A Practical Optional Type System for Clojure

Ambrose Bonnaire-Sergeant

Supervised by Rowan Davies

This report is submitted as partial fulfilment
of the requirements for the Honours Programme of the
School of Computer Science and Software Engineering,
The University of Western Australia,
2012

Abstract

Dynamic programming languages often abandon the advantages of static type checking in favour of their characteristic convenience and flexibility. Static type checking eliminates many common user errors at compile-time that are otherwise unnoticed, or are caught later in languages without static type checking. A recent trend is to aim to combine the advantages of both kinds of languages by adding *optional* static type systems to languages without static type checking, while preserving the idioms and style of the language.

This dissertation describes my work on designing an optional static type system for the Clojure programming language, a dynamically typed dialect of Lisp, based on the lessons learnt from several projects, primarily Typed Racket. This work includes designing and building a type checker for Clojure running on the Java Virtual Machine. Several experiments are conducted using this prototype, particularly involving existing Clojure code that is sufficiently complicated that type checking increases confidence that the code is correct. For example, nearly all of *algo.monads*, a Clojure Contrib library for monadic programming, is able to be type checked. Most monad, monad transformer, and monadic function definitions can be type checked, usually by adding type annotations in natural places like function definitions.

There is significant future work to fully type check all Clojure features and idioms. For example, multimethod definitions and functions with particular constraints on the number of variable arguments they accept (particularly functions taking only an even number of variable arguments) are troublesome. Also, there are desirable features from the Typed Racket project that are missing, such as automatic runtime contract generation and a sophisticated blame system, both which are designed to improve error messages when mixing typed and untyped code in similar systems.

Overall, the work described in this dissertation leads to the conclusion that it appears to be both practical and useful to design and implement an optional static type system for the Clojure programming language.

Keywords: Programming languages, optional type systems, Clojure

Acknowledgements

The past year has included by far my most enjoyable and motivating programming-related experiences.

Last November I attended my first programming conference, Clojure Conj 2011 (Raleigh, NC), where I delivered a talk on logic programming with Clojure and met some of my programming heroes. A week later my supervisor Rowan Davies invited me to visit Carnegie Mellon University where he worked on his PhD project. Most memorably, I was invited to a meeting between Rowan and his PhD supervisor Frank Pfenning.

This trip was funded by donations from the programming community. Thank you to all who helped me get there: this work was directly inspired from conversations during this period.

Many thanks go to my supervisor Rowan. His influence as a lecturer inspired my exploration of functional programming languages. He has exceeded my expectations as an honours supervisor, and we plan to continue to collaborate in this area. Rowan's experience with refinement types brought a unique perspective to this project which was very welcome.

I am grateful to Sam Tobin-Hochstadt for his excellent work in this area of research. He has always been happy to offer assistance and advice throughout this project, which has proved invaluable.

I would also like to thank David Nolen for mentoring this project as part of Google Summer of Code 2012, and Chas Emerick, without whom I would have never even considered attending Clojure Conj 2011 (and this work probably would not have happened). Also, Daniel Spiewak, Dan Friedman and William Byrd, Jim Duey, Baishampayan Ghose, Sam Aaron, and Steve Strickland, thank you for your help and inspiration. Thank you Rich Hickey for creating Clojure, it is a thrill to build tools that make this great language even better.

Fittingly, I gave a talk on Typed Clojure at Clojure Conj 2012, which was extremely satisfying.

Finally, thanks to my family, friends and to my girlfriend Tamara, each for your love and support.

Contents

CHAPTER 1

Introduction

1.1 Thesis

It is practical and useful to design and implement an optional typing system for the Clojure programming language using bidirectional checking that allows Clojure programmers to continue using idioms and style found in current Clojure code.

1.2 Motivation

In the last decade it has become increasingly common to enhance dynamically typed languages with static type systems. This idea is not new (see work by Fagan, Cartwright, Aiken and Murphy [Fag91; CF91; AM91]), but recent attempts are noteworthy for their broad success in matching many of the advantages of statically typed languages (see work by Tobin-Hochstadt [TH10]), notably due to the use of bidirectional checking (like work by Pierce and Turner [PT00]). Instead of always attempting to infer types, this algorithm relies on programmer annotations appearing in some natural places such as giving the type of each top-level function.

The Clojure programming language is a dynamically typed dialect of Lisp invented by Hickey [Hic08], designed to run on popular platforms. It emphasises functional programming with immutable data structures and provides direct interoperability with its host platform. Notable implementations of Clojure exist for the Java Virtual Machine (JVM), the Common Language Runtime, and for Javascript virtual machines. At the current time, Clojure on the JVM is the most mature implementation, and therefore this project focuses on the JVM implementation.

Clojure has attracted wide-spread users in part by concentrating on pragmatism. Performance is a key feature, for example the JVM implementation of Clojure offers ways to access Java-like speed for certain operations. Also,

Clojure's extensive host interoperability offers Clojure programmers access to existing libraries for their platform, such as the vast Java library ecosystem. By coupling pragmatic necessities with elegant features like Lisp-style macros, functional programming, and immutability by default, Clojure is a compelling general purpose programming language.

Recently a number of languages have been created or modified to support aspects of both static and dynamic typing. Dart [Goo12a] (created by Google) is dynamically typed but offers a simple form of optional static typing that specifically do not affect runtime semantics. Typescript [Mic12] (created by Microsoft) adds an optional type system to Javascript, a well-known dynamic language. Typed Racket [THF08; TH10] (by Tobin-Hochstadt et al.) goes further by offering safe interoperability between typed and untyped modules by generating appropriate runtime assertions based on expected static types.

When a static type checker is not available, which describes the situation for most dynamic languages, other techniques are used for checking type invariants. For example, "design by contract" is often used, introduced by Meyer for the Eiffel language [Mey92], in which the programmer defines contracts that are enforced at runtime. Unit testing is also a popular verification technique in dynamic languages. Clojure adopts these approaches, providing easy syntax for defining Eiffel-inspired pre- and post-conditions and a library for writing unit tests.

Static type systems, however, are still desirable for many situations. Powerful type systems like ML's [Mil+97] and Haskell's [Sim10] have proved particularly useful when complicated programming styles are required. For example, Haskell's advanced static type system helps the programmer write correct monadic code (as detailed by Wadler [Wad95]) especially in more complicated situations like combining monads via monad transformers.

1.2.1 Why implement an optional type system for Clojure?

Better error messages

Dynamically typed languages like Clojure often sacrifice some of the advantages of static typing. In particular, statically typed languages generally offer compile-time type errors where dynamically typed languages either give runtime type errors, or none at all.

This is distinction is easily demonstrated by comparing Clojure and Haskell's behaviour at their interactive prompts. Listing 1.1 shows how Clojure happily compiles a nonsensical function that performs addition on a string. It is only until we invoke it that we get a type error.

Listing 1.1: Runtime type errors in Clojure

```
user=> (fn [x] (+ "string" x))
#<user$eval128376$fn__128377 user$eval128376$fn__128377@183ab4e>

user=> ((fn [x] (+ "string" x)) 1)
#<ClassCastException java.lang.ClassCastException: java.lang.String
    cannot be cast to java.lang.Number>
```

The error provided by Clojure is unsatisfying:

- The error message does little to help the programmer identify the source of the error.
- It requires at least some knowledge of Clojure's internals to debug.
- We would rather receive the error at the earliest time possible: compile-time.

