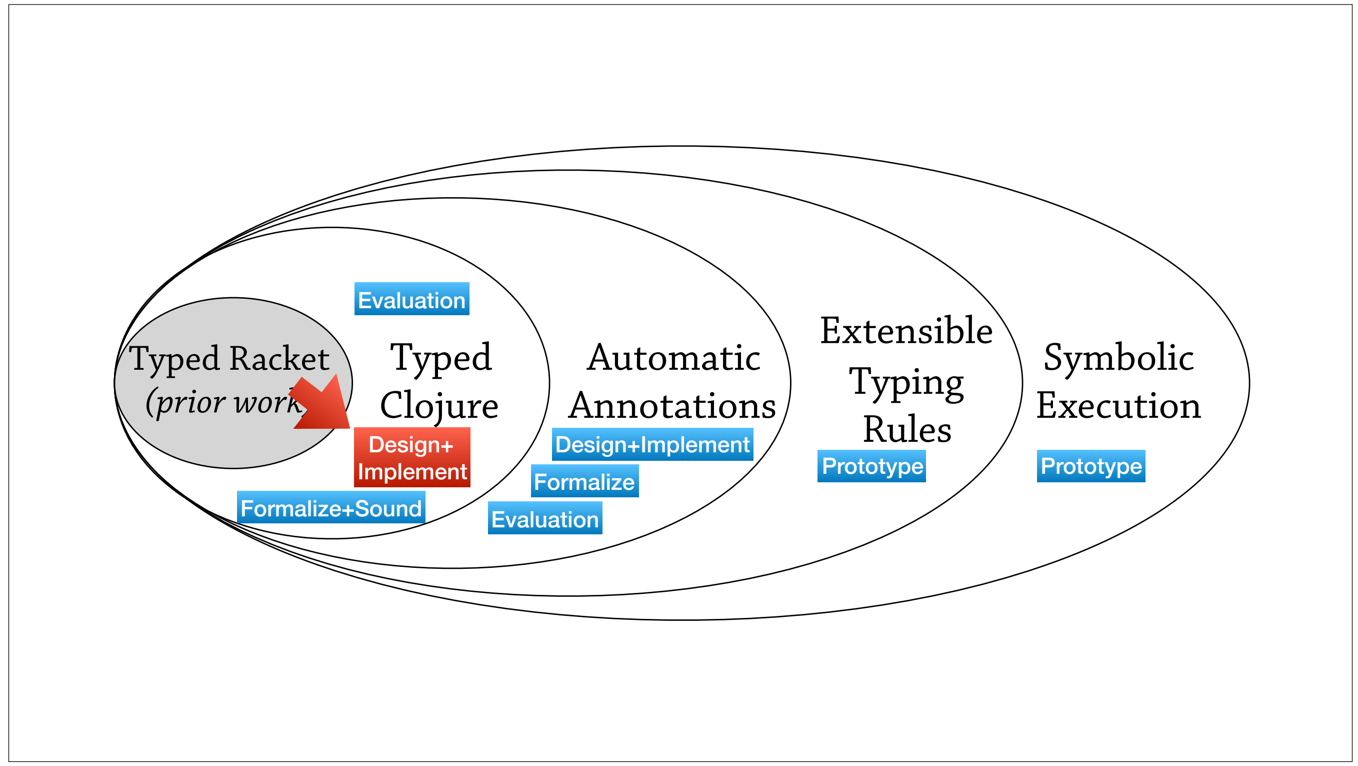
Design and Evaluation of Typed Clojure

The first part of this talk concerns the initial design of Typed Clojure.



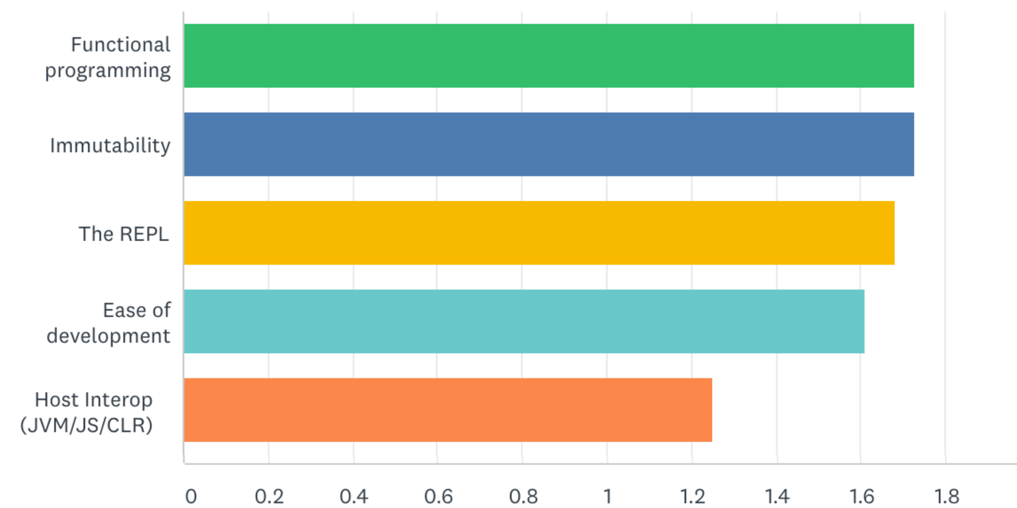
Published:
“Practical Optional Types for Clojure”, **Ambrose Bonnaire-Sergeant**, Rowan Davies, Sam Tobin-Hochstadt; **ESOP 2016**

This part was published in ESOP 2016.



Now, an overview of the design and implementation of Typed Clojure.

Check with Typed Clojure



Let’s go back to these top-rated features of Clojure. I’m going to show you some Clojure programs that exhibit these features, explain how they work, and how to check them with TC.

Simple Functions

Scorecard

Functional programming ✔

Immutability ✔

The REPL

Ease of development

Host Interop

```clojure
(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
```

First, simple functions. Here, a function `point` is defined that takes a pair of coordinates and returns a record with two fields, x and y. On the last two lines, you can see how to lookup these fields. In fact, the curly brace syntax introduces a plain map, we are just using it heterogeneously. So to check this in TC, we add a type alias for this ad-hoc record, and annotate the function. This demonstrates support for FP and immutable data structures, since maps are immutable in Clojure.

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"
```

Next, here’s an example of a HOF which combines the coordinates of a point based on a function, first with plus, then with string concatenation. A polymorphic annotation is needed, that accepts a point and a 2-argument function. This demonstrates a hallmark of FP that strongly contributes to Clojure’s ease of development.

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]                    Str
  (if (string? m)
    (Integer/parseInt m)
    m))
Int

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

Next, an important idiom in Clojure is type-based control flow. Here, we choose branches based on the type of “m”. In the then branch, we use Java interop to convert strings to ints. A union type in the annotation is all we need to check this — occurrence typing automatically follows the control flow since local bindings are immutable.

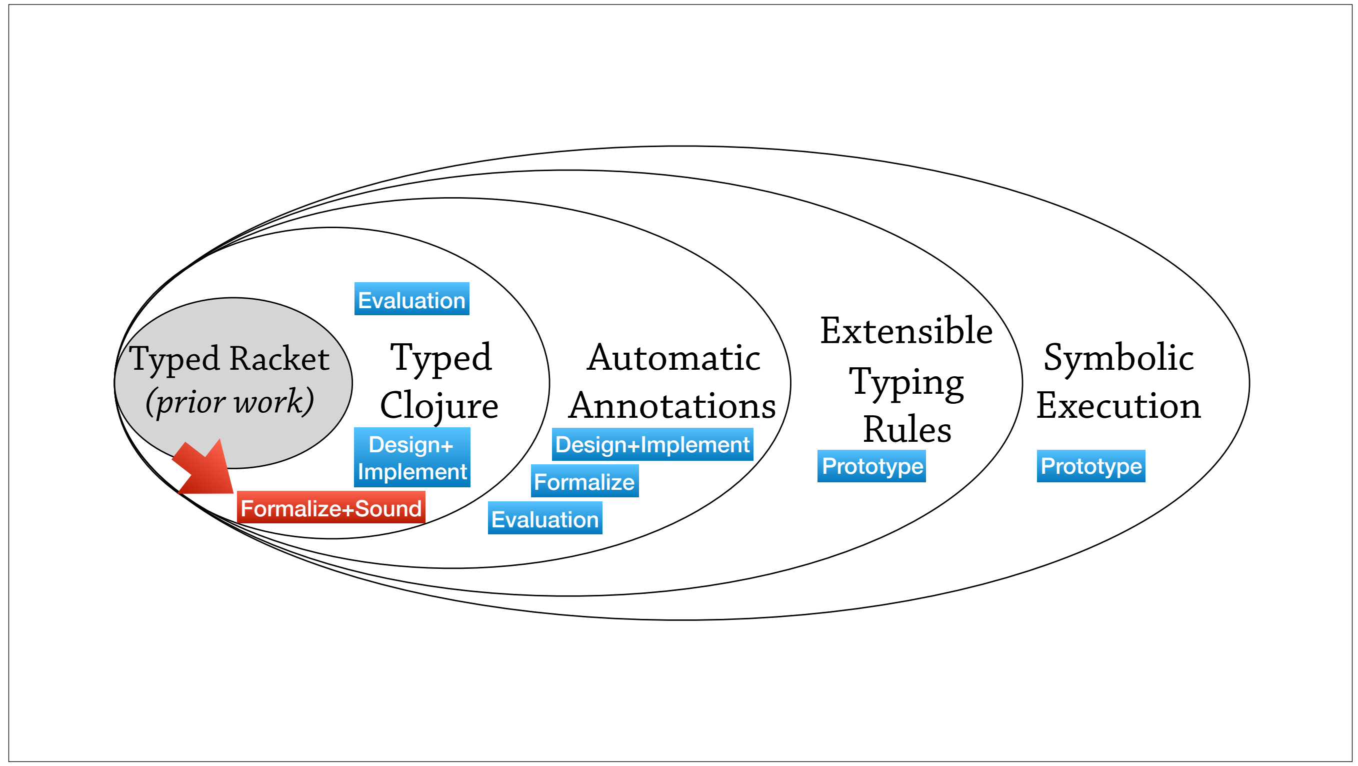
Multimethods

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
```

Here’s the same example, except implemented as an extensible multimethod. By dispatching on the class on an argument using “class” as a first-class function we can install methods for each case. The same annotation is enough to type check this multimethod. Again this shows host interop support in TC, but also first-class functions.



Now, we cover how I formalized and proved Typed Clojure sound.

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

The formalism is based on occurrence typing. I added the TC features heterogeneous maps and multimethods, and some Java interoperability.

