Thesis Proposal: Typed Clojure in Theory and Practice

Ambrose Bonnaire-Sergeant

August 13, 2018

Abstract

We present Typed Clojure, an optional type system for the Clojure programming language. This thesis argues Typed Clojure is sound and practical.

First, I will present Typed Clojure, an optional type system for Clojure. I will develop a formal model of Typed Clojure that includes key features like hash-maps, multimethods, Java interoperability, and occurrence typing, and prove the model type sound. Then, I will demonstrate that Typed Clojure's design is useful and corresponds to actual usage patterns with an empirical study of real-world Typed Clojure usage in over 19,000 lines of code.

Second, we address a major usability flaw in Typed Clojure: users must *manually* write annotations. To remedy this, I will present a tool that automatically generates Typed Clojure annotations based on observed program behavior, including a formal model of the tool, consisting of its runtime instrumentation phase that collects samples from a running program, and type reconstruction phase that creates useful annotations from these samples. Then, I will give an overview of a practical implementation that generates Typed Clojure annotations for real programs. Next, I will study the effectiveness, accuracy, and usability of these annotations by generating annotations for several projects, and then manually amending the annotations until they type check.

The final part of this thesis will either:

- increase the number of type checkable Clojure programs, especially those combining polymorphic higher-order and anonymous functions, by combining an extensible typing rule system with symbolic execution, and study its effectiveness in reducing the changes needed to port Clojure programs to Typed Clojure, or
- repurpose the automatic annotation tool to generate clojure.spec annotations, study its effectiveness in generating good specs over several hundred open source projects, and use it to help answer more general questions about Clojure usage.

1 Introduction

To contextualize the thread of work I propose in this thesis, this section gives a general introduction to Clojure and Typed Clojure, and then motivates the idea of automatically generating Typed Clojure annotations.

1.1 Background: Clojure with static typing

The popularity of dynamically-typed languages in software development, combined with a recognition that types often improve programmer productivity, software reliability, and performance, has led to the recent development of a wide variety of optional and gradual type systems aimed at checking existing programs written in existing languages. These include TypeScript [33] and Flow [16] for JavaScript, Hack [17] for PHP, and mypy [27] for Python among the optional systems, and Typed Racket [43], Reticulated Python [44], and Gradualtalk [2] among gradually-typed systems.[1]

One key lesson of these systems, indeed a lesson known to early developers of optional type systems such as Strongtalk, is that type systems for existing languages must be designed to work with the features and idioms of the target language. Often this takes the form of a core language, be it of functions or classes and objects, together with extensions to handle distinctive language features.

We synthesize these lessons to present *Typed Clojure*, an optional type system for Clojure. Clojure is a dynamically typed language in the Lisp family—built on the Java Virtual Machine (JVM)—which has recently gained popularity as an alternative JVM language. It offers the flexibility of a Lisp dialect, including macros, emphasizes a functional style via immutable data structures, and provides interoperability with existing Java code, allowing programmers to use existing Java libraries without leaving Clojure. Since its initial release in 2007, Clojure has been widely adopted for "backend" development in places where its support for parallelism, functional programming, and Lisp-influenced abstraction is desired on the JVM. As a result, there is an extensive base of existing untyped programs whose developers can benefit from Typed Clojure, an experience we discuss in this paper.

Since Clojure is a language in the Lisp family, we apply the lessons of Typed Racket, an existing gradual type system for Racket, to the core of Typed Clojure, consisting of an extended λ-calculus over a variety of base types

[1] We use "gradual typing" for systems like Typed Racket with sound interoperation between typed and untyped code; Typed Clojure or TypeScript which don't enforce type invariants we describe as "optionally typed".

```
(ann pname [(U File String) -> (U nil String)])
(defmulti pname class)  ; multimethod dispatching on class of argument
(defmethod pname String [s] (pname (new File s))) ; String case
(defmethod pname File [f] (.getName f)) ; File case, static null check
(pname "STAINS/JELLY") ;=> "JELLY" :- (U nil Str)
```

Figure 1: A simple Typed Clojure program (delimiters: Java interoperation (green), type annotation (blue), function invocation (black), collection literal (red), other (gray))

shared between all Lisp systems. Furthermore, Typed Racket's *occurrence typing* has proved necessary for type checking realistic Clojure programs.

However, Clojure goes beyond Racket in many ways, requiring several new type system features which we detail in this paper. Most significantly, Clojure supports, and Clojure developers use, **multimethods** to structure their code in extensible fashion. Furthermore, since Clojure is an untyped language, dispatch within multimethods is determined by application of dynamic predicates to argument values. Fortunately, the dynamic dispatch used by multimethods has surprising symmetry with the conditional dispatch handled by occurrence typing. Typed Clojure is therefore able to effectively handle complex and highly dynamic dispatch as present in existing Clojure programs.

But multimethods are not the only Clojure feature crucial to type checking existing programs. As a language built on the Java Virtual Machine, Clojure provides flexible and transparent access to existing Java libraries, and **Clojure/Java interoperation** is found in almost every significant Clojure code base. Typed Clojure therefore builds in an understanding of the Java type system and handles interoperation appropriately. Notably, `null` is a distinct type in Typed Clojure, designed to automatically rule out null-pointer exceptions.

An example of these features is given in Figure 1. Here, the `pname` multimethod dispatches on the `class` of the argument—for `String`s, the first method implementation is called, for `File`s, the second. The `String` method calls a `File` constructor, returning a non-nil `File` instance—the `getName` method on `File` requires a non-nil target, returning a nilable type.

Finally, flexible, high-performance immutable dictionaries are the most common Clojure data structure. Simply treating them as uniformly-typed key-value mappings would be insufficient for existing programs and programming styles. Instead, Typed Clojure provides a flexible **heterogenous map** type, in which specific entries can be specified.

While these features may seem disparate, they are unified in important ways. First, they leverage the type system mechanisms inherited from Typed Racket—multimethods when using dispatch via predicates, Java interoperation for handling `null` tests, and heterogenous maps using union types and reasoning about subcomponents of data. Second, they are crucial features for handling Clojure code in practice. Typed Clojure's use in real Clojure deployments would not be possible without effective handling of these three Clojure features.

1.2 Automatic Type Annotations

We now shift gears from introducing Typed Clojure to addressing a major usability flaw that many gradually and optionally typed languages share (including Typed Clojure): writing type annotations is a *manual* process. Take `vertices` (below, written in Clojure) a function that returnes the number of vertices in a tree of tagged hash-maps. As is good style, it comes with a unit test. Our goal is to *automatically generate* semi-accurate Typed Clojure annotations for this function, relieving most of the annotation burden.

```
(defn vertices [m]
  (case (:op m)
    :leaf 1
    :node (+ 1 (:left m) (:right m))))

(deftest test-vertices
  (is (= 3 {:op :node
            :left {:op :leaf :val 42}
            :right {:op :leaf :val 24}})))
```

Our approach to automatic annotations features several stages. First, we *instrument* top-level functions. Then, we *exercise* the code by running its unit tests and *observe* the runtime behavior of the program. If we pause at this point, we have collected enough data to generate a preliminary annotation:

```
(ann vertices ['{:op ':node, :left '{:op ':leaf, :val Int}, :right '{:op ':leaf, :val Int}} -> Int])
```

However, this type is too specific: trees are recursively defined and the argument type is difficult to read and maintain. To remedy this, we attempt to roll recursive-looking types to be recursive from their example unrollings. For example, below we have generalized the preliminary annotation's depth 2 tree to the recursive `NodeLeaf`.

```
(defalias NodeLeaf
  (U '{:op ':node :left NodeLeaf :right NodeLeaf}
     '{:op ':leaf :val Int}))
(ann vertices [NodeLeaf -> Int])
```

Now, if `NodeLeaf` is used in multiple positions in the program, we don't want to repeat its definition multiple times. Our type inference algorithm attempts merge recursive types found throughout the program, reusing them in annotations. For example, if another function `sum-tree` accepts two trees, we want reuse `NodeLeaf` in both annotations like so:

```
(ann vertices [NodeLeaf -> Int])
(ann sum-tree [NodeLeaf NodeLeaf -> NodeLeaf])
```

If minor variants of the recursive types occur across a program, we use optional `HMap` entries to reduce redundancy.

```
(defalias NodeLeaf
  (U '{:op ':node :left NodeLeaf :right NodeLeaf}
     (HMap :mandatory {:op ':leaf :val Int}
           :optional {:label Str})))
```

After inserting these annotations, we can run the type checker over them to check their usefulness. Ideally, minimal changes will be needed to successfully type check functions with the generated annotations, mostly consisting of local function and loop annotations, and renaming of type aliases. Annotations should also be readable and minimize redundancy, even when compared to hand-written annotations. We will test this hypothesis with case studies (Section 3.5).