Listing 1.2 shows how Haskell, a statically typed language, handles this interaction. Haskell detects our programming error at compile-time, disallowing our function to compile.

Listing 1.2: Compile-time type errors in Haskell

```
Prelude> (\x -> "string" + x)

<interactive>:1:17:
   No instance for (Num [Char])
   arising from a use of ‘+’
   Possible fix: add an instance declaration for (Num [Char])
   In the expression: "string" + x
   In the expression: (\ x -> "string" + x)
   In an equation for ‘it’: it = (\ x -> "string" + x)
```

Haskell's error here is much preferred over Clojure's:

- The exact source of the error is given in *user code*.
- The error is given as early as possible: compile-time.

By adding static type checking to Clojure, we can achieve errors messages that are closer to Haskell's. Listing 1.3 gives an idea of what kind of error messages Typed Clojure actually provides in these situations. However, it deserves some explanation:

- The given source of the error may be surprising, but it is explained by noting that Clojure "inlines" `(+ "string" a)` to a Java method call `(. clojure.lang.Numbers add "string" a)`.
- `add`'s type has been overridden by Typed Clojure to be more specific (see Section 1.4.9).
- `cf` type checks a form.
- `(fn> [[a :- Number]] ...)` defines a function with its first argument of static type `Number`.

Listing 1.3: Compile-time errors in Typed Clojure

```
user=> (cf (fn> [[a :- Number]] (+ "string" a)))
#<Exception java.lang.Exception: 5:
Static method clojure.lang.Numbers/add could not be applied to
   arguments:
Domains:
  typed.core/AnyInteger typed.core/AnyInteger
  java.lang.Number java.lang.Number

Arguments:
  (Value "string") java.lang.Number

in: (. clojure.lang.Numbers add "string" a)>
```

Typed Clojure lists each of the expected domain types, of which the static method `add` of the Java class `clojure.lang.Numbers` has been assigned two, and also lists the types of the given arguments. A line number and the source of the error is also provided.

Most importantly: the error was caught at *compile-time*.

Higher order programming

The initial motivation for implementing an optional type system for Clojure was outlined in a discussion with Jim Duey, a Clojure programmer, at Clojure Conj 2011. In an apparently heroic effort, Duey managed to implement a Clojure library for conduits [1], an advanced form of "pipes", using arrows, a generalisation of monads. Conduits also significantly surpass monads in complexity and are usually reserved for languages with advanced type systems like Haskell.

[1] `http://www.yesodweb.com/book/conduits`

Duey highlighted a strong desire for a type system for several reasons. Firstly, to verify the correctness of the library. Without a static type system, it is a significant task to verify such an implementation as correct due to heavy use of higher-order functions. Secondly, to aid him while writing the library.

Programmers in statically typed languages take advantage of the static type system to help verify code written in abstract programming styles, like monads. Adding a static type system to Clojure should bring similar benefits to Clojure programmers.

1.2.2 Why does Clojure not already feature static type checking?

Given the advantages of static type checking outlined in the previous section, it is natural to ask: why does Clojure not already support static type checking? The simplest answer is that Clojure was not designed to support a static type system. Instead, features and idioms that are not necessarily compatible with traditional type systems were favoured, like variable-arity functions, untyped hash-maps used like objects, and dynamic extension of program behaviour.

Without specifically speaking for the designers of Clojure, at the time Clojure was designed it was not clear how to reconcile Clojure's feature-set with static type checking in a satisfying way. Since then, Typed Racket, which add static type checking to the Racket programming language, has matured and gained acceptance. Once this occurred, it seemed only a matter of time until the ideas developed for the Typed Racket project were tested for their applicability to Clojure. As far as we know, this project is the first serious attempt to build an optional static type system for Clojure.

1.2.3 What kind of type system does Typed Clojure provide?

There are many concepts associated with *types* and *type systems* in both the literature and informal discourse. A programmer who uses dynamically-typed languages may have a drastically different notion of what a type is than, say, a programmer preferring languages with advanced static type systems. There is some debate as to whether optional static type systems like Typed Clojure can even be called a type system. We choose to follow the terminology of Pfenning [Pfe08] and Reynolds [Rey02], where such optional type systems are *extrinsic* type systems, and more traditional type systems are *intrinsic* type systems. This distinction has a long history, originating in work in the λ-calculus by Church [Chu40] and Curry [Cur34], leading to intrinsic types and extrinsic types also being called Church types and Curry types often in the literature.

An ordinary static type system is used to check whether programs are basically meaningful. Pfenning and Reynolds call these type systems *intrinsic*. A language with an intrinsic static type system has a run-time semantics that depends on the types associated with variables and expressions during type checking. For example, C, Java, ML, and Haskell have intrinsic types. This means programs written in these languages must pass the type checker before being run.

A static type system is *extrinsic* when runtime semantics does not depend on a static type system. In other words, passing a static type checker is not essential to running programs. A dynamically typed language can be viewed as having a trivial static type system that supports exactly one type, a view advocated by Harper [Har12] and common in the literature on static types (for example, Pierce [Pie02]).

1.2.4 What is the community reaction to Typed Clojure?

Since I began developing Typed Clojure, the Clojure community has shown significant interest in this work:

- I developed this project as a Google Summer of Code 2012 project, after it was selected by the Clojure community as a Clojure Google Summer of Code project for 2012 [Goo12b].
- I gave a talk on this project at the Clojure Conj 2012 conference, the main international programming conference related to Clojure, in November.

1.3 Contributions

In this dissertation we develop an optional type system for Clojure. We focus on the features needed to make a practical type system that existing Clojure programmers can use. We also develop a prototype based on Typed Racket [TH10].

The contributions of the work described in this dissertation are as follows:

- We develop a prototype type checker for Clojure running on the Java Virtual Machine which is able to type check many common Clojure idioms. We show how this type checker can be used to aid Clojure programmers use sophisticated programming styles by porting most of an existing library for monadic programming.

- This prototype is extended to support several features that are not yet present in Typed Racket. We support filtering sequences on anonymous functions that convey only negative type information (eg. non-nil), capturing a common Clojure idiom. We enable type checking of very abstract code (like code using monads) by adding *type functions* (functions at the type level).
- We support safer interoperability with Java from Clojure. We show how techniques developed for Typed Racket, such as occurrence typing [THF10], help detect misuses of Java's *null* while still conforming to existing Clojure idioms.
- We can type check several particularly useful Clojure idioms, like common usages of the *sequence abstraction* (Clojure's common interface to all collections) and common usages of "untyped" hash-maps, which have a similar role to records in ML or Haskell, and objects in object-oriented languages.
- We identify the main future issues in typing Clojure code. Multimethod definitions are problematic to type check accurately because of the interaction of its dispatch mechanism and occurrence typing. Records have subtle semantics for dissociating keys, specifically: dissociating all base keys of a record returns a plain map.

Together these contributions support my thesis and motivate further work in this area.