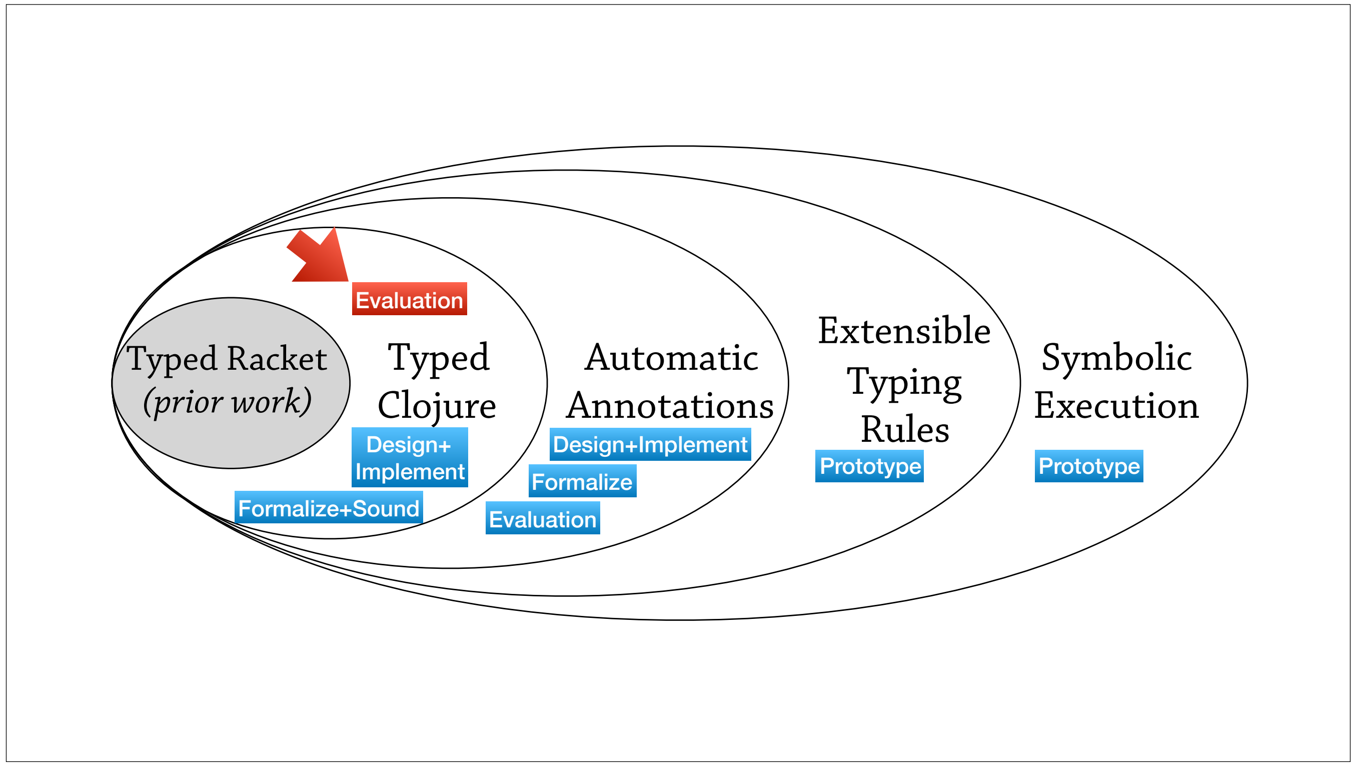
λ_{TC}

Type soundness

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

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

Then I proved type soundness for this fragment of TC, along with the theorem that "well-typed programs don't go wrong". Since I encoded NPE's as "wrong", we get the corollary that TC rules out NPE's. Null is idiomatic and common in Clojure, so this is an important result that distinguishes TC from other systems like Scala and Java.



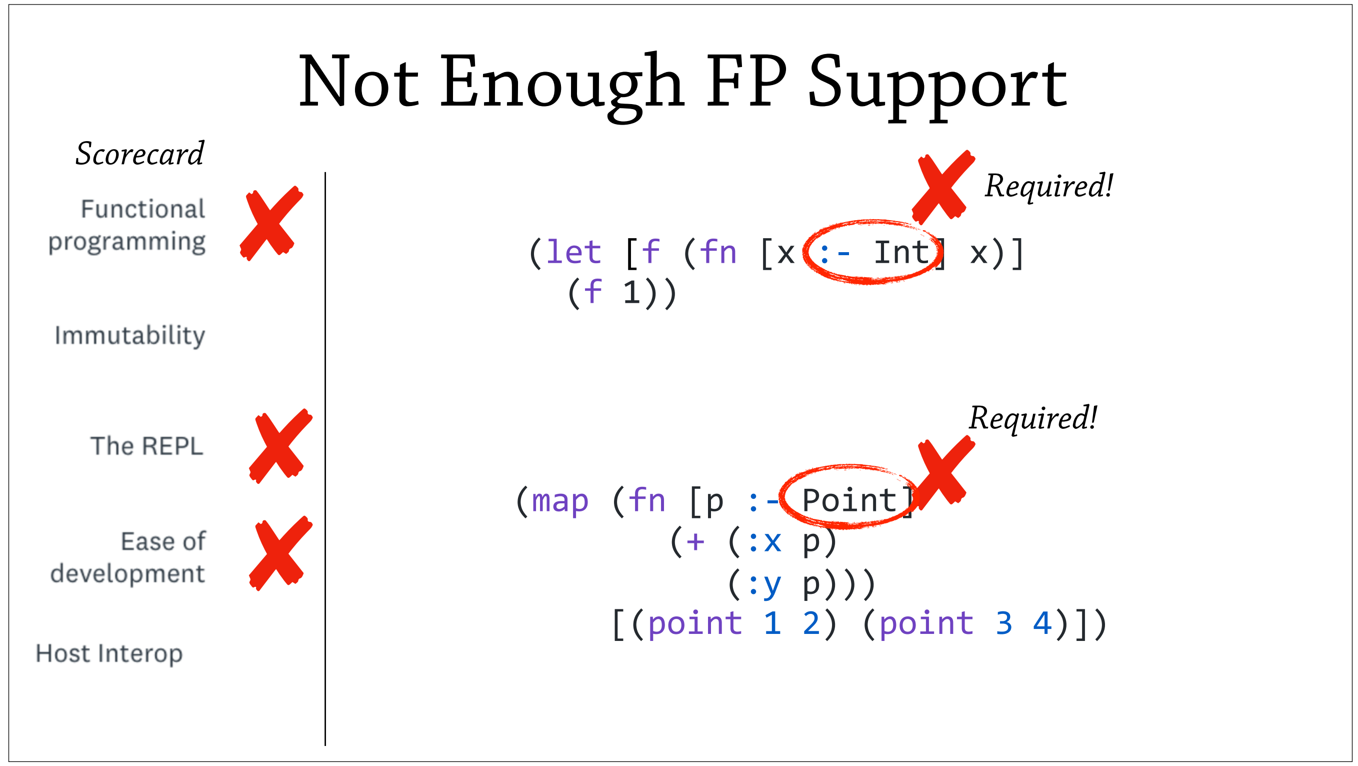
Next, I present my evaluation of the TC's initial design.

Empirical Evaluation of Typed Clojure

circleci

19k lines of Typed Clojure

I surveyed over 19k LOC in a production installation of TC at CircleCI, where it was used to type check their CI tool.



I already showed you the good news of what TC does well. Here's the bad news. Users were frustrated in the amount of local annotations needed. Every local function requires an annotation in practice. This meant the anonymous function sugar was essentially unsupported without an ugly inline annotation. This made Clojure feel less flexible and dynamic.