2 Thesis Statement

My thesis statement is:

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

I will support this thesis statement with the following:

- *Typed Clojure is sound* We formalize Typed Clojure, including its characteristic features like hash-maps, multimethods, and Java interoperability, and prove the model type sound.
- *Typed Clojure is practical*
  - We present an empirical study of real-world Typed Clojure usage in over 19,000 lines of code, showing its features correspond to actual usage patterns.
  - To lower the annotation burden, we formalize and implement a tool to automatically annotate types for top-level user and library definitions, and empirically study the manual changes needed for the generated annotations to pass type checking.

I will augment this thesis statement with one of the following research directions:

1. More Clojure programs can be type checked by first interleaving checking with macroexpansion, then combining symbolic execution with extensible typing rules.
   - *Type checking interleaved with expansion* We motivate and describe how to convert Typed Clojure from a type system that only checks fully expanded programs to one that incrementally checks partially expanded programs, and present an implementation.
   - *Extensible type rules* We describe and implement an extensible system to define custom type rules for usages of top-level functions and macros and study how they improve the inference of core Clojure idioms.
   - *Symbolic analysis* We describe and implement symbolic rewriting strategies for Clojure programs and study how many more programs can be checked.
2. The process of porting to Typed Clojure can be partially-automated, and this automation technology can be repurposed to further reveal how Clojure is used in real projects.

- *Repurpose automation technology* We describe how to automatically generate clojure.spec annotations ("specs") for existing programs by reusing most of the the infrastructure for automatic Typed Clojure annotations. We present a formal model of clojure.spec (an existing and popular runtime verification tool for Clojure) and implement the model in Redex.
- *Study how Clojure is used in real projects* We conduct a study of general Clojure idioms and practices by generating, enforcing, and exercising specs across hundreds of projects, as well as analyzing design choices in Typed Clojure's type system, clojure.spec's features, and our automatic annotation tool.
- *Test effectiveness of clojure.spec annotation generation* We test the effectiveness of our generated specs by generating, enforcing, and exercising specs across hundreds of projects, as well as analyze design choices in Typed Clojure's type system and clojure.spec's features.

3 Technical Overview

3.1 Overview of Typed Clojure

We now begin a tour of the central features of Typed Clojure. Our presentation uses the full Typed Clojure system to illustrate key type system ideas.[2]

3.1.1 Typed Clojure

A simple one-argument function `greet` is annotated with `ann` to take and return strings.

```
(ann  greet [Str -> Str])
(defn greet [n] (str "Hello, " n "!"))
(greet "Grace") ;=> "Hello, Grace!" :- Str
```

Providing `nil` (exactly Java's `null`) is a static type error—`nil` is not a string.

```
(greet nil) ; Type Error: Expected Str, given nil
```

Unions To allow `nil`, we use *ad-hoc unions* (`nil` and `false` are logically false).

```
(ann  greet-nil [(U nil Str) -> Str])
(defn greet-nil [n] (str "Hello" (when n (str ", " n)) "!"))
(greet-nil "Donald") ;=> "Hello, Donald!" :- Str
(greet-nil nil)      ;=> "Hello!"         :- Str
```

Typed Clojure prevents well-typed code from dereferencing `nil`.

Flow analysis Occurrence typing [42] models type-based control flow. In `greetings`, a branch ensures `repeat` is never passed `nil`.

```
(ann  greetings [Str (U nil Int) -> Str])
(defn greetings [n i]
  (str "Hello, " (when i (apply str (repeat i "hello, "))) n "!"))
(greetings "Donald" 2)  ;=> "Hello, hello, hello, Donald!" :- Str
(greetings "Grace" nil) ;=> "Hello, Grace!"                :- Str
```

Removing the branch is a static type error—`repeat` cannot be passed `nil`.

```
(ann  greetings-bad [Str (U nil Int) -> Str])
(defn greetings-bad [n i]            ; Expected Int, given (U nil Int)
  (str "Hello, " (apply str (repeat i "hello, ")) n "!"))
```

[2]Full examples: `https://github.com/typedclojure/esop16`

3.1.2 Java interoperability

Clojure can interact with Java constructors, methods, and fields. This program calls the `getParent` on a constructed `File` instance, returning a nullable string.

```
(.getParent (new File "a/b"))  ;=> "a" :- (U nil Str)
```

Example 1

Typed Clojure can integrate with the Clojure compiler to avoid expensive reflective calls like `getParent`, however if a specific overload cannot be found based on the surrounding static context, a type error is thrown.

```
(fn [f] (.getParent f)) ; Type Error: Unresolved interop: getParent
```

Function arguments default to `Any`, which is similar to a union of all types. Ascribing a parameter type allows Typed Clojure to find a specific method.

```
(ann parent [(U nil File) -> (U nil Str)])
(defn parent [f] (if f (.getParent f) nil))
```

Example 2

The conditional guards from dereferencing `nil`, and—as before—removing it is a static type error, as typed code could possibly dereference `nil`.

```
(defn parent-bad-in [f :- (U nil File)]
  (.getParent f)) ; Type Error: Cannot call instance method on nil.
```

Typed Clojure rejects programs that assume methods cannot return `nil`.

```
(defn parent-bad-out [f :- File] :- Str
  (.getParent f)) ; Type Error: Expected Str, given (U nil Str).
```

Method targets can never be `nil`. Typed Clojure also prevents passing `nil` as Java method or constructor arguments by default—this restriction can be adjusted per method.

In contrast, JVM invariants guarantee constructors return non-null.[3]

```
(parent (new File s))
```

Example 3

3.1.3 Multimethods

Multimethods are a kind of extensible function—combining a *dispatch function* with one or more *methods*—widely used to define Clojure operations.

Value-based dispatch This simple multimethod takes a keyword (`Kw`) and says hello in different languages.

```
(ann hi [Kw -> Str]) ; multimethod type
(defmulti hi identity) ; dispatch function `identity`
(defmethod hi :en [_] "hello") ; method for `:en`
(defmethod hi :fr [_] "bonjour") ; method for `:fr`
(defmethod hi :default [_] "um...") ; default method
```

Example 4

When invoked, the arguments are first supplied to the dispatch function—`identity`—yielding a *dispatch value*. A method is then chosen based on the dispatch value, to which the arguments are then passed to return a value.

```
(map hi [:en :fr :bocce]) ;=> ("hello" "bonjour" "um...")
```

For example, `(hi :en)` evaluates to `"hello"`—it executes the `:en` method because `(= (identity :en) :en)` is true and `(= (identity :en) :fr)` is false.

Dispatching based on literal values enables certain forms of method definition, but this is only part of the story for multimethod dispatch.

Class-based dispatch For class values, multimethods can choose methods based on subclassing relationships. Recall the multimethod from Figure 1. The dispatch function `class` dictates whether the `String` or `File` method is chosen. The multimethod dispatch rules use `isa?`, a hybrid predicate which is both a subclassing check for classes and an equality check for other values.

```
(isa? :en :en)       ;=> true
(isa? String Object) ;=> true
```

The current dispatch value and—in turn—each method's associated dispatch value is supplied to `isa?`. If exactly one method returns true, it is chosen. For example, the call `(pname "STAINS/JELLY")` picks the `String` method because `(isa? String String)` is true, and `(isa? String File)` is not.

[3] http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.9.4

3.1.4 Heterogeneous hash-maps

The most common way to represent compound data in Clojure are immutable hash-maps, typicially with keyword keys. Keywords double as functions that look themselves up in a map, or return `nil` if absent.