1.4 Typed Clojure through Examples

This section introduces Typed Clojure with example code. Typed Clojure is developed for the JVM implementation of Clojure, therefore the rest of this chapter uses that implementation. An attempt is made to introduce some Clojure syntax and semantics to those unfamiliar or needing a refresher. A basic knowledge of Lisp syntax is handy, but a brief tutorial is given for newcomers.

1.4.1 Preliminary: Lisp Syntax

The core of understanding Lisp syntax when coming from a popular language like Java or Javascript can be summarised by these points.

- Operators are always in prefix position.

- Invocations are always wrapped in a pair of balanced parenthesis.
- Parenthesis start to the left of the operator.

For example, the Java expressions *(1 + 2) / 3* is written in Lisp pseudocode `(/ (+ 1 2) 3)` and *numberCrunch(1, 2)* written `(numberCrunch 1 2)`.

Clojure also adds other syntax:

- Prefixing `:` to a symbol defines a *keyword*, often used for map keys. eg. `:my-keyword`.
- Square brackets delimit vector literals. eg. `[1 2]` is a 2 place vector.
- Curly brackets define map literals. eg. `{:a 1 :b 2}` is a map from `:a` to `1` and `:b` to `2`.
- Commas are always optional and treated as whitespace, but often used to show the intended structure.

1.4.2 Simple Examples

We begin with the obligatory *Hello world* example.

Listing 1.4: Typed Hello world

```
(ns typed.test.hello-world
  (:require [typed.core :refer [check-ns]]))

(println "Hello world")
```

At this point, it is worth understanding Clojure's namespacing feature. Clojure code is always executed in a *namespace*, and each file of Clojure code should have a `ns` declaration with the namespace name and its dependencies, which switches the current namespace and executes the given dependency commands. There is one special namespace, `clojure.core` which is loaded with every namespace, implicitly "referring" all its vars in the namespace. For example, `ns` refers to the var `clojure.core/ns`, similarly `println` refers to `clojure.core/println` (vars are global bindings in Clojure).

The example in listing 1.4 declares a dependency to `typed.core`, Typed Clojure's main namespace. It also refers the var `typed.core/check-ns` into scope (`check-ns` is the top level function for type checking a namespace). Other than this dependency, this is identical to the untyped *Hello world*.

More complex code may require extra annotations to type check:

Listing 1.5: Annotating vars in Typed Clojure (adapted from a Typed Scheme/Racket example by Tobin-Hochstadt [TH10])

```
(ns typed.test.collatz
  (:require [typed.core :refer [check-ns ann]]))

(ann collatz [Number -> Number])
(defn collatz [n]
  (cond
    (= 1 n)
      1
    (and (integer? n)
         (even? n))
      (collatz (/ n 2))
    :else
      (collatz (inc (* 3 n)))))
```

(`(defn collatz [n] ...)` expands to `(def collatz (fn [n] ...))`, where `(def name init)` defines a new var in the current namespace called `name`, and `fn` creates a function value).

In this example, we define a new var `collatz` (when unambiguous, I omit the qualifying namespace/package for the remainder of the chapter). Typed Clojure type checks at the namespace granularity. All vars that occur in a "typed" namespace must have a type annotation. Here `typed.core/ann`, a Typed Clojure macro for annotating vars, annotates `collatz` to be a function from `Number` to `Number` (`Number` refers to `java.lang.Number`, due to every Clojure namespace implicitly importing all Classes in the `java.lang` package, the equivalent of Java's `import java.lang.*;`).

1.4.3 Datatypes and Protocols

As well as `def` and `defn` definitions, Clojure programmers typically include datatype and protocol definitions. Protocols are similar to Java interfaces [Ora12] and datatypes are similar to classes implementing interfaces. We can annotate datatype and protocol definitions similarly.

Listing 1.6: Annotating protocols and datatypes in Typed Clojure

```
(ns typed.test.deftype
  (:require
    [typed.core :refer [check-ns ann-datatype defprotocol>
                        ann-protocol AnyInteger]]))

(ann-protocol Age
  age [Age -> AnyInteger])
(defprotocol> Age
  (age [this]))

(ann-datatype Person
  [name :- String
   age :- AnyInteger])
(deftype Person [name age]
  Age
  (age [this] age))

(age (Person. "Lucy" 34))
```

`defprotocol>` defines a new Clojure protocol[2] with a set of methods. It is identical to the usual `defprotocol`, but is required when using Typed Clojure due to its complicated macroexpansion. `ann-protocol` is a Typed Clojure macro for annotating a protocol with the types of its methods. In this example, we define a protocol `Age` with an `age` method, which is really a first-class function taking the target object as the first parameter. The type signature provided with `ann-protocol`, here `[Age -> AnyInteger]` which is the type of the function taking an `Age` and returning an `AnyInteger`, is for this function.

`deftype` defines a new Clojure datatype[3] in the current namespace with a number of fields and methods. `typed.core/ann-datatype` is a Typed Clojure form for annotating datatypes, including its field types. In this example, we create a datatype `typed.test.person.Person` (datatype defintions generate a Java Class, where the current namespace is used as a starting point for its qualifying package) with fields `name` and `age` and extend the `Age` protocol by implementing the `age` method.

Java constructors are invoked in Clojure by suffixing the Class we want to instantiate with a dot. Datatypes are implemented as Java Classes with immutable fields (by default) and a single constructor, taking as arguments its fields in the order they are passed to `deftype (Person. "Lucy" 34)` constructs

[2]See http://clojure.org/protocols for a full description of protocols.

[3]See http://clojure.org/datatypes for a full description of datatypes.

a new `Person` instance, setting the fields to their corresponding positional arguments. Typed Clojure checks the datatype constructor to be the equivalent of `[String AnyInteger -> Person]`.

Finally, Typed Clojure checks invocations of Protocol methods. It infers `Person` is an instance of `Age` from the datatype definition, therefore `(age (Person. "Lucy" 34))` is type-safe. Since generally we only need to annotate definitions and not uses, the number of annotations is relatively small.

1.4.4 Polymorphism

Typed Clojure supports F-bounded polymorphism, first introduced by Canning, Cook, Hill and Olthoff [Can+89]. F-bounded polymorphism is an extension of bounded polymorphism, where polymorphic type variables can be restricted by *bounds*. In particular, F-bounded polymorphism allows type variable bounds to recursively refer to the variable being bounded. Typed Clojure supports upper and lower type variable bounds. This would be required mainly for future work involving Java Generics, like typing invocations of *java.util.Collection/max*[4].

For accurate type checking, Typed Clojure parameterises some of Clojure's data structures. For example, the interface behind Clojure's *seq* abstraction for sequences `clojure.lang.Seqable` has one covariant parameter[5].

Listing 1.7: Polymorphism in Typed Clojure

```
...
(ann to-set
     (All [x]
       [(U nil (Seqable x)) -> (clojure.lang.PersistentHashSet x)]))
(defn to-set [a]
  (set a))
...
```

In this example[6], we define `to-set`, aliasing `clojure.core/set`. `All` introduces a set of type variables to the body of a type, here `x` is used to define a relationship between the input type and return type.

`(U nil (Seqable x))` is a common type in Typed Clojure, read as the union of the type `nil` and the type `(Seqable x)`. The vast majority of types for collection

[4] `http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Collections.html#max(java.util.Collection)`

[5] Covariant means that `(Seqable Integer)` is a subtype of `(Seqable Number)` because Integer is a subtype of Number.