Global Annotation Burden

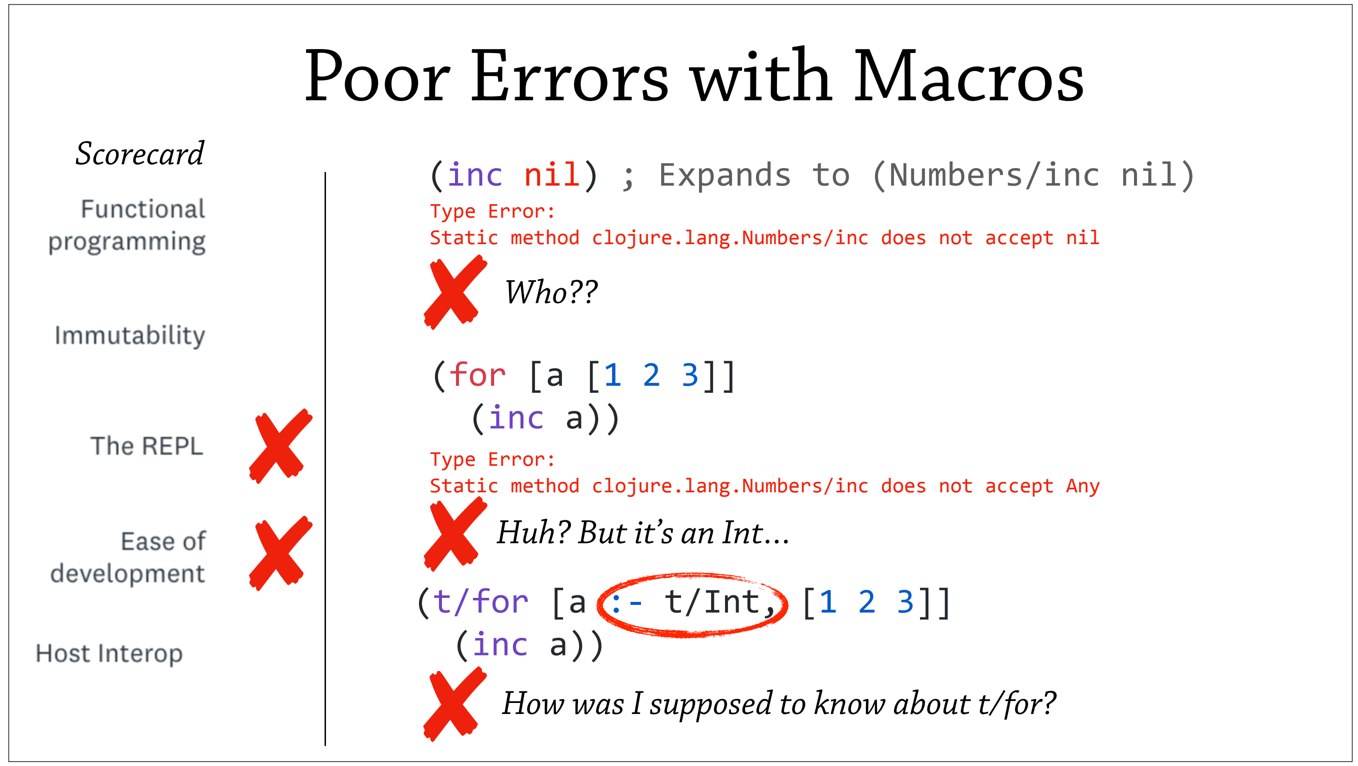
Scorecard

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

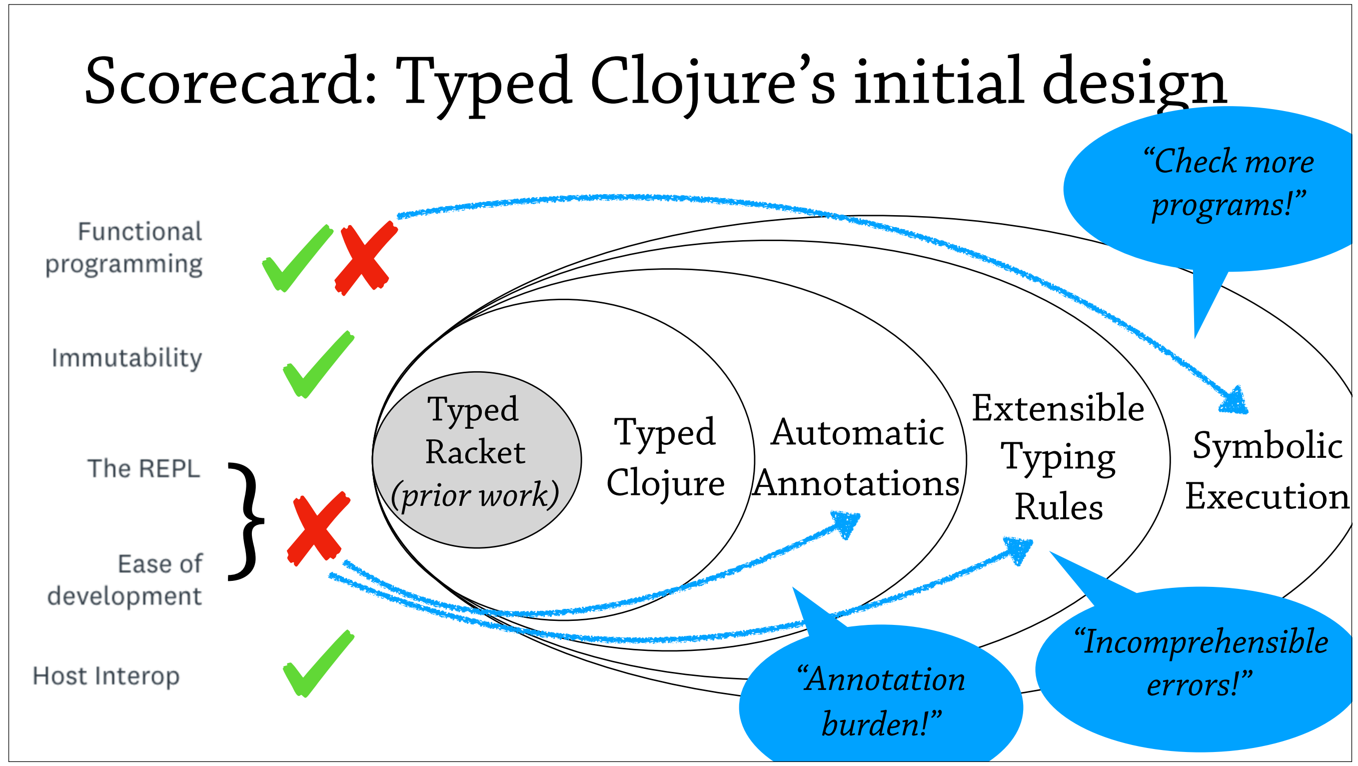
```
(defalias Point
  '{:x Int :y Int})

(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! ✗

Users felt the annotation burden was too high, since all top-level functions must be annotated. They also needed to reverse engineer libraries they used to derive annotations. This was very disruptive.



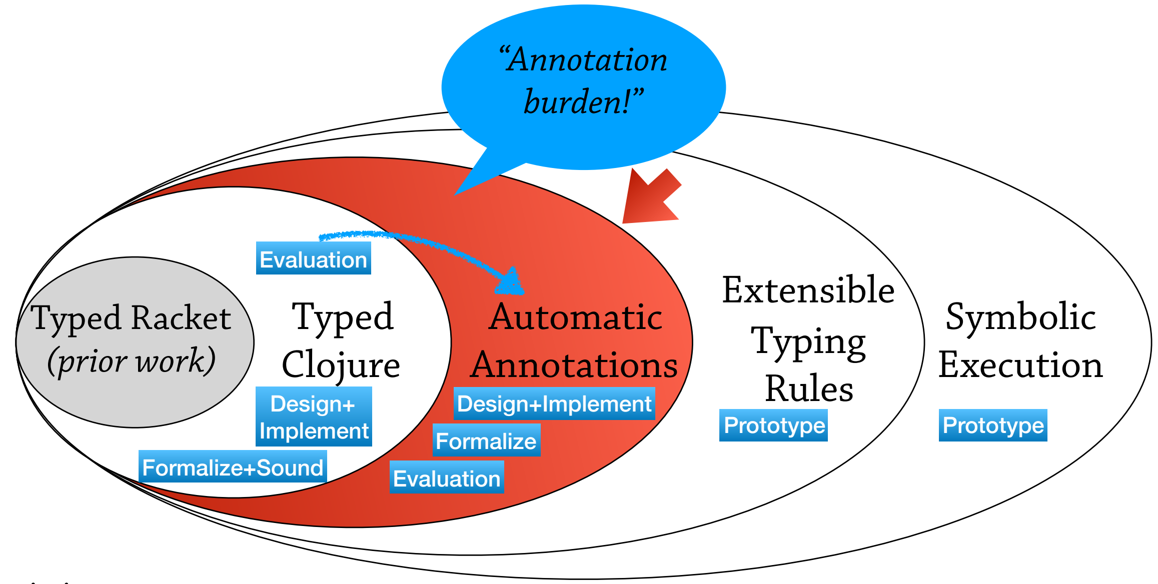
And finally there was a lot of confusion around TC's approach to checking macro usages. For example, the error message for (inc nil) refers to its inlining. More complicated macros like the "for" list-comprehension could not infer good enough types, and users had to use "wrapper macros" (if they knew how, the error didn't say).



So how did TC’s initial design do overall? Well, it’s good a functional programming, immutability, and host interop. But it has some limitations in checking FP idioms like requiring too many annotations, and there are various issues that make Clojure development less enjoyable. I address these issues in three parts. First I help users write annotations. Second I build a system to extend TC with typing rules. Third, I use symbolic execution to check more programs.

Part II
Automatic Annotations

Now, we cover automatic annotations for Typed Clojure.



In submission:
“Squash the work: A Workflow for Typing Untyped Programs that use Ad-Hoc Data Structures”,
Ambrose Bonnaire-Sergeant, Sam Tobin-Hochstadt

This work is currently in submission, and is the first response to the evaluation.

Annotation burden

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

(ann point [Int Int -> Point])
```

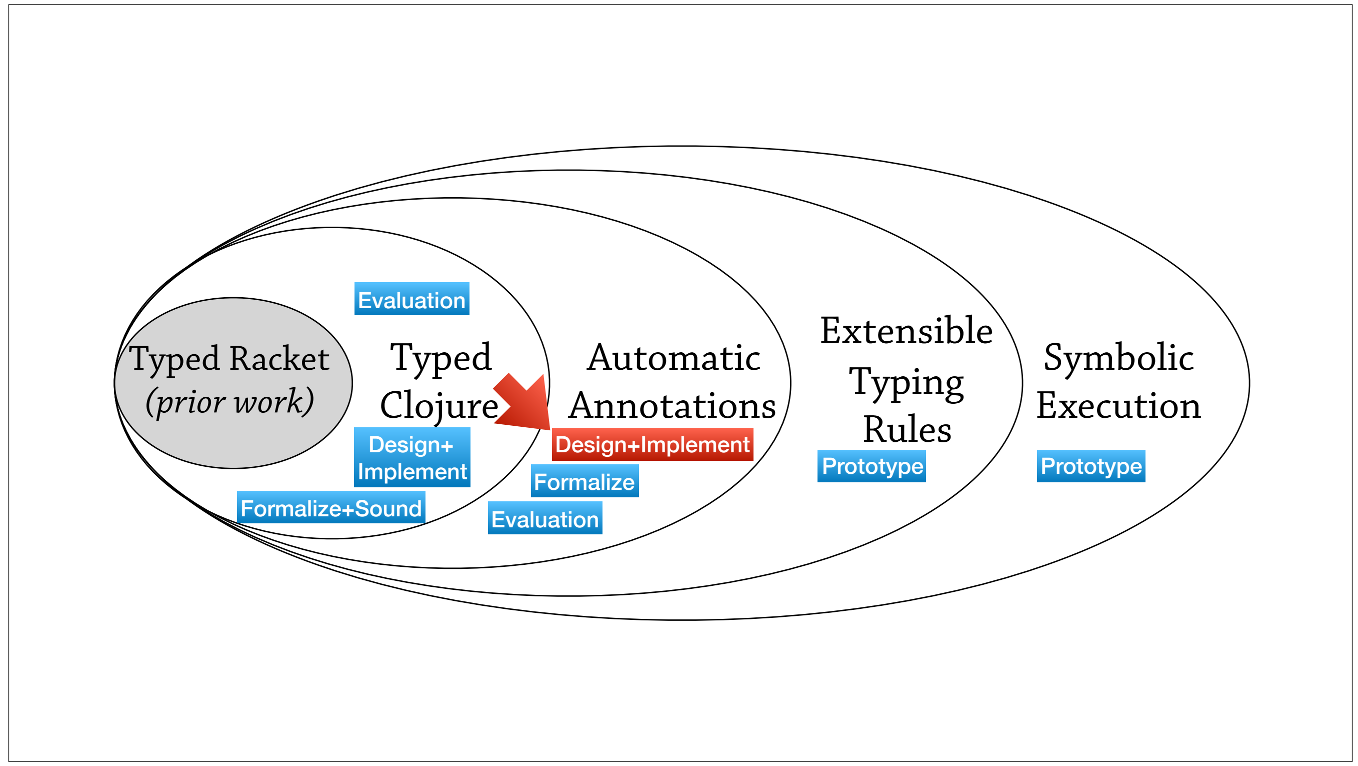
```
(ann combine
  (All [a]
    [Point [Int Int -> a] -> a]))
```