Example 5
```
(def breakfast {:en "waffles" :fr "croissants"})
(:en breakfast)    ;=> "waffles" :- Str
(:bocce breakfast) ;=> nil       :- nil
```

HMap types describe the most common usages of keyword-keyed maps.

```
breakfast ; :- (HMap :mandatory {:en Str, :fr Str}, :complete? true)
```

This says `:en` and `:fr` are known entries mapped to strings, and the map is fully specified—that is, no other entries exist—by `:complete?` being `true`.

HMap types default to partial specification, with `'{:en Str :fr Str}` abbreviating `(HMap :mandatory {:en Str, :fr Str})`.

Example 6
```
(ann lunch '{:en Str :fr Str})
(def lunch {:en "muffin" :fr "baguette"})
(:bocce lunch) ;=> nil :- Any ; less accurate type
```

3.2 A Formal Model of Typed Clojure

In this section, we present a fragment of a formal model of Typed Clojure called λ_{TC}, deferring any treatment of Java interoperability, multimethods, and heterogeneous maps to the thesis. Our presentation will build on occurrence typing, then incrementally add each novel feature of Typed Clojure to the formalism, interleaving presentation of syntax, typing rules, operational semantics, and subtyping.

3.3 Core type system

We start with a review of occurrence typing [42], the foundation of λ_{TC}.

Expressions Syntax is given in Figure 2. Expressions e include variables x, values v, applications, abstractions, conditionals, and let expressions. All binding forms introduce fresh variables—a subtle but important point since our type environments are not simply dictionaries. Values include booleans b, nil, class literals C, keywords k, integers n, constants c, and strings s. Lexical closures $[\rho, \lambda x^{\tau}.e]_{\mathsf{c}}$ close value environments ρ—which map bindings to values—over functions.

Types Types σ or τ include the top type $\top$, *untagged* unions $(\bigcup\ \overrightarrow{\tau})$, singletons $(\mathbf{Val}\, l)$, and class instances C. We abbreviate the classes **Boolean** to **B**, **Keyword** to **K**, **Nat** to **N**, **String** to **S**, and **File** to **F**. We also abbreviate the types $(\bigcup)$ to $\bot$, $(\mathbf{Val}\,\mathsf{nil})$ to **nil**, $(\mathbf{Val}\,\mathsf{true})$ to **true**, and $(\mathbf{Val}\,\mathsf{false})$ to **false**. The difference between the types $(\mathbf{Val}\, C)$ and C is subtle. The former is inhabited by class literals like **K** and the result of $(\mathit{class}\ \mathsf{:a})$—the latter by *instances* of classes, like a keyword literal $\mathsf{:a}$, an instance of the type **K**. Function types $x{:}\sigma \xrightarrow[o]{\psi|\psi} \tau$ contain *latent* (terminology from [30]) propositions ψ, object o, and return type τ, which may refer to the function argument x. They are instantiated with the actual object of the argument in applications.

Objects Each expression is associated with a symbolic representation called an *object*. For example, variable m has object m; $(\mathbf{class}\ (\mathsf{:lunch}\ m))$ has object $\mathbf{class}(\mathbf{key}_{\mathsf{:lunch}}(m))$; and 42 has the *empty* object $\emptyset$ since it is unimportant in our system. Figure 2 gives the syntax for objects o—non-empty objects $\pi(x)$ combine of a root variable x and a *path* π, which consists of a possibly-empty sequence of *path elements* (pe) applied right-to-left from the root variable. We use two path elements—**class** and $\mathbf{key}_k$—representing the results of calling *class* and looking up a keyword k, respectively.

Propositions with a logical system Occurrence typing pairs *logical formulas*, that can reason about arbitrary non-empty objects, with a *proof system*. The logical statement σ_x says variable x is of type σ. We further extend logical statements to *propositional logic*. Figure 2 describes the syntax for propositions ψ, consisting of positive and negative *type propositions* about non-empty objects—$\tau_{\pi(x)}$ and $\overline{\tau}_{\pi(x)}$ respectively—the latter pronounced "the object $\pi(x)$ is *not* of type τ". The other propositions are standard logical connectives: implications, conjunctions, disjunctions, and the trivial $(\mathbb{tt})$ and impossible $(\mathbb{ff})$ propositions. The full proof system judgement $\Gamma \vdash \psi$ says *proposition environment* Γ proves proposition ψ.

$$
\begin{array}{llll}
e & ::= & x \mid v \mid (e\ e) \mid \lambda x^{\tau}.e \mid (\text{if } e\ e\ e) \mid (\text{let } [x\ e]\ e) & \text{Expressions} \\
v & ::= & l \mid n \mid c \mid s \mid [\rho, \lambda x^{\tau}.e]_{\mathsf{c}} & \text{Values} \\
c & ::= & \mathit{class} \mid \mathit{n?} & \text{Constants} \\
\sigma, \tau & ::= & \top \mid (\bigcup\ \overrightarrow{\tau}) \mid x{:}\tau \xrightarrow[o]{\psi|\psi} \tau \mid (\mathbf{Val}\, l) \mid C & \text{Types} \\
l & ::= & k \mid C \mid \mathsf{nil} \mid b & \text{Value types} \\
b & ::= & \mathsf{true} \mid \mathsf{false} & \text{Boolean values} \\
 & & & \\
\psi & ::= & \tau_{\pi(x)} \mid \overline{\tau}_{\pi(x)} \mid \psi \supset \psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathtt{tt} \mid \mathtt{ff} & \text{Propositions} \\
o & ::= & \pi(x) \mid \emptyset & \text{Objects} \\
\pi & ::= & \overrightarrow{pe} & \text{Paths} \\
pe & ::= & \mathbf{class} \mid \mathbf{key}_k & \text{Path elements} \\
 & & & \\
\Gamma & ::= & \overrightarrow{\psi} & \text{Proposition environments} \\
\rho & ::= & \{\overrightarrow{x \mapsto v}\} & \text{Value environments}
\end{array}
$$

Figure 2: Syntax of Terms, Types, Propositions and Objects

$$
\textsc{T-Local}\ \frac{\Gamma \vdash \tau_x \qquad \sigma = (\cup\ \mathbf{nil}\ \mathbf{false})}{\Gamma \vdash x : \tau\ ;\ \overline{\sigma}_x | \sigma_x\ ;\ x}
\qquad
\textsc{T-Abs}\ \frac{\Gamma, \sigma_x \vdash e : \sigma'\ ;\ \psi_+|\psi_-\ ;\ o}{\Gamma \vdash \lambda x^{\sigma}.e : x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \sigma'\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset}
$$

$$
\textsc{T-If}\ \frac{\Gamma \vdash e_1 : \tau_1\ ;\ \psi_{1+}|\psi_{1-}\ ;\ o_1 \qquad \Gamma, \psi_{1+} \vdash e_2 : \tau\ ;\ \psi_+|\psi_-\ ;\ o \qquad \Gamma, \psi_{1-} \vdash e_3 : \tau\ ;\ \psi_+|\psi_-\ ;\ o}{\Gamma \vdash (\text{if } e_1\ e_2\ e_3) : \tau\ ;\ \psi_+|\psi_-\ ;\ o}
$$

$$
\textsc{T-App}\ \frac{\Gamma \vdash e : x{:}\sigma \xrightarrow[o_f]{\psi_{f+}|\psi_{f-}} \tau\ ;\ \psi_+|\psi_-\ ;\ o \qquad \Gamma \vdash e' : \sigma\ ;\ \psi'_+|\psi'_-\ ;\ o'}{\Gamma \vdash (e\ e') : \tau[o'/x]\ ;\ \psi_{f+}|\psi_{f-}[o'/x]\ ;\ o_f[o'/x]}
$$

$$
\textsc{T-Subsume}\ \frac{\Gamma \vdash e : \tau\ ;\ \psi_+|\psi_-\ ;\ o \qquad \Gamma, \psi_+ \vdash \psi'_+ \qquad \Gamma, \psi_- \vdash \psi'_- \qquad \vdash \tau <: \tau' \qquad \vdash o <: o'}{\Gamma \vdash e : \tau'\ ;\ \psi'_+|\psi'_-\ ;\ o'}
$$

Figure 3: Select core typing rules

Each expression is associated with two propositions—when expression e_1 is in test position like (if e_1 e_2 e_3), the type system extracts e_1's 'then' and 'else' proposition to check e_2 and e_3 respectively. For example, in (if o e_2 e_3) we learn variable o is true in e_2 via o's 'then' proposition $\overline{(\cup\ \mathbf{nil}\ \mathbf{false})}_o$, and that o is false in e_3 via o's 'else' proposition $(\cup\ \mathbf{nil}\ \mathbf{false})_o$.