[6] When convenient, namespace declarations are omitted for the remainder of the chapter.

processing functions in the Clojure core library feature it as an input type, where passing `nil` either has some special behaviour or is synonymous with passing an empty `Seqable`. The ability to specify types that explicitly include or exclude `nil` is one of the strengths of Typed Clojure, and an aspect where it is more expressive than standard type systems like that for Java.

1.4.5 Singleton Types

Following Typed Racket, singleton types for certain values are provided in Typed Clojure. A singleton type is a type with a single member, like `1`, `:a`, or `"a"`[7]. Typed Clojure provides syntax for singleton types, either by passing the value literal to the `Value` primitive, or by prefixing a quote (').

Listing 1.8: Singleton Types

```
(ann k ':my-keyword)
(def k :my-keyword)
```

Listing 1.8 shows a simple example of using singleton types in Typed Clojure. Singleton types are discussed further in section 2.5.

1.4.6 Heterogeneous Maps

It is idiomatic in Clojure to use plain maps with known keyword keys as lightweight objects. Clojure provides a hash-map literal (using curly braces), making this idiom particularly convenient. For example, `{:a 1, :b 2}` is a map value with two key-value entries: from keyword key `:a` to value `1`, and keyword key `:b` to value `2`. Note that commas are always whitespace in Clojure and are included occasionally for readability.

Typed Clojure provides heterogeneous map types which captures this common "maps as objects" pattern. A heterogeneous map type has only *positive* information on the types of key-value entries. In other words, it conveys whether a particular key is present, but not whether it is absent. The implications of this is discussed in section 3.7. Heterogeneous maps only support keys that are singleton Keyword types. This restriction is reflected in the syntax for defining heterogeneous map types.

[7]Strings are delimited by " in Clojure

Listing 1.9: Heterogeneous map types in Typed Clojure

```
(ann config '{:file String, :ns Symbol}))
(def config
  {:file "clojure/core.clj",
   :ns 'clojure.core})
```

This example checks `config` to be a heterogeneous map with `:file` and `:ns` keys, with values of type `String` and `Symbol` respectively.

Heterogeneous vector and seq types are also provided and work similarly, except for their lack of keys like maps, hence are just a sequence of types.

1.4.7 Variable-Arity Functions

Functions in Clojure are multi-arity (a very commonly used feature) which means a function can be defined with several sets of function arguments and bodies (or *arities*) and which arity is executed depends on the number of arguments passed to the function. A function can have any number of arities with fixed parameters, and at most one arity with variable-parameters. Each arity must have a unique number of parameters and have a lower number of parameters than the "variable arity", if present.

Strickland, Tobin-Hochstadt, and Felleisen invented a calculus and corresponding implementation in Typed Racket for variable-arity polymorphism [STHF09] that is sufficient to handle uniform and non-uniform variable-arity functions. Typed Clojure includes an implementation of the most immediately useful parts of variable-arity polymorphism using algorithms, nomenclature, and implementation based on this work.

Uniform Variable-Arity Functions

A function with uniform variable parameters can treat its variable parameter as a homogeneous list. Strickland et al. attaches a *starred pre-type* `T *` to the right of the fixed arguments in a function type, where T is some type, and we take an identical approach. For example, `+` in Clojure accepts any number of arguments of type `Number`, represented by the type `[Number * -> Number]`.

Listing 1.10: Typing multi-arity functions

```
(ann not= (Fn [Any -> boolean]
              [Any Any -> boolean]
              [Any Any Any * -> boolean]))
(defn not=
  "Same as (not (= obj1 obj2))"
  ([x] false)
  ([x y] (not (= x y)))
  ([x y & more]
   (not (apply = x y more))))
```

It is common to find Clojure library functions that define seemingly redundant function arities for performance reasons. Listing 1.10 defines the multi-arity function `not=`, taken from the Clojure standard library `clojure.core` that uses this pattern. `not=` has three arities in its definition, including one that takes a variable number of arguments. The `Fn` type constructor builds an *ordered function intersection type* from function types. Each arity must have at least one matching function type associated with it. The two "fixed arities" are given familiar function types, with one and two fixed parameters respectively. The type given for the arity with variable arguments `[Any Any Any * -> boolean]` uses a starred pre-type to signify any number of arguments of type `Any` can be provided to the right of its fixed arguments.

Non-uniform Variable-Arity Functions

Where *uniform* variable-arity function types use *starred pre-types*, *non-uniform* variable-arity function types use *dotted pre-types*. Typed Clojure supports usages of *non-uniform* variable-arity functions, where the variable parameter is a heterogeneous list, represented by a *dotted type variable*.

For example, the variable argument function `clojure.core/map` takes a function and one or more sequences, and returns a sequence of the results of applying the function to each element of the sequences simultaneously. Its type is given in listing 1.11.

Listing 1.11: Type signature for `clojure.core/map`

```
(ann clojure.core/map
     (All [c a b ...]
       [[a b ... b -> c] (U nil (Seqable a)) (U nil (Seqable b)) ... b
          -> (LazySeq c)]))
```

By adding `...` after the last type variable in an `All` binder we can introduce a

dotted type variable into scope, which is a placeholder for a sequence of types. A *dotted pre-type* `T ... b` over *base* `T` (a type) and *bound* `b` (a dotted type variable) serves as a placeholder for this sequence of types. Dotted pre-types must appear to the right of all fixed parameters in a function type, and cannot be mixed with other kinds of variable parameters like starred pre-types. When a sequence of types of length n is associated with a dotted pre-type, the dotted pre-type is expanded to n copies of `T`. One other special property of a dotted pre-type is that the bound `b` is *in scope* as a *normal* type variable in its base `T`.

To demonstrate dotted pre-types, we use `typed.core/inst` to instantiate `map`. `inst` takes a polymorphic expression and a number of types that are satisfactory for instantiating the polymorphic type and returns an expression of the instantiated polymorphic type.

The instantiation

```
(inst map Number boolean String)
```

returns an expression of type

```
[[boolean String -> Number] (U (Seqable boolean) nil)
  (U (Seqable String) nil) -> (LazySeq Number)]
```

However, if sufficient types are given, the instantiation for `map` can be inferred. The invocation

```
(map (ann-form (fn [a b c] c)
               [boolean String Number -> Number])
     [true false] ["mystr" "astr"] [1 4])
```

uses `typed.core/ann-form` to assign an expected type to the first argument, which is sufficient to infer the result type

```
(LazySeq Number)
```

1.4.8 Occurrence Typing

It is common in Clojure, like other dynamically typed languages, to encode implicit type invariants in conditional tests. In listing 1.12, conditional tests are used to refine the types of the bindings `a` and `b`. Occurrence typing [THF10] is a technique useful for capturing these kinds of type invariants (occurrence typing is discussed in further detail in Section 2.11).