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

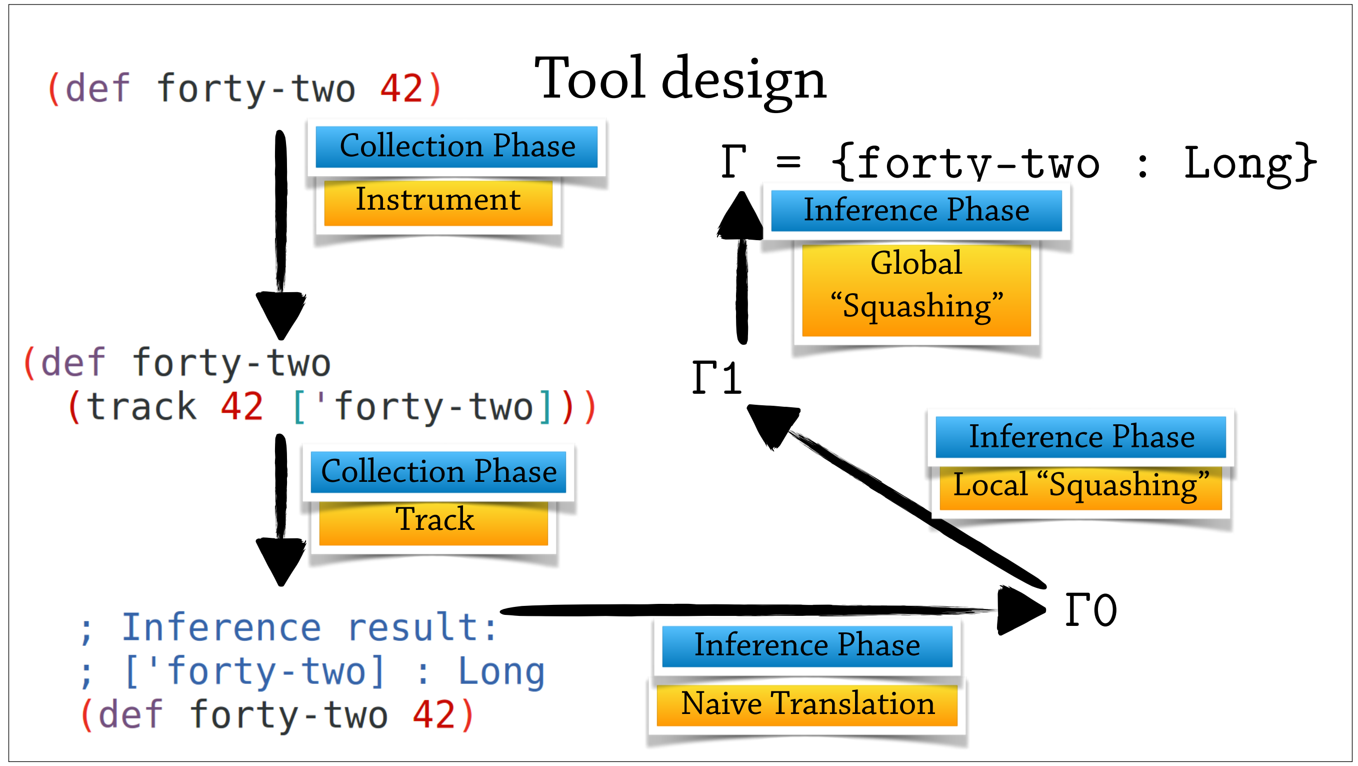
```
(ann extract-int-mm
  ['{:value (U Int Str)} -> Int])
```

Goal: Automatically generate

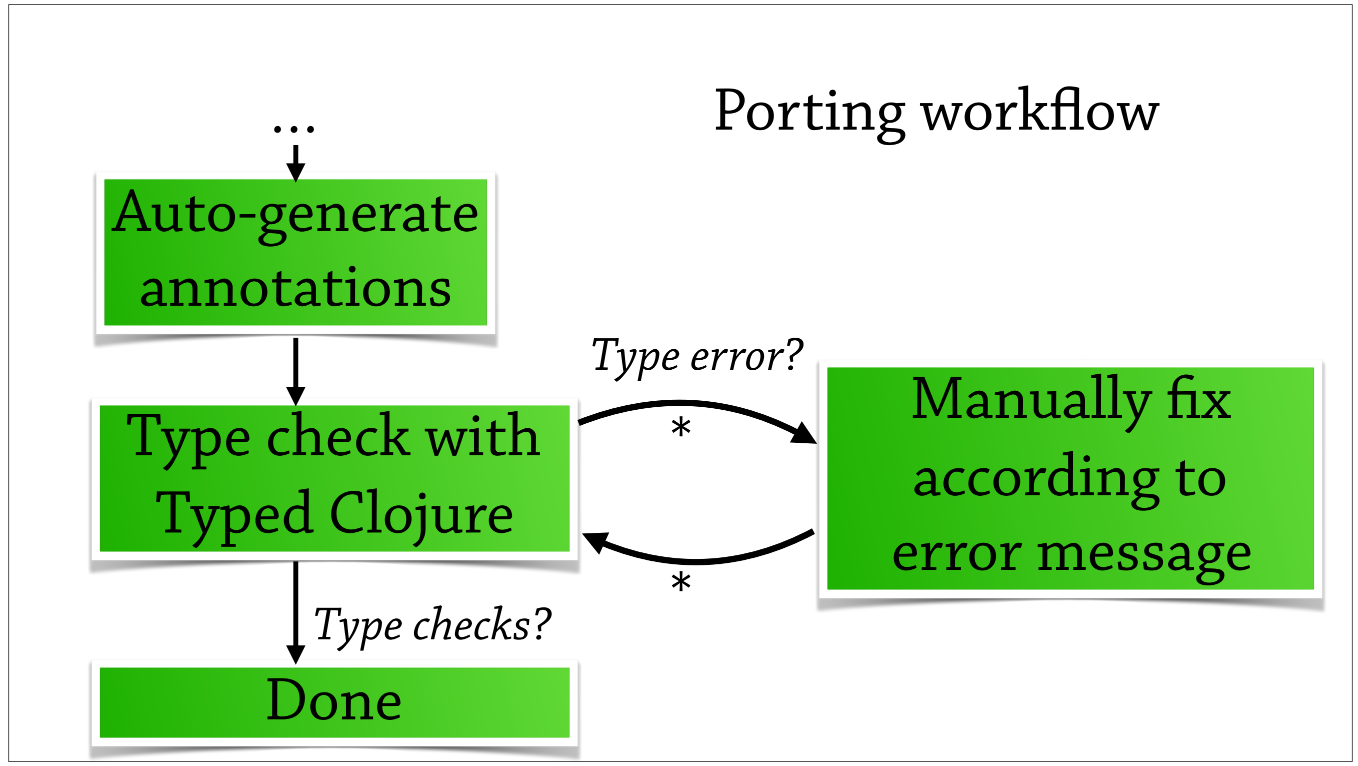
So the overall goal of this work is to automatically generate top-level annotations so users don't have to write them (in full).



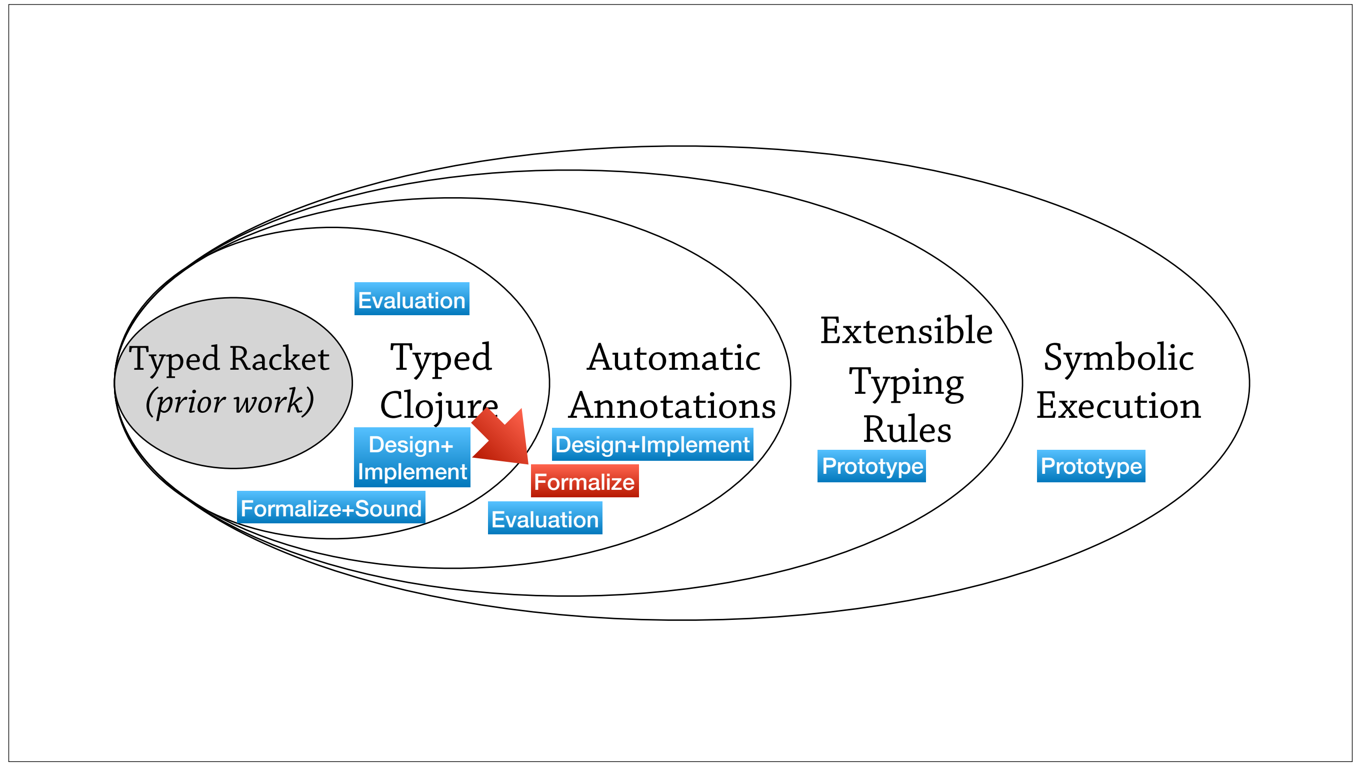
First, we cover the design and implementation of my tool that achieves this.



The tool is based on dynamic analysis, so it observes your running program. It's split into two phases. First the collection phase collects runtime samples, then the inference phase translates samples into an annotation. For example, to annotate this program, it is first instrumented, then tracked, and several passes are used to make compact annotations. Local squashing creates recursive types from directly nested types. Global squashing combines types from different functions.



This tool is the first part of a porting workflow. First, you run the tool to generate types. Then you type check the result in TC and keep fixing type errors until it checks. The idea is that TC is a sound system so this way you get meaningful specifications in the end.



Now the formalism for our annotation tool.

λ_{track}

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

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

“Track and annotate x’s in program e”

The main driver is this “annotate” function that tracks definitions x’s in program e.

λtrack

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

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

$$\mathsf{annotate}(\underbrace{(f\ \ \{\mathsf{:a}\ 42\})}_{\text{Test}}, \underbrace{[f]}_{\text{Track-me}}) = \underbrace{\{f : [\{\mathsf{:a}\ \mathsf{N}\} \to \mathsf{N}]\}}_{\text{Derived type}}$$

For example, we have a definition “f” and a test. Plugging them into annotate gives the desired type environment.