Typing judgment We formalize our system following Tobin-Hochstadt and Felleisen [42]. The typing judgment $\Gamma \vdash e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$ says expression e rewrites to e', which is of type τ in the proposition environment Γ, with 'then' proposition ψ_+, 'else' proposition ψ_- and object o.

We write $\Gamma \vdash e \Rightarrow e' : \tau$ to mean $\Gamma \vdash e : \tau\ ;\ \psi'_+|\psi'_-\ ;\ o'$ for some ψ'_+, ψ'_- and o', and abbreviate self rewriting judgements $\Gamma \vdash e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$ to $\Gamma \vdash e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$.

Typing rules A selection of core typing rules are given as Figure 3.

$$
\textsc{S-UnionSuper}\ \frac{\exists i.\ \vdash \tau <: \sigma_i}{\vdash \tau <: (\bigcup\ \overrightarrow{\sigma}^{i})}
\qquad
\textsc{S-UnionSub}\ \frac{\overrightarrow{\vdash \tau_i <: \sigma}^{i}}{\vdash (\bigcup\ \overrightarrow{\tau}^{i}) <: \sigma}
$$

$$
\textsc{S-Fun}\ \frac{\vdash \sigma' <: \sigma \qquad \vdash \tau <: \tau' \qquad \psi_+ \vdash \psi'_+ \qquad \psi_- \vdash \psi'_- \qquad \vdash o <: o'}{\vdash x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau <: x{:}\sigma' \xrightarrow[o']{\psi'_+|\psi'_-} \tau'}
$$

Figure 4: Select core subtyping rules

The essense of occurrence typing is contained in the first three rules T-Local, T-Abs, and T-If. The first rule T-Local type checks local bindings, and defers to the proof system to infer its type. The propositions on a local binding say that if bound value is logically true (not nil or false), then so is its binding, and similar if the value is logically false (nil or false). The symbolic object of a local binding is itself.

The second rule T-Abs extends the type environment with information on the parameter's type to type check the body. It stores latent propositions and object in the resulting function type, which are used when the function is applied in T-App. Notice, T-App substitutes the actual argument object for the parameter's name in the resulting type, propositions, and object of an application.

The third rule T-If extends the type environment with information learnt from the test. For example, to check the then branch, the proposition environment is extended with the test's then proposition. The subsumption rule T-Subsume allows us to combine the resulting type, propositions, and object of both branches as the resulting type, propositions, and object of the entire expression.

Subtyping Subtyping is as a reflexive and transitive relation with top type $\top$. Figure 4 presents select rules. Subtyping for untagged unions is standard. Function subtyping is contravariant left of the arrow—latent propositions, object and result type are covariant.

Operational semantics We define the dynamic semantics for λ_{TC} in a big-step style using an environment, following [42]. We include both errors and a *wrong* value, which is provably ruled out by the type system. The main judgment is $\rho \vdash e \Downarrow \alpha$ which states that e evaluates to answer α in environment ρ. We omit the core rules (included in supplemental material of [5]).

3.4 Metatheory of Typed Clojure

We will prove type soundness following Tobin-Hochstadt and Felleisen [42]. Our model is extended to include errors err and a *wrong* value, and we prove well-typed programs do not go wrong; this is therefore a stronger theorem than proved by Tobin-Hochstadt and Felleisen [42]. Errors behave like Java exceptions—they can be thrown and propagate "upwards" in the evaluation rules.

Rather than modeling Java's dynamic semantics, a task of daunting complexity, we instead make our assumptions about Java explicit. We concede that method and constructor calls may diverge or error, but assume they can never go wrong (other assumptions will be provided in the thesis).

We can now state our main lemma and soundness theorem. The metavariable α ranges over v, err and *wrong*.

Lemma 1. *If* $\Gamma \vdash e' : \tau \ ; \ \psi_+|\psi_- \ ; \ o$, $\rho \models \Gamma$, ρ *is consistent, and* $\rho \vdash e \Downarrow \alpha$ *then either*

- $\rho \vdash e \Downarrow v$ *and all of the following hold:*
  1. *either* $o = \emptyset$ *or* $\rho(o) = v$,
  2. *either* $\mathsf{TrueVal}(v)$ *and* $\rho \models \psi_+$ *or* $\mathsf{FalseVal}(v)$ *and* $\rho \models \psi_-$,
  3. $\vdash v : \tau \ ; \ \psi'_+|\psi'_- \ ; \ o'$ *for some* ψ'_+, ψ'_- *and* o', *and*
  4. v *is consistent with* ρ, *or*
- $\rho \vdash e \Downarrow \mathsf{err}$.

Theorem 1 (Type soundness)**.** *If* $\Gamma \vdash e' : \tau \ ; \ \psi_+|\psi_- \ ; \ o$ *and* $\rho \vdash e \Downarrow v$ *then* $\vdash v : \tau \ ; \ \psi'_+|\psi'_- \ ; \ o'$ *for some* ψ'_+, ψ'_- *and* o'.

The full thesis will provide a proof of Theorem 1.

3.5 Typed Clojure Evaluation

Throughout this thesis, we will focus on three interrelated type system features: heterogeneous maps, Java interoperability, and multimethods. Our hypothesis is that these features are widely used in existing Clojure programs in interconnecting ways, and that handling them as we have done is required to type check realistic Clojure programs.

To evaluate this hypothesis, we will present an analysis of two existing `core.typed` code bases, one from the open-source community, and one from a company that uses `core.typed` in production (Figure 5).

	feeds2imap	CircleCI
Total number of typed namespaces	11 (825 LOC)	87 (19,000 LOC)
Total number of `def` expressions	93	1834
• checked	52 (56%)	407 (22%)
• unchecked	41 (44%)	1427 (78%)
Total number of Java interactions	32	105
• static methods	5 (16%)	26 (25%)
• instance methods	20 (62%)	36 (34%)
• constructors	6 (19%)	38 (36%)
• static fields	1 (3%)	5 (5%)
Methods overriden to return non-nil	0	35
Methods overriden to accept nil arguments	0	1
Total HMap lookups	27	328
• resolved to mandatory key	20 (74%)	208 (64%)
• resolved to optional key	6 (22%)	70 (21%)
• resolved of absent key	0 (0%)	20 (6%)
• unresolved key	1 (4%)	30 (9%)
Total number of `defalias` expressions	18	95
• contained HMap or union of HMap type	7 (39%)	62 (65%)
Total number of checked `defmulti` expressions	0	11
Total number of checked `defmethod` expressions	0	89

Figure 5: Typed Clojure Features used in Practice

3.6 Automatic Annotation Approach

We now describe our philosophy and overall approach to automatic annotations for Typed Clojure. At a high level, there are three phases to generating annotations: instrumentation, runtime tracking, and type reconstruction.

The first phase, **instrumentation**, involves rewriting the code we wish to annotate such that we can record its runtime behavior. In this phase, we usually require the programmer to indicate which code we wish to generate types for in advance, with a file-level granuality.

Once instrumented, we observe our running program via **runtime tracking**. To exercise our programs, we usually run their unit tests, generative tests, or just normally run the program (eg. to generate types for a game, we can simply play the game for a few minutes). We accumulate the results of tracking via **paths**. If we think of types as trees and supply a label for each branching path, our inference results specify the type down a particular path in this tree.

Finally, the information collected during runtime tracking is combined into annotations by our **inference algorithm**. We first combine all inference result into a large tree of types. If we were to convert this tree into annotations directly, our annotations would be too specific—they would be too deep and fine-grained. Instead, our algorithm iterates over several passes to massage this tree, generating good names for the nodes, compacting similar types across the tree, and eventually converting the tree into a directed graph by reconstructing recursive types.