Listing 1.12: Example of occurrence typing in Typed Clojure

```
(ann num-vec2
     [(U nil Number) (U nil Number) -> (Vector* Number Number)])
(defn num-vec2 [x y]
  [(if x x 0)
   (if y y 0)])
```

To check Listing 1.12, occurrence typing infers type information based on the results of conditional tests. A conditional in Clojure is written `(if test then else)`, where `then` is executed if `test` is a true value, or `else` is executed if `test` is a false value. In Clojure, `nil` and `false` are the only false values: all other values are true values. `(if x x 0)` in Listing 1.12 chooses a branch based on the truth value of the test expression `x`. Before this expression, the binding `x` is known to be of type `(U nil Number)`. There are two cases to consider when checking this conditional: when the test `x` is a true value, and when it is a false value. If `x` is a true value, then the binding `x` must be of type `Number`. If `x` is a false value, then the binding `x` must be of type `nil`. Occurrence typing understands these invariants and applies them when checking each branch: occurrences of `x` in the first branch are considered to be of type `Number`, and occurrences of `x` in the second branch are considered to be of type `nil`. This infers `[(if x x 0) (if y y 0)])` to be of type `(Vector* Number Number)` (the type for a two-place heterogeneous vector of `Number`s).

1.4.9 Java Interoperability

Typed Clojure supports integration with Java's type system during interoperability from Clojure. This offers many of the same advantages as using Java's type system from Java, for example it is a type error to pass arguments of the wrong type to methods. Typed Clojure also makes different decisions to Java's type system, particularly in the treatment of Java's *null*. Note that these decisions do not change Java's type system, but only how Typed Clojure handles Java interoperability compared to how someone familiar with Java might expect.

From the perspective of static types in Java, *null* is included in all reference types. *null* is represented in Clojure by the value `nil`. Unlike Java's type system Typed Clojure explicitly separates *null* and reference types giving Typed Clojure more accurate types: we can directly express *nullable* and *not nullable* types. (A type is *nullable* if it may also be *null*, which is expressed in Typed Clojure by creating a union of the type and `nil`. A type is *not nullable* if it doesn't include *null*).

Listing 1.13: Java interoperability with Typed Clojure

```
(ns typed.test.interop
  (:import (java.io File))
  (:require [typed.core :refer [ann non-nil-return check-ns]]))

(ann f File)
(def f (File. "a"))

(ann prt (U nil String))
(def prt (.getParent ^File f))

(non-nil-return java.io.File/getName :all)
(ann nme String)
(def nme (.getName ^File f))
```

Listing 1.13 shows how Typed Clojure handles *null* while creating and using an instance of *java.io.File*.

Typed Clojure checks calls to Java constructors by requiring the provided arguments be acceptable input to at least one constructor for that Class. In this case, *java.io.File* has a constructor accepting a *java.lang.String* argument, so `(File. "a")` is type safe. Java constructors never return *null*, so Typed Clojure assigns the return type to be `File`. This constructor is equivalent to `[String -> File]` in Typed Clojure.

Next, we see how Typed Clojure's default behaviour treats method return positions as nullable. By default, the *java.io.File* instance method *getParent* is assigned the type `[-> (U nil String)]` in Typed Clojure. This happens to be a valid approximation of the method as *getParent* returns *null* "if the pathname does not name a parent directory"[8]. On the other hand, the *java.io.File* instance method *getName* always returns an instance of *java.lang.String*, so we set the return position of *getName* to non-nil with `typed.core/non-nil-return` (the second parameter to `non-nil-return` specifies which arities to assume non-nil returns, accepting either a set of parameter counts of the relevant arities, or `:all` to override all arities of that method).

Such annotations are in a sense assumptions; should they turn out to be wrong, Typed Clojure will infer incorrect types. This is somewhat troubling, but type safety can be restored by using other techniques. For example, Typed Racket generates runtime contracts based on static types to ensure that type

[8]See Javadoc for *java.io.File*: http://docs.oracle.com/javase/1.4.2/docs/api/java/io/File.html

errors are caught at runtime, while giving good error messages which indicate (or "blame" [WF09]) the original source of the error. A similar system for verifying that method annotations like `non-nil-return` are correct is planned as future work (see section 6.2).

CHAPTER 2

Literature Review

In this chapter we fit the design of Typed Clojure with literature from related fields. Forward-references are often given to link the discussed literature with this later parts of this dissertation where there is some relevance or influence.

2.1 Dynamic Typing in Typed Languages

Typed Clojure adds static type checking to an existing untyped language. Coming from the other direction, Abadi, Cardelli, Pierce and Rémy [Aba+95] extend a static language to be more dynamic. This work centres around adding a type *Dynamic*, which represents a dynamic type in a statically typed language.

Rossberg [Ros07] describes the advantages of having a type *Dynamic* over other approaches to making statically typed languages more like dynamic ones and surveys work in this area. He says that type *Dynamic* allows aspects of dynamic style to be represented in statically typed languages without compromising the type system. However, he claims type *Dynamic* is rarely used in practical languages because it is too inconvenient in real programs. He claims to have fixed these issues, but a practical language has yet to integrate his changes.

2.2 Static Typing for Untyped Languages

This section briefly reviews work on designing static type systems for untyped languages.

2.2.1 Soft Typing

Soft typing [CF91] is an approach for ensuring type safety in untyped languages. A soft type system infers types for programs, distinguishing between degrees of

potential type safety. A soft type checker uses this information to preserve type safety by inserting appropriate checks and informs the programmer of potential inconsistencies. For example, if the soft type system detects a portion of code is sometimes not type safe, the soft type checker inserts a check that throws a runtime error upon unsafe usages. Thus, soft type systems differ from traditional type systems in that type inference never fails and an inconsistency always results in a runtime check.

Wright and Cartwright [WC97] developed Soft Scheme, a soft type system for Scheme. It extended earlier work by Cartwright and Fagan [CF91] and Fagan [Fag91], adding support for practical features such as first-class continuations and variable-arity functions. Soft Scheme does not require any extra type annotations.

Type systems for Scheme have since moved away from soft typing. Alas, the only reasonably complete account of this transition appears to be slides from a talk by Felleisen [Fel09], a leader in this area of research for over 20 years. Felleisen comments that while Soft Scheme discovered type problems, it suffers from incomprehensible error reporting that required PhD-level expertise to decipher.

2.2.2 Program Analysis

Scheme then moved to program analysis techniques like Set-Based Analysis [FF97] (SBA). They, however, suffered from modularity issues due to the lack of declared intended types. MrSpidey[1] is a static type checker for Racket that uses uses SBA. Felleisen states the main problem with these kinds of systems: "Mr Spidey and friends infer brittle and large types; errors remain difficult to explain and fix" [Fel09].

2.2.3 Gradual Typing

Gradual typing combines static and runtime type checking, so programmers can choose the most appropriate one for the situation. A key feature of gradual type systems is its use of *bidirectional checking* like [PT00], which requires type annotations in natural places like functions. Type errors in gradual type systems are often more manageable than in Soft Typing and SBA, often more comparable to type errors in statically typed languages.

[1] `http://www.plt-scheme.org/software/mrspidey/`

Gradually typed languages have different degrees of runtime checking. Typed Racket was developed as a path for module-by-module porting of existing untyped Racket modules to a typed sister language [TH10]. Once a module is ported and type checked, it is protected from untyped modules by inserting runtime checks. Typescript [Mic12], a gradual type system for Javascript, does not add runtime checks to untyped interactions. Runtime checks affect performance, but give better errors and ensure type errors result in the program failing more quickly.