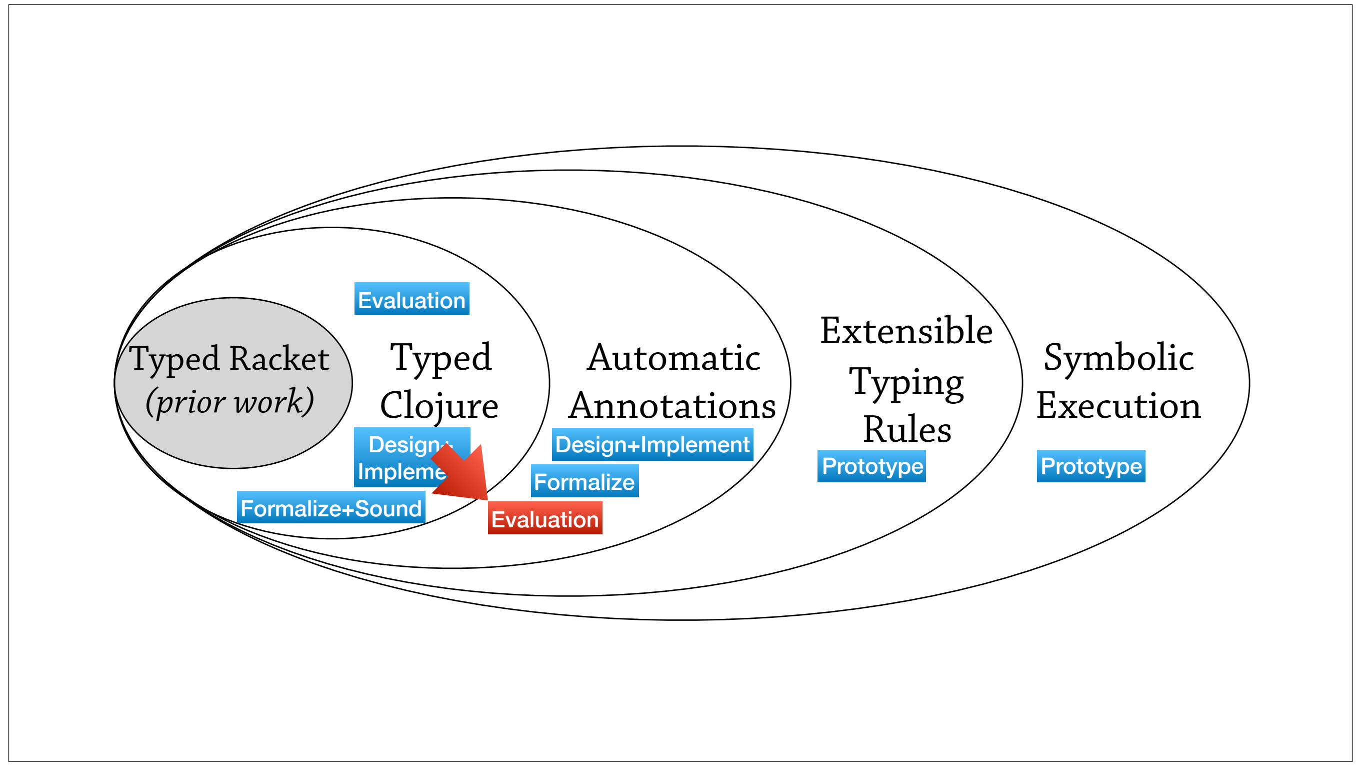
λtrack

Intentionally unsound

Aggressively combines
types to create compact aliases
and recursive types

Tailored for the workflow

This model is intentionally unsound because it aggressively creates recursive types from unrolled examples. These’s not much to prove about it, so let’s move on…



… to the evaluation of the porting workflow.

Evaluation

Ported 5 open-source programs (~1500 LOC)

Measured the kinds of manual changes needed

I ported 5 open source programs from Clojure to TC and measured the kinds of manual changes needed.

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

For example, the annotation for “mult” is actually supposed to accept any number of integers. I had to manually change a type based on a type error (this was because mult was only exercised with 2 arguments). Similarly, the next function takes a map of ints to ints, but actually the map may contain any keys. The fix is to manually “upcast” the 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?}
    ...         ...
    ...         ...))
```

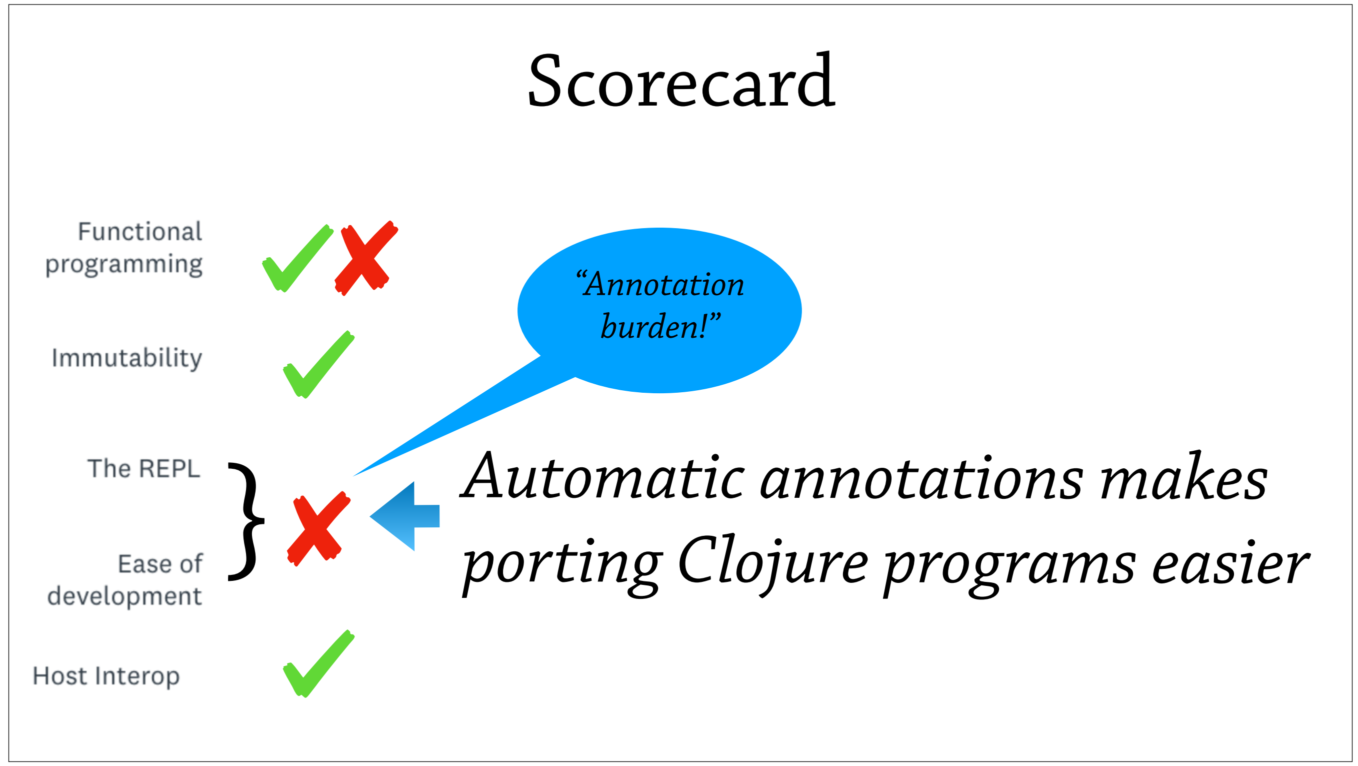
Our tool can also generate recursive types. Here's the function we're generating types for. This function creates an AST from Clojure data, and the automatically generated type is recursive and shared amongst several function annotations. However, it's missing cases due to spotty tests, and I manually had to add some cases (but only in one place).

Manual effort

Mostly deleting/upcasting types

Adding missing cases to (generated) recursive types

I found most of the effort was deleting or upcasting generated types, and adding missing cases to recursive types.

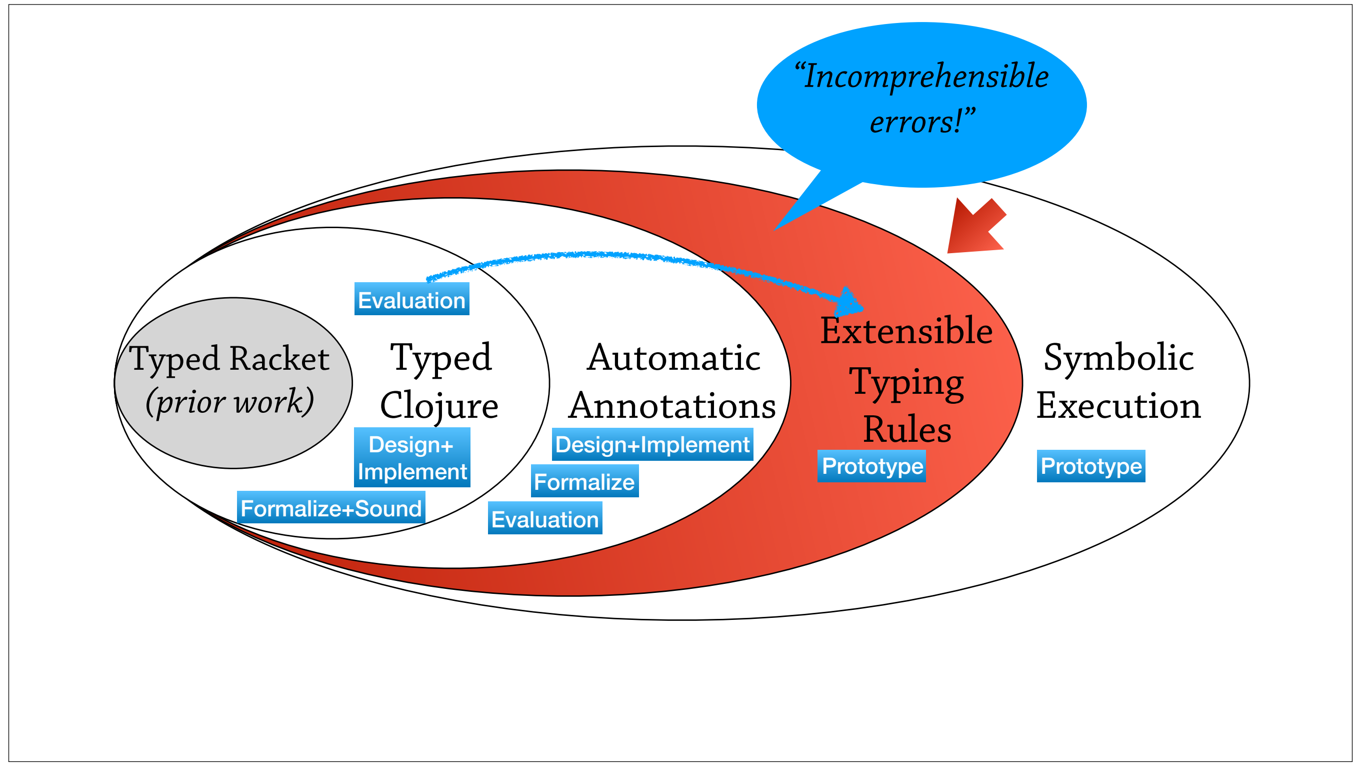


Based on this experience, this porting workflow makes porting Clojure programs easier, and addresses a key concern in our evaluation of TC.

Part III

Extensible Typing Rules

Next, we look at how I address poor type error messages due to macroexpansion.



This is the second response to my evaluation.

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
```

Here's a recap of the problem. If a type error happens in a macro-expansion, it's difficult for the user to tell why it happened. One way to prevent this is to propagate type information so then less type errors happen in the first place.