An important question to answer is "how accurate are these annotations?". Unlike some previous work in this area [3], we do not aim for soundness guarantees in our generated types. Our main contribution is a tool that Clojure programmers can use to help learn about and specify their programs. In that spirit, annotations should meet several criteria.

Good names Typed Clojure annotations are abundant with useful names for types. A good name often increases readability. Good names can sometimes be reconstructed from the program source, like function or parameter names, and other times we can use the shape of a type to summarize it.

Compact Idiomatic Clojure code rarely mixes certain types in the same position, unless the program is polymorphic. Using this knowledge—which we observed by the annotations and specs assigned to idiomatic Clojure code—we can rule out certain combinations of types to compact our resulting output, without losing information that would help us type check our programs.

Recursive Maps in Clojure are often heterogeneous, and recursively defined. Typed Clojure supplies mechanisms for the most common case: maps of known keyword entries. We strategically **squash** flat types to be recursive based

$$
\begin{array}{llll}
v & ::= & n \mid k \mid [\lambda x.e, \rho] \mid \{\overrightarrow{k\ v}\} & \text{Values} \\
e & ::= & x \mid v \mid (\textbf{track}\ e\ \pi) \mid \lambda x.e \mid \{\overrightarrow{e\ e}\} \mid (e\ e) & \text{Expressions} \\
\rho & ::= & \{\overrightarrow{x \mapsto v}\} & \text{Runtime environments} \\
l & ::= & x \mid \textbf{dom} \mid \textbf{rng} \mid \textbf{key}_{\overrightarrow{k}}(k) & \text{Path Elements} \\
\pi & ::= & \overrightarrow{l} & \text{Paths} \\
r & ::= & \{\overrightarrow{\pi : \tau}\} & \text{Inference results} \\
\tau, \sigma & ::= & \mathsf{N} \mid [\tau \to \tau] \mid \{\overrightarrow{k\ \tau}\} & \\
 & \mid & (\mathsf{HMap}\ \{\overrightarrow{k\ \tau}\}\ \{\overrightarrow{k\ \tau}\}) \mid \tau \cup \tau \mid a \mid ? & \text{Types} \\
\Gamma & ::= & \{\overrightarrow{x : \tau}\} & \text{Type environments} \\
A & ::= & \{\overrightarrow{a \mapsto \tau}\} & \text{Type alias environments} \\
\Delta & ::= & (A, \Gamma) & \text{Combined environments}
\end{array}
$$

Figure 6: Grammar for the Automatic Annotation tool

B-Var
$$\rho \vdash x \Downarrow \rho(x)\ ; \{\}$$

B-Track
$$\frac{\rho \vdash e \Downarrow v\ ; r \qquad \mathsf{track}(v, \pi) = v'\ ; r'}{\rho \vdash (\textbf{track}\ e\ \pi) \Downarrow v'\ ; r \sqcup r'}$$

B-App
$$\frac{\begin{array}{c}\rho \vdash e_1 \Downarrow [\lambda x.e, \rho']\ ; r_1 \\ \rho \vdash e_2 \Downarrow v\ ; r_2 \\ \rho', x \mapsto v \vdash e \Downarrow v'\ ; r_3\end{array}}{\rho \vdash (e_1\ e_2) \Downarrow v'\ ; r_1 \sqcup r_2 \sqcup r_3}$$

B-Clos
$$\rho \vdash \lambda x.e \Downarrow [\lambda \overrightarrow{x}.e, \rho]\ ; \{\}$$

B-Val
$$\rho \vdash v \Downarrow v\ ; \{\}$$

B-Get
$$\frac{\rho \vdash e_1 \Downarrow \{\overrightarrow{k\ v}\}\ ; r_1 \qquad \rho \vdash e_2 \Downarrow k_1\ ; r_2}{\rho \vdash (get\ e_1\ e_2) \Downarrow \{\overrightarrow{k\ v}\}[k_1]\ ; r_1 \sqcup r_2}$$

B-Assoc
$$\frac{\rho \vdash e_1 \Downarrow \{\overrightarrow{k\ v}\}\ ; r_1 \qquad \rho \vdash e_2 \Downarrow k_1\ ; r_2 \qquad \rho \vdash e_3 \Downarrow v\ ; r_3}{\rho \vdash (\textbf{assoc}\ e_1\ e_2\ e_3) \Downarrow \{\overrightarrow{k\ v}\}[k_1 \mapsto v]\ ; r_1 \sqcup r_2 \sqcup r_3}$$

Figure 7: Runtime instrumentation semantics for the automatic annotation tool

on their unrolled shape. For example, a recursively defined union of maps almost always contains a known keyword "tag" mapped to a keyword. By identifying this tag, we can reconstruct a good recursive approximation of this type.

3.7 Automatic Annotations Formalism

We provide a preliminary grammar for our formal treatment of automatic annotations, and a semantics for the instrumentation phase of our automatic annotator. The full thesis will describe the type reconstruction algorithm in detail.

Figure 6 presents the grammar. Similar to Typed Clojure's formalism, we use paths π to represent a path through a value. Path elements consist of the domain of a function **dom**, the range of a function **rng**, and the result of a map lookup of key k_2 on a map with key set $\overrightarrow{k_1}$ $\textbf{key}_{\overrightarrow{k_1}}(k_2)$. Inference results r are collected during execution, and associate a path π with the type of value observed at that path τ.

Figure 7 gives a semantics for the instrumentation phase of the automatic annotation tool. Most rules are standard, with B-Get and B-Assoc responsible for looking up and associating new entries onto maps. The B-Track rule is the entry point for tracking values, calling track (Figure 8), that rewrites a value v to v' and generates inference results r' based on the input that π is the path of v.

Several extensions to this model are possible. First, we will add *space-efficient* runtime instrumentation, This ensures that stack space usage is efficient by collapsing directly stacked wrappers of the same kind. For example, wrapping a function twice takes the same stack space as wrapping once. The propagation of path information is then extended to support *multiple* simultaneous paths for each value, which enables the optimization to not lose any tracking information.

Second, we can infer *polymorphic* types by tracking the pointer identity of objects. This information can be combined with the base tracking information to generate polymorphic type annotations that also feature concrete types when appropriate.

$$\mathsf{track}(v, \pi) = v \ ; \ r$$

$$\begin{array}{lcl}
\mathsf{track}(n, \pi) & = & n \ ; \{\pi : \mathsf{N}\} \\
\mathsf{track}([\lambda x.e, \rho], \pi) & = & [\lambda y.(\textbf{track}\ ((\lambda x.e)(\textbf{track}\ y\ \pi :: [\textbf{dom}]))\ \pi :: [\textbf{rng}]), \rho] \ ; \{\pi : [? \rightarrow ?]\} \\
 & & \text{where } y \text{ is fresh} \\
\mathsf{track}(\{\overrightarrow{v_1\ v_2}\}, \pi) & = & \{\overrightarrow{v_1\ {v_2}'}\} \ ; \ \overrightarrow{\sqcup r} \sqcup \{\pi : \{\overrightarrow{v_1\ ?}\}\} \\
 & & \overrightarrow{\text{where } \mathsf{track}(v_2, \pi :: [\textbf{key}_{\overrightarrow{v_1}}(v_1)]) = {v_2}' \ ; \ r}
\end{array}$$

Figure 8: Value tracking

Library	Lines of types	Local annotations	Manual Line +/- Diff
startrek-clojure	133	3	+70 -41
math.combinatorics	395	147	+124 -120
fs	157	1	+119 -86
data.json	168	9	+94 -125
mini.occ	49	1	+46 -26

Figure 9: Amending automatically generated types to type check

3.8 Typed Clojure Automatic Annotations Evaluation

Along with a manual inspection of the generated Typed Clojure annotations, we will perform several other experiments to measure the quality of the generated annotations.

For example, we measure the number of changes needed needed to amend generated annotations to actually type check. Figure 9 shows some preliminary results.

3.9 Two Areas of Further Study

The following two subsections describe several possible research areas for the final section of this dissertation.