2.3 Interlanguage Interoperability

This section compares several existing languages that feature interlanguage interoperability.

Clojure is a dynamic functional language hosted on the Java Virtual Machine. It provides interoperability with Java libraries. As Clojure is a dynamically typed language, it does not give strong type guarantees at compile time that interactions with Java are type safe.

Scala is a statically typed language on the Java Virtual Machine offering integrated interoperability with Java, a typed language. Scala objects and classes can "inherit from Java classes and implement Java interfaces" [Ode+06] with the usual static type guarantees normal Scala code enjoys. Scala offers an Option type to safely eliminate null pointers. Java Generics are also fully supported by Scala, accounting for Scala support existential types

Typed Racket includes safe interoperability between any combination of typed and untyped Racket modules [TH10; THF08]. Interactions with untyped modules are protected by adding runtime checks based on expected types. Typed Racket implements a sophisticated blame calculus, based on Wadler [WF09]. It ensures error messages always correctly *blame* the source of type errors, which can be difficult to determine in the presence of higher-order functions.

2.4 Record Types

O'Caml-style extensible record types have been the subject of extensive research (eg. Wand [Wan89], Cardelli and Mitchell [CM91], Harper and Pierce [HP91]). Typed Clojure's heterogeneous map types show some resemblance to extensible record types, but a survey of the literature did not reveal research flexible enough to capture common usages of maps in Clojure. This most likely implies that

designing such a system for Typed Clojure would be hard. For this reason, heterogeneous map types are kept as simple as possible while still being useful.

2.5 Intersection, Union, and Singleton Types

Intersection and union types are interesting type constructs relevant to capturing the complicated types common in dynamic languages. An expression of type `(I a b)`, an intersection type including types `a` and `b` in Typed Clojure, can be used safely in both positions expecting type `a`, and positions expecting type `b`. An expression of type `(U a b)`, a union type including types `a` in Typed Clojure, and expressions of this type can be used safely in positions that expect a type that is *either* type `a` or `b`. For example, `(U Number Integer)` cannot be used in positions expecting `Integer`, but `(I Number Integer)` can be used in positions expecting `Integer`.

Intersection, union, and singleton types are best considered as extrinsic types. Hayashi [Hay91] describes two type theories, ATT which includes intersection, union, and singleton types, and ATTT which further extends ATT to include refinement types, which can be classified as extrinsic types. Typed Clojure can also be considered an extrinsic type system which supports union, intersection, and singleton types, but in less sophisticated forms than ATT and ATTT. In particular, singleton types in ATTT are much more advanced than in Typed Clojure, which offers only singleton types for values like `:a` and `nil` and, unlike ATTT, no way to parameterise over singletons.

Several interesting projects have used intersection or union types. Intersection types were originally introduced by Coppo, Dezani-Ciancaglini, and Venneri [CDCV81] for the λ-calculus. The use of union types to type check dynamically typed languages with soft typing systems dates back to Cartwright and Fagan [CF91].

Forsythe, a modern ALGOL dialect by Reynolds [Rey81; Rey96] was the first wide-spread programming language to use intersection types. Uses of intersections in Forsythe include representing extensible record types and function overloading.

Refinement types add a level of extrinsic types refining an existing intrinsic type system in order to type check more detailed properties and invariants than standard static type systems (described by Freeman and Pfenning [FP91]). Refinement types have similarities with Typed Clojure, in that Typed Clojure adds a level of extrinsic types on top of Clojure. Intersection types are critical here to allow more than one property or invariant to be expressed for a function. SML

CIDRE is a refinement type checker for Standard ML by Davies [Dav05].

St-Amour, Tobin-Hochstadt, Flatt, and Felleisen describe the *ordered intersection types* used in Typed Racket [SA+12] that provide a kind of function overloading. Typed Clojure takes a similar approach for its representation of functions. Typed Racket uses union types which are used in Typed Clojure in very similar ways, like for creating ML-like "datatypes".

2.6 Java Interoperability in Statically Typed Languages

Typed Clojure features integration with Java's static type system to help verify interoperability with Java as correct. Section 3.2.2 presents Typed Clojure's treatment of Java's *null* pointer. Section 3.2.1 shows how Typed Clojure deals with shortcomings of Java's static type system relating to primitive arrays.

Other statically typed languages have similar goals to support Java interoperability, like, most completely, Scala. Scala [Ode+06] is tightly integrated with Java. It manages interactions with Java's *null* pointer by using an *Option* type, which (if used correctly) provides strong elimination for *null*. *null* is still not expressible as a type in Scala, however. Arrays are parameterised by a non-variant (or invariant) parameter, which are checked using *conservative approximation* [OSV08]. For example, this checks that covariant parameters are used only in covariant positions.

2.7 Function Types

There are several different approaches to representing functions in programming languages.

In typed languages like Haskell, functions are as simple as possible, taking a single argument. A function with multiple arguments is represented by chaining several single-argument function together, or by using lists, or using tuples. This first style is characterised by direct syntactic function currying, where applying a function to less than its maximum number of arguments results in another function that takes the remaining arguments. Using tuples instead requires that the same number of arguments be supplied, while using lists allows any number, but requires that all arguments have the same type.

In untyped languages like Scheme, functions can have more complicated arity. In this context, functions have a fixed number of required parameters, and an

optional *rest* parameter, which represents the sequence of the arguments provided after the required parameters. It is an error to apply a function to fewer arguments than the functions number of required parameters. This style features sophisticated support for functions with variable-arity. For example, Scheme functions can dispatch on the number of arguments provided, and support an optional *rest* parameter which can accept any number of arguments.

In this regard, Clojure takes an approach identical to Scheme, and supports all the features mentioned in the previous paragraph, and none characterised by Haskell-style functions. For this reason, we ignore the tradeoffs associated between the two approaches and move directly to literature applicable to typing Scheme-style functions.

2.7.1 Variable-Arity Polymorphism

Strickland *et al.* invented a type system supporting variable-arity polymorphism [STHF09] a version of which is included in the current implementation of Typed Racket. Their main innovation centres around *dotted type variables*, which represent a heterogeneous sequence of types. Dotted type variables allow *non-uniform* variable-arity function types, which are used to check definitions and usages of functions with non-trivial rest parameters

For example in Clojure, the function `map` takes a function and at least one sequence, and returns a sequence of the results of applying the function to each subsequent element of the sequences simultaneously.

Listing 2.1: An application of the non-uniform variable-arity function `map`

```
(map + [1 2] [2.1 3.2])
;=> (3.1 5.2)
```

(Line comments in Clojure begin with `;` and comments to the end of the line. We use `;=>` to mean *evaluates to*).

To statically check calls to `map`, we must enforce the provided function argument can accept as many arguments as there are sequence arguments to `map`, and the parameter types of the provided function can accept the pair-wise application of the elements in each sequence. This is a complex relationship between the variable parameters and the rest of the function. Listing 2.1 requires the first argument to `map` to be a function of 2 parameters because there are two sequence parameters. `+` takes any number of `Number` parameters, and applying pair-wise arguments of `(Vector Long)` and `(Vector Double)` results in types `Long` and `Double` being applied to `+`. These are subtypes of `Number`, so the expression is well typed.