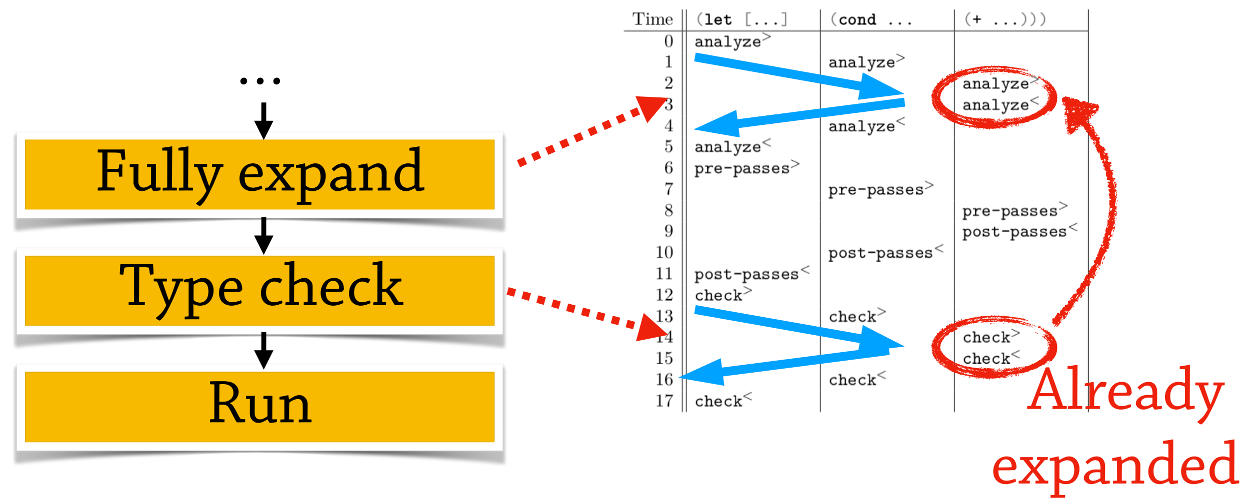
Idea

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

Allow the user to define custom typing rules for macros

The way to achieve this is to define custom typing rules for macros.

Roadblock:
Expansion comes *before* check

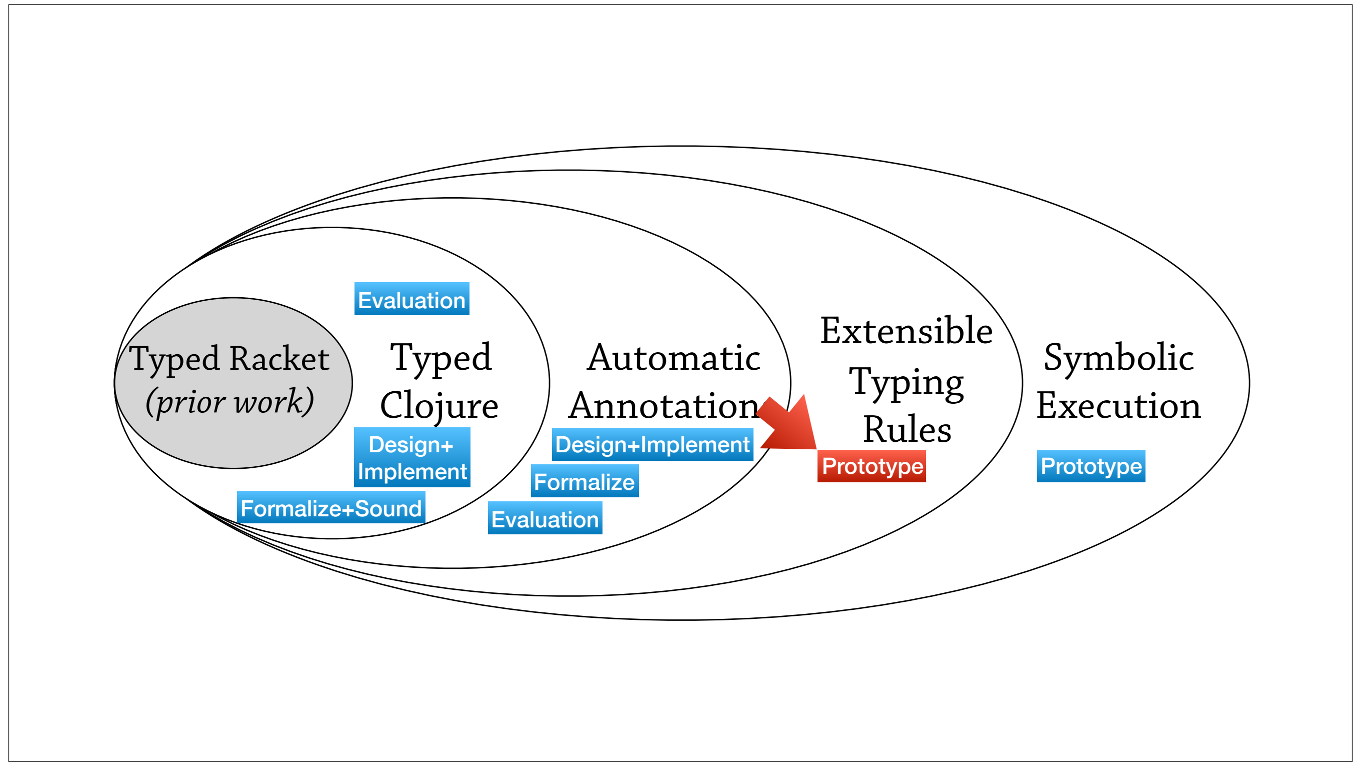


However, there's a problem. Typed Clojure fully expands code before it type checks. If we view expansion and checking as several passes over the same expression, then by the time the checker finds the expression, is has already been expanded. This is inherited from Typed Racket's design.

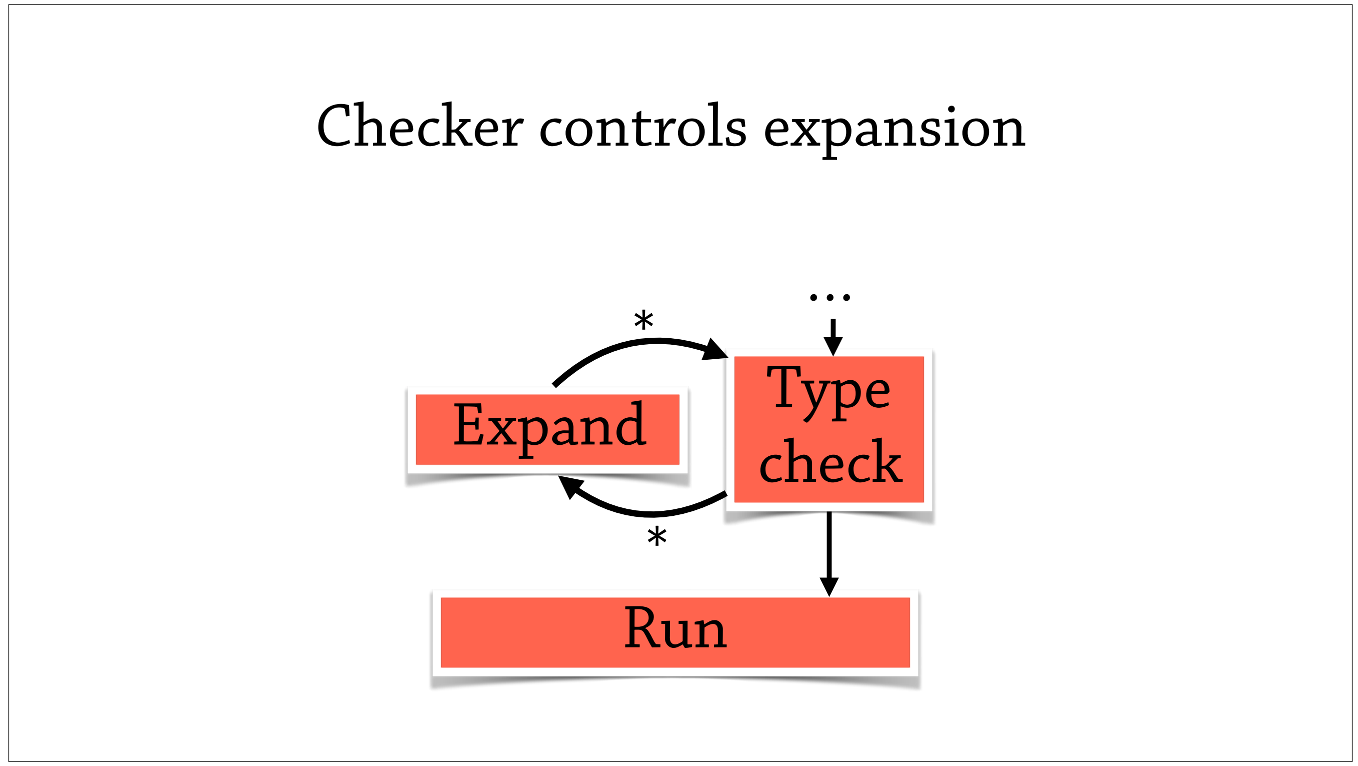
Solution

Allow Typed Clojure to interleave macroexpansion and type checking

The way I address this is to allow TC to interleave macroexpansion and type checking.



Now I present the prototype that demonstrates this idea.



Interleaving macroexpansion and type checking, essentially gives the type checker the control over expansion. Once the checker finds an expression it knows how to check, it can fire a rule to check it.

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

To achieve this, I wrote a new code analyzer that gives the checker the ability to expand expressions as needed.