At least one of the following investigations will be included in the dissertation. The selection process will be informed by the success of early prototypes and preliminary investigations.

3.9.1 Option 1: Type check more Clojure programs

A complaint from industrial users about Typed Clojure is that support for core Clojure functionality and idioms is limited. Furthermore, even if a set of functionality is supported, combining them in a helper function often give undesired results.

CircleCI summarized their experience with Typed Clojure [13], citing several specific frustrations. One representative issue we will discuss is that Typed Clojure cannot reliably infer usages of polymorphic higher-order functions. The root of this problem is known as the *Hard to synthesize arguments* problem [23], and has two competing forces: local type argument synthesis for polymorphic applications and the bidirectional propagation of anonymous function argument types.

For example, consider mapping a collection over the identity function `(map (fn [x] x) [1 2 3])`. To infer `map`'s type variables, we first type check the arguments. Unfortunately, `(fn [x] x)` has no annotation, and so its type is `[Any -> Any]`, making the entire invocation type `(Seq Any)` (instead of the more desirable `(Seq Int)`).

Even `(map identity [1 2 3])` has its own set of problems: both `map` and `identity` are polymorphic, and thus cannot be inferred accurately by many systems based on Pierce and Turner's Local Type Inference [37] (like Typed Clojure and Typed Racket). This specific problem is addressed by later work in the realm of set-theoretic types [8], by generating substitutions for the arguments of applications. However, even there, it cannot simultaneously infer the type of the function argument of `(map (fn [x] x) [1 2 3])`.

Anonymous functions are common in Clojure. For example, the ClojureScript compiler uses around 40 anonymous functions in around 140 top-level definitions. Most of these functions would require extra annotations to type check, and over half of them are used as arguments to polymorphic higher-order functions.

To address excessive annotations and broaden the number of checkable programs, we will design and implement an approach to mitigate the issue of *Hard to synthesize arguments* in common Clojure code, and report its the effectiveness in reducing the number of changes in the porting process from Clojure to Typed Clojure.

Some initial investigation in this direction has centered around building an extensible system for specifying type rules, and enhancing symbolic execution capabilities. For example `(update m :a (fn [a] (inc a)))` increments the `:a` entry of map `m`. A custom `update` rule effectively inlines the call to `update` as `(assoc m :a ((fn [a] (inc a)) (get m :a)))`, which allows parameter `a` to inherit the type of `(get m :a)`.

A more complicated example uses Clojure transducers. The expression `(comp (map (fn [a] (dec a))) (map (fn [a] (inc a))))` is a transducer that first decrements, then increments (transducers compose left-to-right). Since `comp`, and `map` are polymorphic higher-order functions, and we use anonymous functions, this is hard to type check. However, by combining custom type rules for `comp`, and `map` with symbolic computations and adequate type propagation, we could potentially distil the original expression to `(inc (dec input))`, which is much easier to check.

3.9.2 Option 2: clojure.spec Automatic Annotations

Clojure.spec is popular runtime verification system for Clojure programs included in the core Clojure distribution. We will repurpose our automatic annotation tool to generate clojure.spec runtime specifications.

We will then conduct a larger scale investigation to evaluate the quality of these annotations by generating clojure.spec annotations for several hundred projects we have sourced from the open source community. This investigation will also serve to further analyze the assumptions made in designing Typed Clojure and clojure.spec. For example, Typed Clojure and clojure.spec are designed differently around qualified entries in maps, and by generating and enforcing specs we can investigate whether either are compatible with actual usage.

We can empirically investigate which function spec checking semantics are applicable in the majority of code. Since clojure.spec provides two distinct function checking semantics, including a surprising "generative testing" semantics, we can measure the likelihood of tests still passing after generating each kind of function spec.

Other interesting questions can be asked by using clojure.spec's generative testing features. After generating our own specs automatically for each project, we can test to see if our annotation algorithm feeds sufficient information to clojure.spec's value generators to create effective generative tests. We can also use the generative tests as a "second pass" over the functions, by re-instrumenting our functions and analyzing whether we get better test coverage than just running the provided unit tests.

clojure.spec supports stubbing-out functions, useful for running unit tests that depend on side effecting functions. For example, we might stub-out functions that call a database that only exists in production. Using our generated specs, we could experimentally stub-out functions that consistently fail their specs under generative testing, and measure how useful the chosen stubs are by manually inspecting which functions were stubbed. Similarly, we could experiment in stubbing out function arguments to higher-order functions to make instrumentation with `fspec`s more predictable.

There is potential for other, more general, research questions to be answered about Clojure usage with our testing setup. Part of the unfinished work towards this thesis will be devising research questions, but, for now, we can imagine a spectrum of questions from simple measurements of the frequency of particular features like variable-arguments, keyword maps, or higher-order functions, to more ambitious questions like detecting breaking changes between project versions and quantifying the test coverage of unit tests versus generative tests.

4 Related Work

Typed Multimethods Millstein and collaborators present a sequence of systems [9, 10, 34] with statically-typed multimethods and modular type checking. In contrast to Typed Clojure, in these system methods declare the types of arguments that they expect which corresponds to exclusively using `class` as the dispatch function in Typed Clojure. However, Typed Clojure does not attempt to rule out failed dispatches.

Occurrence Typing Occurrence typing [43, 42] extends the type system with a *proposition environment* that represents the information on the types of bindings down conditional branches. These propositions are then used to update the types associated with bindings in the *type environment* down branches so binding occurrences are given different types depending on the branches they appear in, and the conditionals that lead to that branch.

Record Types Row polymorphism [45, 7, 21], used in systems such as the OCaml object system, provides many of the features of HMap types, but defined using universally-quantified row variables. HMaps in Typed Clojure are instead designed to be used with subtyping, but nonetheless provide similar expressiveness, including the ability to require presence and absence of certain keys.

Dependent JavaScript [12] can track similar invariants as HMaps with types for JS objects. They must deal with mutable objects, they feature refinement types and strong updates to the heap to track changes to objects.

TeJaS [28], another type system for JavaScript, also supports similar HMaps, with the ability to record the presence and absence of entries, but lacks a compositional flow-checking approach like occurrence typing.

Typed Lua [31] has *table types* which track entries in a mutable Lua table. Typed Lua changes the dynamic semantics of Lua to accommodate mutability: Typed Lua raises a runtime error for lookups on missing keys—HMaps consider lookups on missing keys normal.

Java Interoperability in Statically Typed Languages Scala [35] has nullable references for compatibility with Java. Programmers must manually check for `null` as in Java to avoid null-pointer exceptions.

Other optional and gradual type systems Several other gradual type systems have been developed for existing dynamically-typed languages. Reticulated Python [44] is an experimental gradually typed system for Python, implemented as a source-to-source translation that inserts dynamic checks at language boundaries and supporting Python's first-class object system. Clojure's nominal classes avoids the need to support first-class object system in Typed Clojure, however HMaps offer an alternative to the structural objects offered by Reticulated. Similarly, Gradualtalk [2] offers gradual typing for Smalltalk, with nominal classes.

Optional types have been adopted in industry, including Hack [17], and Flow [16] and TypeScript [33], two extensions of JavaScript. These systems support limited forms of occurrence typing, and do not include the other features we present.

Automatic annotations There are two common implementation strategies for such tools. The first strategy, "ruling-out" (for invariant detection), assumes all invariants are true and then use runtime analysis results to rule out impossible invariants. The second "building-up" strategy (for dynamic type inference) assumes nothing and then uses runtime analysis results to build up invariant/type knowledge.

Examples of invariant detection tools include Daikon [15], DIDUCE [20], and Carrot [40], and typically enhance statically typed languages with more expressive types or contracts. Examples of dynamic type inference include Rubydust [3], JSTrace [41], and TypeDevil [39], and typically target untyped languages.

Both strategies have different space behavior with respect to representing the set of known invariants. The ruling-out strategy typically uses a lot of memory at the beginning, but then can free memory as it rules out invariants. For example, if `odd(x)` and `even(x)` are assumed, observing `x = 1` means we can delete and free the memory recording `even(x)`. Alternatively, the building-up strategy uses the least memory storing known invariants/types at the beginning, but increases memory usage as more the more samples are collected. For example, if we know `x : Bottom`, and we observe `x = "a"` and `x = 1` at different points in the program, we must use more memory to store the union `x : String` $\cup$ `Integer` in our set of known invariants.