2.8 Type Inference

2.8.1 Local Type Inference

Typed Racket uses Local Type Inference [PT00] as an inference and checking tool. Pierce and Turner [PT00] divide Local Type Inference into two complementary algorithms. *Local type argument synthesis* synthesises type arguments to polymorphic applications, and *bidirectional propagation* propagates type information both down and up the source tree, known as *checking* and *synthesis* mode respectively.

Listing 2.2: Bidirectional checking algorithm with Typed Clojure pseudocode

```
(map (fn [[a :- Long] [b :- Float]]
       (+ a b))
     [1 2]
     [2.1 3.2])
;=> (3.1 5.2)
```

The pseudocode in Listing 2.2 show both algorithms in action. Local type argument synthesis is able to infer the type arguments to `map` by observing the argument types of the first argument to `map` and the types of subsequent sequence arguments. Bidirectional checking then *synthesises* the resulting type of the expression by *checking* each element of `[1 2]` is a subtype of `a`, and each element of `[2.1 3.2]` is a subtype of `b`. The result of the anonymous function argument is *synthesised* from the type of `(+ a b)` as `Float`. We now have sufficient information to synthesise the type of Listing 2.2 to be `(LazySeq Float)`.

Pierce and Turner split *local type argument synthesis* into two further algorithms: bounded, and unbounded quantification [PT00]. Typed Racket supports unbounded polymorphism [TH10], implementing the latter algorithm by Pierce and Turner. Scala supports bounded quantification with F-bounded polymorphism [Can+89], basing its type argument synthesis on the bounded quantification algorithm.

Pierce and Turner explicitly forbid [PT00] attempting to synthesise type variables with interdependent bounds, including F-bounds, having failed to devise an algorithm to infer these cases. Scala's type argument synthesis implementation deviates from Pierce and Turner and supports these features. I am not aware of papers specifically describing Scala's modifications, but they are at least inspired by Scala's spiritual ancestors Generic Java [Bra+98] and Pizza [OW97].

Hosoya and Pierce [HP99] reiterate two common problems with Local Type Inference: "hard-to-synthesise arguments" and "no best type argument". The

first problem occurs because both local type argument synthesis and bidirectional propagation cannot perform synthesis simultaneously.

Listing 2.3: Hard-to-synthesise expression

```
(map (fn [a b]
       (+ a b))
     [1 2]
     [2.1 3.2])
```

Listing 2.3 shows an example of this limitation, here caused by both not providing type arguments to `map` and not providing the parameter types of `(fn [a b] (+ a b))`. Cases where both algorithms can simultaneously recover new type information are usually "hard-to-synthesise". "No best type argument" describes the situation where the results of local type argument synthesis yield more than one type, and no type is better than the other. Sometimes we cannot recover and synthesis fails.

2.8.2 Colored Local Type Inference

Scala's type checking uses Colored Local Type Inference [OZZ01], a variant of Local Type Inference [PT00] specifically designed to improve inference with certain kinds of Scala pattern matching expressions. It allows *partial* type information to propagate down the syntax tree, instead of only full type information as required by Local Type Inference.

Colored types contain extra contextual information, including the propagation direction and missing parts of the type. They are generally useful for describing "information flow in polymorphic type systems with propagation-based type inference" [OZZ01]. Colored Local Type Inference is a candidate for future extensions to Typed Clojure's inference.

2.9 Bounded and Unbounded Polymorphism

Local type inference by Pierce and Turner [PT00] describe two implementations of type variables, for bounded and unbounded type variables. The bounded implementation is presented as an optional extension to the unbounded implementation, which preserves all properties described in the Local Type Inference algorithm.

An unbounded type variable does not have subtype constraints. Bounded type variables can have subtype constraints, and subsume unbounded type vari-

ables [PT00], as a unbounded variable can be represented as a variable bounded by the Top type.

Still, unbounded type variables have an advantage: their implementations are simpler in the presence of a *Bottom* type. The constraint resolution algorithm for bounded variables is more subtle, due to "some surprising interactions between bounded quantifiers and the *Bot* type" [PT00], described fully by Pierce [Pie97].

Typed Racket [THF08] supports unbounded polymorphism, while Scala [Ode+06] supports an extended form of bounded polymorphism called F-bounded polymorphism [Can+89], which allows the bound variable to occur in its own bound. F-bounded polymorphism is useful in the context of object-oriented abstractions, as demonstrated by Odersky [Ode+06]. This is one possible explanation why Typed Racket, which is not built on abstractions like Scala, does not support bounded quantification. Unfortunately, no Typed Racket paper mentions bounded quantification, so the rationale is not clear.

Clojure, like Scala, is built on object-oriented abstractions. Clojure protocols and Java interfaces (interfaces are supported by Clojure) are good candidates for bounds in bounded or F-bounded polymorphism.

2.10 Typed Racket

Typed Racket is a statically typed sister language of Racket. It attempts to preserve existing Racket idioms and aims to type check existing Racket code by simply adding top level type annotations [TH10].

Typed Racket fully expands all macro calls before type checking [TH10], avoiding the complex semantics of type checking macro definitions, an ongoing research area summarised by Herman [Her10]. Typed Clojure follows a similar strategy; only the fully macro-expanded form will be type checked. Type checking macro definitions are outside the scope of this project.

Along with a full static type system, Typed Racket also uses runtime contracts to enforce type invariants at runtime at the interface with untyped code [THF08]. Utilising runtime contracts to aid type checking is outside the scope of this project, but would be considered desirable and accessible future work.

Two other Typed Racket features that will be explored are recursive types and refinement types [TH10]. Recursive types allow a type definition to refer to itself, enabling structurally recursive types like binary trees. Refinement types let the programmer define new types that are subsets of existing types, such as the type for even integers, a subset of all integers. Both these features would fit

well in a future implementation of this project.

2.11 Occurrence Typing

Dynamically typed languages use an ad-hoc combination of type predicates, selectors, and conditionals to steer execution flow and reason about runtime types of variables. Typed Racket uses occurrence typing to capture these ad-hoc type refinements. For example, Listing 2.4 shows occurrence typing following the implications of the type predicate *number?* and the selector *first*, and utilises those implications to refine the type of `x`. If the test at line 3 succeeds, occurrence typing refines the type of `(first x)` to be `Number`, which allows `(+ 1 (first x))` to be well typed. Similarly at line 4, we can be sure that `(first x)` is a `String`, since we have ruled out the case of being a `Number`.

Listing 2.4: A well typed form utilising occurrence typing with Clojure syntax

```
(let [x (list (number-or-string))]
  (cond
    (number? (first x)) (+ 1 (first x))
    :else               (str (first x))))
```

Occurrence typing [THF08; THF10] 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. (See Section 4.1.3 for details on how Typed Clojure uses occurrence typing).

For occurrence typing to infer propositions from type predicate usages, it requires two extra annotations: a "then" proposition when the result is a true value, and an "else" proposition for a false value. For example, `number?` has a "then" proposition that says its argument is of type `Number`, and an "else" proposition that says its argument is not of type `Number`.