This was non-trivial

Must also interleave *evaluation*

Maintains correct lexical scope

Interacts with Clojure's type hinting system

This was not easy for many reasons, here are 3. First, Clojure's evaluation model already interleaves macroexpansion and evaluation, and it was not obvious how to integrate TC's checker into that scheme. Second, it was imperative to maintain correct lexical scope while incrementally expanding code, which does not come for free in Clojure. Third, Clojure already has a "type hinting" system that must also be accounted for.

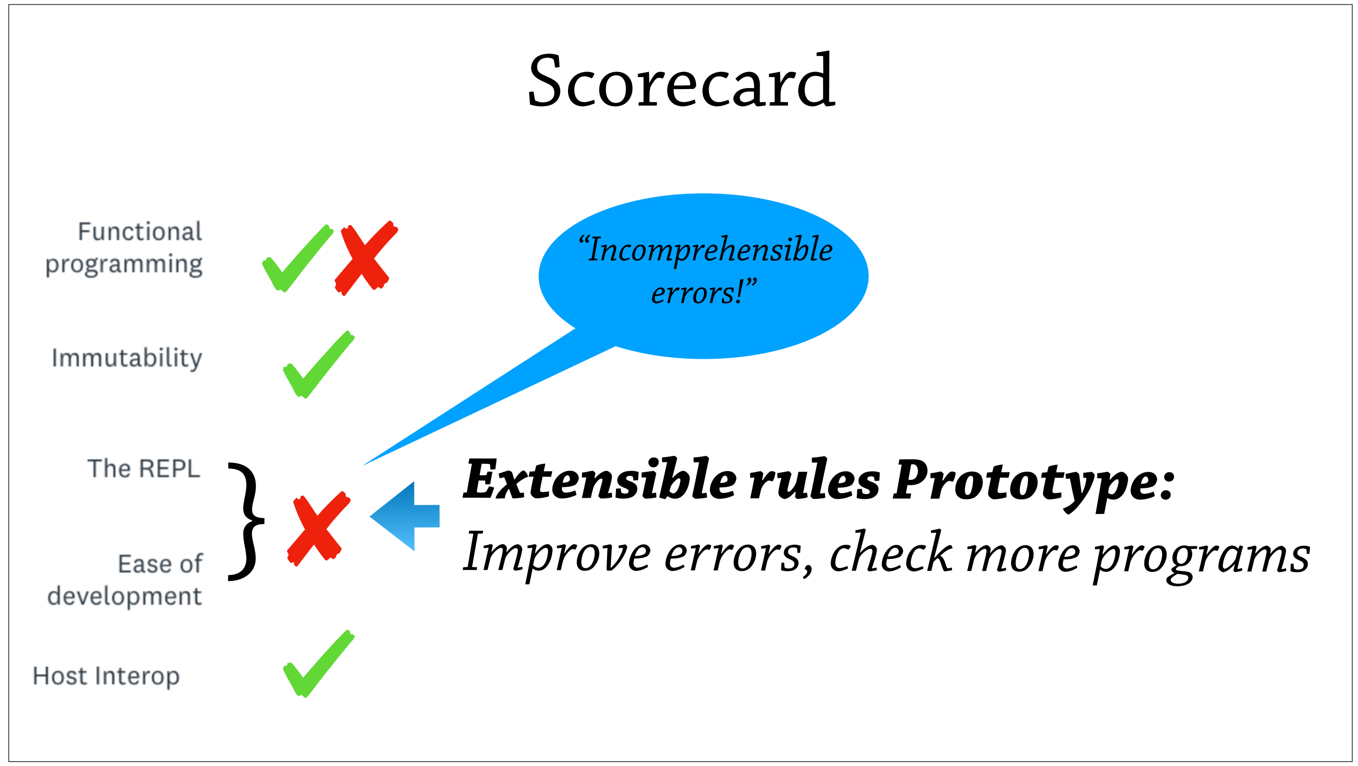
Example type checker
with new analyzer

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))))
```

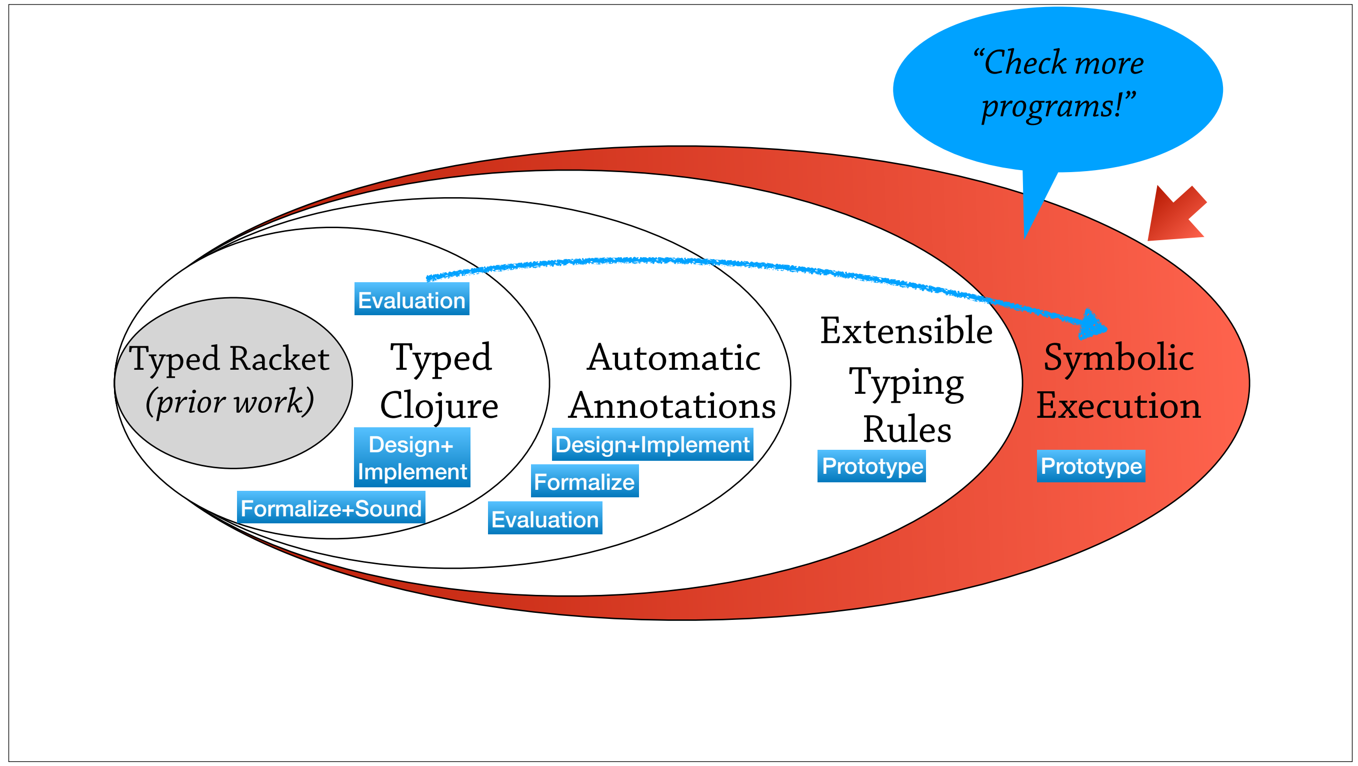
But, once this analyzer was built, we can build type checkers that interleave expansion and checking. Here’s an example, where the type checker asks if an expression is partially expanded and then rules custom rules based on that.



This prototype demonstrates how to improve type error messages involving macros, and check more programs, which should improve the experience of TC users.

Part VI
Symbolic Execution

Now we discuss adding symbolic execution to TC.



This is the final response to the shortcomings identified in my evaluation.

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)])
```

The goal is to reduce local annotations.

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

The reason these annotations are needed is because type checking proceeds outside-in. Types for parameters are needed when a function is discovered by the checker.

Intuition

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

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

The intuition behind my solution is to notice that useful type information is available adjacent to these functions. If only we could delay the checking of these functions until those points.

Approach

New type rule for checking (unannotated) functions:

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

The type of a function is its <u>code</u>

…and the <u>type environment</u> it was "defined" at

The way this is achieved is by adding a new type rule for checking unannotated functions. The type of these functions is its code, coupled with the type environment it was defined with.

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*

This approach resembles runtime closures, except they are executed symbolically, so we call this a symbolic closure type. They are similar to “abstract closures” in control flow analysis.

Approach

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

Application rule?

What about an application rule? The idea is that all the information to check a symbolic closure is maintained in the symbolic closure itself, and only the argument type is needed. So, we rearrange the various pieces to derive the output type.

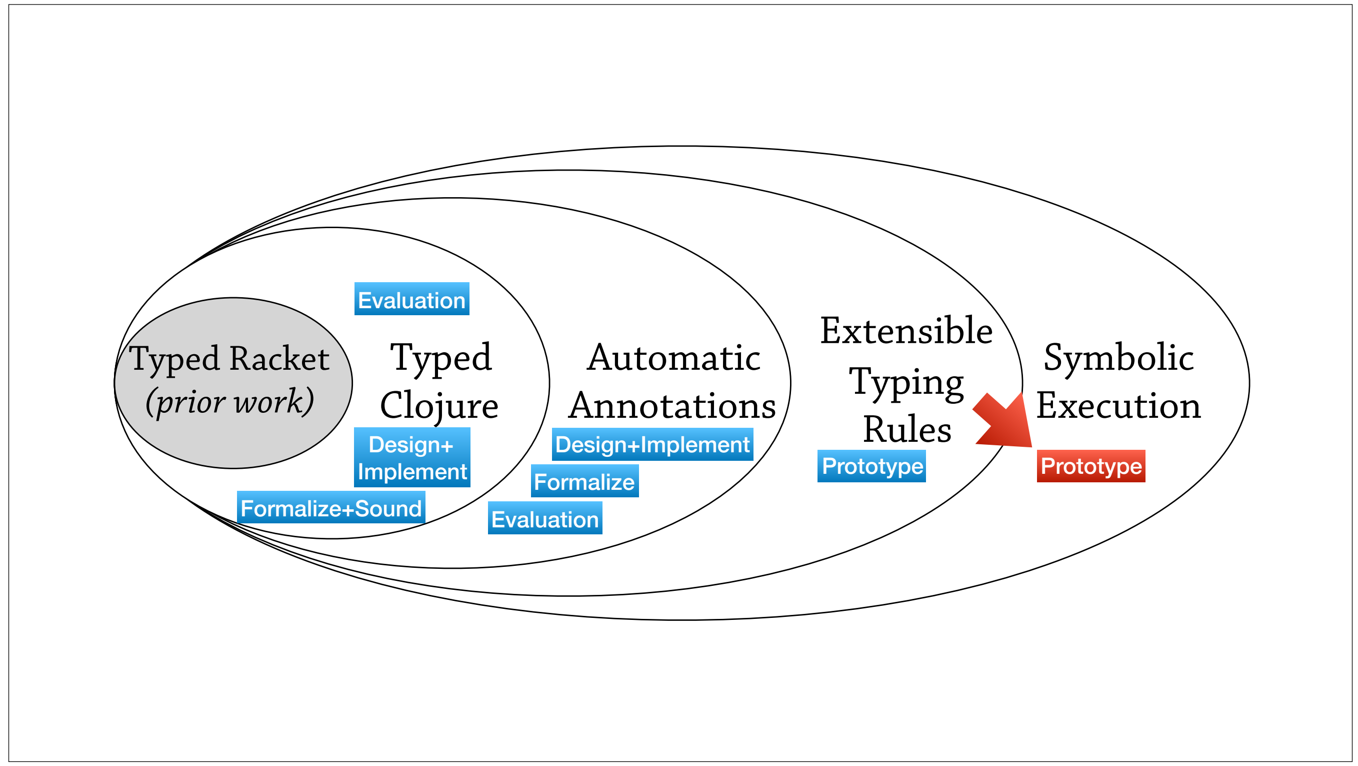
Tradeoffs

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

There are important tradeoffs involved here. First, symbolic closures are undecidable in general. However, they are viable because many local functions in Clojure are small and non-recursive, so they are cheap to symbolically analyze. We can then rely on the (mandatory) top-level annotations to drive the symbolic execution.