Daikon Daikon can reason about very expressive relationships between variables using properties like ordering ($x < y$), linear relationships ($y = ax + b$), and containment ($x \in y$). It also supports reasoning with "derived variables" like fields ($x.f$), and array accesses ($a[i]$).

Typed Clojure's dynamic inference can record heterogeneous data structures like vectors and hash-maps, but otherwise cannot express relationships between variables.

There are several reasons for this. The most prominent is that Daikon primarily targets Java-like languages, so inferring simple type information would be redundant with the explicit typing disciplines of these languages. On the other hand, the process of moving from Clojure to Typed Clojure mostly involves writing simple type signatures without dependencies between variables. Typed Clojure recovers relevant dependent information via occurrence typing, and gives the option to manually annotate necessary dependencies in function signatures when needed.

TypeScript Annotation Generation Kristensen and Møller [26] present TSInfer and TSEvolve that generate TypeScript annotation files using static analysis of JavaScript code. They submitted corrections back to libraries they found descrepancies in, which were accepted with little to no changes in the tool's output.

NoRegrets [32] uses dynamic analysis to learn how a program is used, and automatically runs the tests of downstream projects to improve test coverage. Its concept of representing a program sample as a path paired with a type is very similar to Typed Clojure's approach.

How dynamic languages are used Several languages have seen similar investigations into their idioms as I am proposing for Clojure.

A popular motivation is to discover which type system features to support when retrofitting a type system. Akerblom et. al [1] trace dynamic features in Python programs via instrumentation. They measured the prevalence of dynamic features in startup versus user code, and recorded usage frequencies for a set of dynamic features. They concluded dynamism is prevalent in Python, and thus should be supported in a retrofitted type system for Python. A study along similar lines is also applicable to Clojure, in particular analysing Typed Clojure's support for Clojure's dynamic features.

Callaú et al. [6] also conducted a large-scale study of dynamic Smalltalk idioms to inform future language extensions tooling support. Notably, they further perform a qualitative analysis aiming to identify the reasons why Smalltalk use these features in the first place, and whether they can be replaced with more predictable features. They also measure which kinds of projects (e.g., testing frameworks, user-level libraries, or core system libraries) use particular features more frequently. Due to the their prevalence in the open-source Clojure ecosystem, Typed Clojure has mainly been tested on user-level libraries. We could predict Typed Clojure's applicability to other kinds of projects by gathering similar data on how frequently different types of Clojure libraries use Clojure's various features.

Andreasen et. al [4] developed *trace typing* to explore the design space of JavaScript type systems. Using runtime observations, they studied which control flow techniques are used most often in JavaScript programs, and thus, which should be supported by an effective type system for JavaScript. Typed Clojure implements occurrence typing to reason about control flow in Clojure which seems to work well in practice, but a similar quantitative analysis could reveal further insights.

Interleaving Type Checking with Expansion, Extensible type systems and Symbolic Analysis Turnstile [11] type checks a program during expansion by repurposing the Racket macro system. It provides a fully extensible framework for specifying and combining core typing rules. On the other hand, Typed Clojure does not have the goal of allowing users to override how language primitives type check. Instead, our goal is to provide a simple interface to write type rules for library functions and macros in a style that hides the necessary bookkeeping surrounding occurrence typing and scope management.

SugarJ [14] adds syntactic language extensibility to languages like Java, such as pair syntax, embedded XML, and closures. Desugarings are expressed as rewrite rules to plain Java. Similarly, work on *type-specific languages* [36] adds extensible systems for the definitions of specialized syntax literals to existing languages. The *type* of an expression determines how it is parsed and elaborated.

SoundX [29] presents a solution to a common dilemma in typed metaprogramming: whether to desugar before type checking, or vice-versa. The authors present a system where a form is type checked before being desugared, with a guarantee that only well-typed code is generated. Programmers specify desugarings with a combination of typing and rewriting rules, which are then connected to form a valid type derivation in a process called *forwarding.* We will explore whether we can get the same effect in Typed Clojure without requiring the user to understand typing rules.

Ziggurat [18] allows programmers to define the static and dynamic semantics of macros separately. To demonstrate its broad applicability, they choose Scheme-like macros that generate assembly code for the dynamic semantics. They advocate building towers of static analyses, so macros can be statically checked in terms the static semantics of other macros, instead of just their assembly code expansions which would otherwise be too difficult to check. This idea resembles our prototypes in defining custom typing rules for functions and macros in Typed Clojure, where the dynamic semantics are defined by runtime Clojure constructs (`defn` and `defmacro`), and towers of static semantics are progressively specified in terms of the static analysis of other Clojure forms.

Mix [24] cleanly separates symbolic execution [25] from type checking in the same system, specifying a mode for (nested) regions of code. They argue this tradeoff keeps the predictability of type checking, while preserving enough symbolic execution to drive further checking. In Typed Clojure, symbolic execution is managed by occurrence typing [42]. Our preliminary explorations in symbolic execution for Typed Clojure, for example, type checks an anonymous function if annotated, otherwise treats it symbolically. As the authors envision, this is akin to automatically inserting the mode of a code region based on its context, with a Mix-like language becoming the intermediate language.

Type Tailoring [19] is an approach to provide more information to a host type system than it might be capable of by itself. In particular, the authors use the host platform's metaprogramming functionality to refine the types of calls based on the program syntax alone, as well as improve error messages by incorporating surface syntax. Their experiments are based in Typed Racket, that fully expands syntax before checking it. Since Typed Clojure recently changed to interleave macroexpansion and type checking, we could extend this technique to also refine calls based on the types of their arguments (like SoundX).

Other work is relevant to our investigations of improving the user experience of Typed Clojure. SweetT [38] automatically infers type rules for syntactic sugar. Helium [22] provides hooks into the type inference process for domain-specific type error messages.

5 Research Plan and Timeline

I have already made progress towards my thesis:

- I have formalized Typed Clojure, including its characteristic features like hash-maps, multimethods, and Java interoperability, and prove the model type sound.

- I have conducted an empirical study of real-world Typed Clojure usage in over 19,000 lines of code, showing its features correspond to actual usage patterns.
- I have implemented and publicly released a tool that generates Typed Clojure and clojure.spec annotations, and started on a formalism.
- I have started an empirical study the of manual changes needed for the generated annotations to pass type checking is in progress.
- I have successfully run my clojure.spec annotation tool on several hundred open-source projects that will be used to drive further studies.
- I have prototyped an extensible typing rule system and symbolic execution for Typed Clojure.

To complete my thesis, I plan to follow this timeline:

- [**June-July 2018 - Completed**] Fix spec generation.
  - devise and implement a one or more strategies to handle clojure.spec's heterogeneous map spec (and intelligently register global spec aliases)
  - test out specs generation on candidate projects and improve the tool to fix obvious defects
- [**July-August 2018 - Completed**] Proof-of-concept extensible typing rule system
  - convert core.typed to control macroexpansion, rather than type-checking fully-expanded code.
  - prototype interface for defining custom type rules for usages of top-level functions and macros.
- [**August 2018**] Finish formal model of automatic annotation tool.
  - update formal model for annotation tool to include latest optimizations and fixes in recursive type reconstruction.
- [**September-October 2018**] Devise and carry out automatic annotation experiments
  - complete study that quantifies the changes needed to go from automatically annotated types to checked code.
- [**October-November 2018**] Write and submit automatic annotation paper (PLDI submission)
- [**November-December 2019**] Improve extensible typing rule system
  - support more core Clojure idioms using typing rules.
  - revisit study quantifying manual type annotations using enhanced local inference
- [**January-May 2019**] Write dissertation
- [**June 2019**] Defend

5.1 Publications

I plan to publish the following papers:

- **ESOP 2016 - Published** *Practical Optional Types for Clojure* [5]. A paper that provides a formal model of Typed Clojure and presents an empirical study of usage patterns.
- **Fall 2018** *Squash the work: Automatic annotations for Typed Clojure and clojure.spec* A paper that presents a tool capable of generating Typed Clojure and clojure.spec annotations based on runtime observations of the program. We present a formal model of the tool, as well as several manual experiments in how accurate our annotations are. We conduct a larger study by sourcing several hundred projects and automatically generating types and specs, then using these annotations to answer various questions about how clojure.spec and Clojure is used.