Now we cover my prototype that combines type checking and symbolic closures.

Naive formalism

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

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

The typing rules associated with symbolic closures strongly resemble the big-step reduction rules for runtime closures. The introduction rule for symbolic closures just packages the code with its definition environment. The application rule unpacks the pieces, extends the parameter to be the derived type, and the body is checked to give the type of the entire application.

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
```

I also provide a prototype implementation for experimentation. The tc operator takes an expected type and an expression, and return the type of the expression. Integers can synthesize their type. Providing an expected type to a function triggers the usual bidirectional propagation. Omitting a type gives a symbolic closure. Applying a symbolic closure uses symbolic execution to derive the result type.

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)
```

The prototype is also extended to work with polymorphic types. By inspecting the type of “map”, the prototype knows how to feed type information to its function arguments. Similarly, this works in the presence of function composition.

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

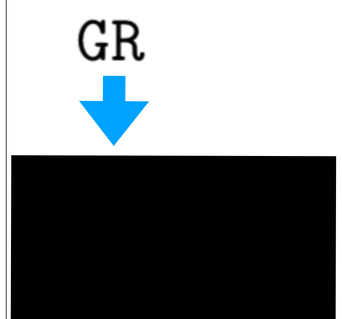
```
(ann id (All [a] [a -> a]))
(defn id [x]
  (GR (fn [_] (fn [_] x))))
```

[1] LICS'88, Giannini & Rocca

To test the limits of the prototype, I used this GR term, which is a strongly normalizing term that is untypable in System F (and probably Typed Clojure). This result was proven by Giannini and Rocca. However, evaluating it in plain Clojure, it's clear it's "morally" well-typed as an identity function. So, can we check this quirky identity function?

Prototype Implementation

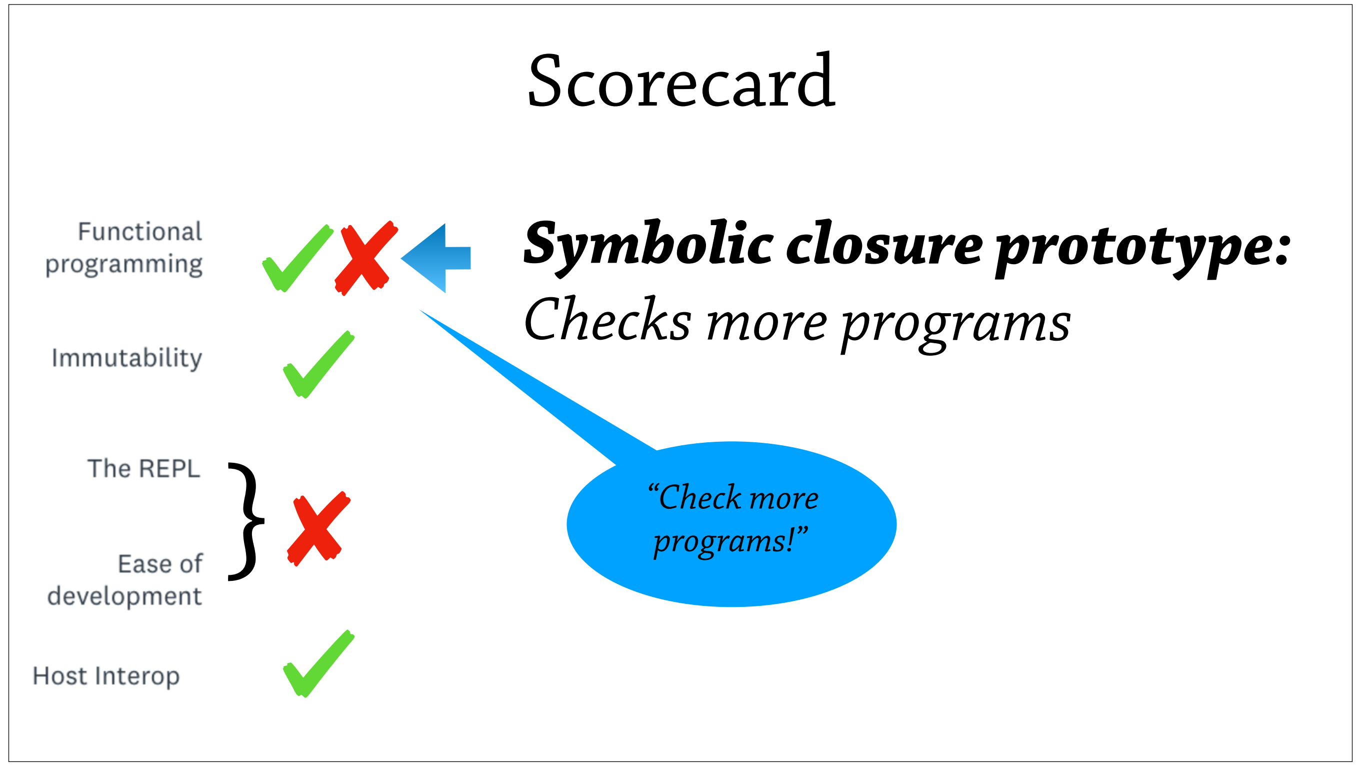
Symbolic closures let us treat GR as a **black box**
until it is executed symbolically




```
(tc (All [a] [a -> a])
    (fn [x]
      (GR (fn [_] (fn [_] x)))))
=> (All [a] [a -> a])
```

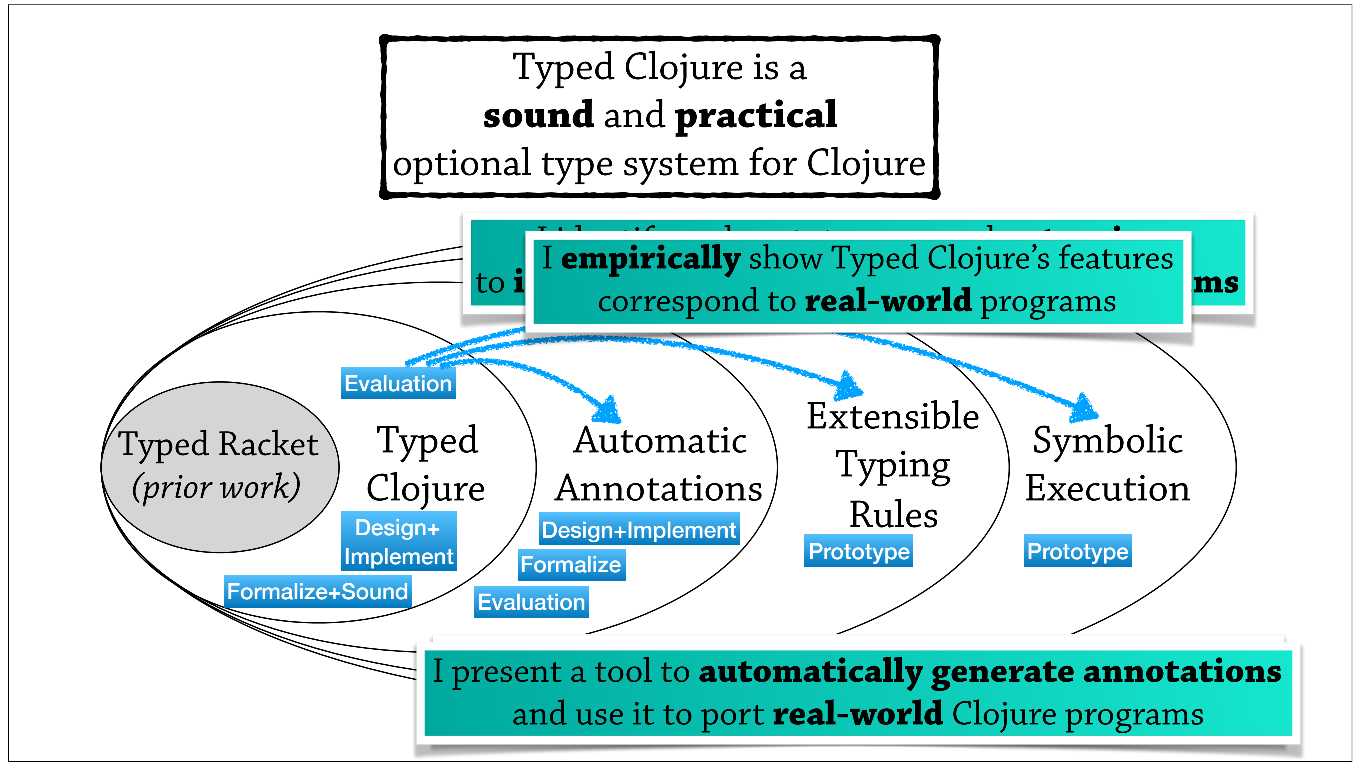
Symbolic Closures make the most
of top-level annotations

Yes, symbolic closures allow us to treat GR as a black box until enough type information is available to symbolically reduce it. First, x is a given type a. Then symbolic closures are symbolically executed until x pops out at the correct type. This shows how symbolic closures can check even hopelessly difficult-to-check expressions to traditional techniques.



So, based on this experience with symbolic closures, I claim that it is powerful enough to solve many of the type inference problems in TC.

Conclusion



To conclude, my thesis argues that Typed Clojure is a sound and practical optional type system for Clojure. I present the design of TC and prove it sound. I empirically show TC's features correspond to real-world programs. I present a tool to automatically generate annotations and port it to real-world programs. And I show how to extend TC with custom typing rules and symbolic execution to address user-experience shortcomings.

Thanks

Thanks for your attention.

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**