References

[1] Beatrice Akerblom et al. "Tracing Dynamic Features in Python Programs". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 292–295. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597103. URL: http://doi.acm.org/10.1145/2597073.2597103.

[2] Esteban Allende et al. "Gradual typing for Smalltalk". In: *Science of Computer Programming* 96 (2014), pp. 52–69.

[3] Jong-hoon (David) An et al. "Dynamic Inference of Static Types for Ruby". In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 459–472. ISSN: 0362-1340. DOI: 10.1145/1925844.1926437. URL: http://doi.acm.org/10.1145/1925844.1926437.

[4] Esben Andreasen et al. "Trace Typing: An Approach for Evaluating Retrofitted Type Systems". In: *ECOOP*. 2016.

[5] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. "Practical Optional Types for Clojure". In: *European Symposium on Programming Languages and Systems*. Springer. 2016, pp. 68–94.

[6] Oscar Callaú et al. "How (and why) developers use the dynamic features of programming languages: the case of smalltalk". In: *Empirical Software Engineering* 18.6 (2013), pp. 1156–1194. ISSN: 1573-7616. DOI: 10.1007/s10664-012-9203-2. URL: https://doi.org/10.1007/s10664-012-9203-2.

[7] Luca Cardelli and John C. Mitchell. "Operations on records". In: *Mathematical Structures in Computer Science*. 1991, pp. 3–48.

[8] G. Castagna et al. "Polymorphic Functions with Set-Theoretic Types. Part 2: Local Type Inference and Type Reconstruction". In: *POPL '15, 42nd ACM Symposium on Principles of Programming Languages*. 2015, pp. 289–302.

[9] Craig Chambers. "Object-Oriented Multi-Methods in Cecil". In: *Proc. ECOOP*. 1992.

[10] Craig Chambers and Gary T. Leavens. "Typechecking and Modules for Multi-methods". In: *Proc. OOPSLA*. 1994.

[11] Stephen Chang, Alex Knauth, and Ben Greenman. "Type Systems As Macros". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 694–705. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009886. URL: http://doi.acm.org/10.1145/3009837.3009886.

[12] Ravi Chugh, David Herman, and Ranjit Jhala. "Dependent Types for JavaScript". In: *Proc. OOPSLA*. 2012.

[13] Marc CircleCI; O'Morain. *Why we're no longer using core.typed*. 2015. URL: http://blog.circleci.com/why-were-no-longer-using-core-typed/.

[14] Sebastian Erdweg et al. "SugarJ: Library-based Syntactic Language Extensibility". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 391–406. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048099. URL: http://doi.acm.org/10.1145/2048066.2048099.

[15] Michael D. Ernst et al. *The Daikon system for dynamic detection of likely invariants*. 2006.

[16] Facebook. *Flow Language Specification*. Tech. rep. Facebook, 2015.

[17] Facebook. *Hack Language Specification*. Tech. rep. Facebook, 2014.

[18] David Fisher. "Static analysis for syntax objects". In: *In ACM SIGPLAN International Conference on Functional Programming*. 2006.

[19] Ben Greenman, Stephen Chang, and Matthias Felleisen. "Type Tailoring (Unpublished manuscript)". In: (). URL: http://www.ccs.neu.edu/home/types/resources/type-tailoring.pdf.

[20] Sudheendra Hangal and Monica S Lam. "Tracking down software bugs using automatic anomaly detection". In: *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE. 2002, pp. 291–301.

[21] Robert Harper and Benjamin Pierce. "A Record Calculus Based on Symmetric Concatenation". In: *Proc. POPL*. 1991.

[22] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. "Scripting the Type Inference Process". In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP '03. Uppsala, Sweden: ACM, 2003, pp. 3–13. ISBN: 1-58113-756-7. DOI: 10.1145/944705.944707. URL: http://doi.acm.org/10.1145/944705.944707.

[23] Haruo Hosoya and Benjamin C Pierce. "How Good is Local Type Inference?" In: *Technical Reports (CIS)* (1999), p. 180.

[24] Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. "Mixing Type Checking and Symbolic Execution". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. Toronto, Ontario, Canada: ACM, 2010, pp. 436–447. ISBN: 978-1-4503-0019-3. DOI: `10.1145/1806596.1806645`. URL: `http://doi.acm.org/10.1145/1806596.1806645`.

[25] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: `10.1145/360248.360252`. URL: `http://doi.acm.org/10.1145/360248.360252`.

[26] Erik Krogh Kristensen and Anders Møller. "Inference and evolution of typescript declaration files". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2017, pp. 99–115.

[27] Jukka Lehtosalo. *mypy*. URL: `http://mypy-lang.org/`.

[28] Benjamin S. Lerner et al. "TeJaS: Retrofitting Type Systems for JavaScript". In: *Proceedings of the 9th Symposium on Dynamic Languages*. DLS '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 1–16. ISBN: 978-1-4503-2433-5. DOI: `10.1145/2508168.2508170`. URL: `http://doi.acm.org/10.1145/2508168.2508170`.

[29] Florian Lorenzen and Sebastian Erdweg. "Sound Type-dependent Syntactic Language Extension". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: ACM, 2016, pp. 204–216. ISBN: 978-1-4503-3549-2. DOI: `10.1145/2837614.2837644`. URL: `http://doi.acm.org/10.1145/2837614.2837644`.

[30] John M. Lucassen and David K. Gifford. "Polymorphic effect systems". In: *Proc. POPL*. 1988.

[31] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. "Typed Lua: An Optional Type System for Lua". In: *Proc. Dyla*. 2014.

[32] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. "Type Regression Testing to Detect Breaking Changes in Node.js Libraries". In: *Proc. 32nd European Conference on Object-Oriented Programming (ECOOP)*. 2018.

[33] Microsoft. *Typescript Language Specification*. Tech. rep. Version 1.4. Microsoft, 2014.

[34] Todd Millstein and Craig Chambers. "Modular Statically Typed Multimethods". In: *Information and Computation*. Springer-Verlag, 2002, pp. 279–303.

[35] Martin Odersky et al. *An overview of the Scala programming language (second edition)*. Tech. rep. EPFL Lausanne, Switzerland, 2006.

[36] Cyrus Omar et al. "Safely composable type-specific languages". In: *European Conference on Object-Oriented Programming*. Springer. 2014, pp. 105–130.

[37] Benjamin C. Pierce and David N. Turner. "Local Type Inference". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. San Diego, California, USA: ACM, 1998, pp. 252–265. ISBN: 0-89791-979-3. DOI: `10.1145/268946.268967`. URL: `http://doi.acm.org/10.1145/268946.268967`.

[38] Justin Pombrio and Shriram Krishnamurthi. "Inferring type rules for syntactic sugar". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2018, pp. 812–825.

[39] Michael Pradel, Parker Schuh, and Koushik Sen. "TypeDevil: Dynamic type inconsistency analysis for JavaScript". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 314–324.

[40] Brock Pytlik et al. "Automated fault localization using potential invariants". In: *arXiv preprint cs/0310040* (2003).

[41] Claudiu Saftoiu. *JSTrace: Run-time type discovery for JavaScript*. Tech. rep. Technical Report CS-10-05, Brown University, 2010.

[42] Sam Tobin-Hochstadt and Matthias Felleisen. "Logical Types for Untyped Languages". In: *Proc. ICFP*. ICFP '10. 2010.

[43] Sam Tobin-Hochstadt and Matthias Felleisen. "The Design and Implementation of Typed Scheme". In: *Proc. POPL*. 2008.

[44] Michael M. Vitousek et al. "Design and Evaluation of Gradual Typing for Python". In: *Proc. DLS*. 2014.

[45] Mitchell Wand. *Type Inference for Record Concatenation and Multiple Inheritance*. 1989.