An exciting application of occurrence typing as yet unexplored is facilitating null-safe interoperability with Java. By declaring `nil` (Clojure's value of Java's `null`) to *not* be a subtype of reference types, we can begin to statically disallow potentially inconsistent usages of `nil` as part of the type system.

Listing 2.5: Observing nil-checks using occurrence typing

```
(let [a (ObjectFactory/getObject)]
  (when a
    (expects-non-nil a)))
```

Listing 2.5 infers from the Java signature `Object getObject()` that `a` is of type `(U nil Object)`. This is equivalent to Java's `Object` static type, as `null` is a member of all reference types. By surrounding the call `(expects-non-nil a)` with `(when a ...)`, we guarantee that `a` is non-nil when passed to `expects-non-nil`. Occurrence typing infers this by observing `nil` is a false value in Clojure, therefore `a` cannot be `nil` in the body of the `when`, refining `a`'s type to `Object` from `(U nil Object)`.

Occurrence typing is a relatively simple technique used successfully in Typed Racket. Clojure is similar enough to Racket for occurrence typing to work without issues, and has good potential to help programmers avoid using `nil` incorrectly

2.12 Statically Typed Multimethods

Clojure provides multimethods as a core language feature. This section discusses systems that statically verify type safety for multimethods.

Millstein and Chambers [MC02] describe Dubious, a simple statically typed core language including multimethods that dispatch on the type of its arguments. They tackle a key challenge for statically typing multimethods: "it is possible for two modules containing arbitrary multimethods to typecheck successfully in isolation but generate type errors when linked together." [MC02]

After some investigation, typing multimethods with Typed Clojure is assigned as future work (see Section 6.3).

2.13 Higher Kinded Programming

Many advanced type systems provide support for *higher kinds*, which are "another level up" from types. For example, a type `Number` is distinguished from a *type constructor* `Monad` (see Section 5.3), which is a type level function (similar to the difference between values and functions on the value level, but on the type level).

F_ω is a typed λ-calculus with support for higher kinds, specifically type constructors (see Pierce [Pie02]). Haskell [Sim10] distinguishes between types and type constructors, the latter of which is an essential part of its monad library

(see Section 5.3 for a monad library ported to Typed Clojure that uses type constructors similarly).

2.14 Conclusion

Many related components must come together in the design of a static type system. Typed Racket achieves a satisfying balance of occurrence typing, local type inference and variable-arity polymorphism. Scala features F-bounded polymorphism, a class hierarchy that is compatible with Java, and colored local type inference. Typed Clojure takes inspiration from these, and similar, projects.

SML CIDRE [Dav05] was also a secondary influence - particularly its use of bidirectional checking of extrinsic types including subtyping, intersection types, and its aim to capture program invariants. However, SML CIDRE starts from the intrinsically typed language SML which already has a rich static type system, which leads to many different considerations compared to starting from a dynamically typed language. Much of this influence was via a student-supervisor relationship, making it particularly hard to pin down the specifics.

CHAPTER 3

Design Choices

This chapter describes the design choices made for Typed Clojure. Typed Clojure is designed to be of practical use to Clojure programmers. To this end, many of the design choices are borrowed from other existing projects. Some of the main design choices that specifically concern Typed Clojure are given more detailed descriptions.

3.1 Typed Racket

The majority of the design and implementation of Typed Clojure is based on Typed Racket [TH10]. This is a deliberate choice: the purpose of this work is to produce a practical tool, and there is already good evidence that Typed Racket is practical and makes good tradeoffs and design choices. Our differences are mostly due to the differences between Clojure and Racket. This section details some of these differences.

3.1.1 Occurrence Typing

Occurrence typing, described by Tobin-Hochstadt and Felleisen [THF10], plays a major role in capturing many common Clojure idioms. One such idiom is Clojure's *sequence abstraction*, which provides a common interface to all collections. For example, `seq` tests whether a collection is empty, `first` returns the first element of a collection, and `rest` returns the tail of its collection argument as a new sequence. While these summaries describe their most common usages, they do not tell the whole story.

`seq`'s description might suggest it is better named `not-empty?`, but `seq` is perhaps surprising in two main ways: it accepts `nil` as an argument (returning `nil`), and, rather than returning `true` or `false`, returns a non-empty sequence or `nil` respectively.

From a typing perspective, the subtlety of `seq` is expressing that `(seq coll)`

- is a true value when `coll` is a non-empty collection, and
- is a false value when `coll` is either `nil` or an empty collection.

Occurrence typing allows us to capture this idiom (Listing 3.1).

Listing 3.1: Type associated with clojure.core/seq

```
(All [x]
  [(U nil (Seqable x)) -> (U nil (Seqable x))
   :filters {:then (is (CountRange 1) 0)
             :else (| (is nil 0)
                      (is (ExactCount 0) 0))}])
```

This function type (Listing 3.1) takes an argument of type `(U nil (Seqable x))` (the syntax `(U` $\overrightarrow{t}$`)` constructs a union of types `t`), and returns a type `(U nil (Seqable x))`. Occurrence typing allows us to attach propositions to the return type of a function. Roughly, these say if `(seq coll)` returns a true value, then `coll` is of type `(CountRange 1)` (a type representing a sequence of length 1 or greater), otherwise `coll` is *either* of type `nil` or of type `(ExactCount 0)` (a type representing an empty sequence). (See Section 4.6 for a more detailed explanation of the filter syntax).

`first` is overloaded in a similar fashion to `seq`: `(first coll)` returns

- `nil` if `coll` is `nil` or an empty collection, and
- returns the first item of the collection if `coll` is a non-empty collection.

Listing 3.2 shows the type of `first`. The main thing to grasp from this definition is that three function types are given: the first type handling the first case above, the second type handling the second case above, and the third arity acting as a base case, general enough to catch both cases.

Listing 3.2: Type associated with clojure.core/first

```
(All [x]
  (Fn [(U nil (I (Seqable x) (ExactCount 0))) -> nil]
      [(I (Seqable x) (CountRange 1)) -> x]
      [(U nil (Seqable x)) -> (U nil x)])))
```

`seq` and `first` interact via occurrence typing. For example, Listing 3.3 tests `(seq coll)` which means occurrences of `coll` in the "then" branch are non-empty and those in the "else" branch are empty. We can safely call `(.method (first coll))` in the "then" branch (assuming elements of `coll` have a no-argument method called `method`) always avoiding a *NullPointerException*.

Listing 3.3: Example of using the sequence abstraction with occurrence typing

```
...
(let [coll (possibly-empty-coll)
  (if (seq coll)
    (.method (first coll))
    nil))
...
```

3.1.2 Variable-arity Polymorphism

Strickland, Tobin-Hochstadt, and Felleisen describe variable-arity polymorphism [STHF09], which has been implemented in Typed Racket. Typed Clojure directly ports most of Typed Racket's implementation of variable-arity polymorphism, as described in Section 4.4.

Some common idiomatic Clojure functions like `assoc`, `hash-map`, and `partial` cannot be typed by Strickland *et al.*. Instead, Typed Clojure hard-codes some of these functions as primitives that cannot be used in a higher-order context. Expressing the types of these functions is set as future work (Section 6.1).

3.2 Safer Host Interoperability

Clojure implementations are designed to target popular platforms, providing fast, unrestricted access to their respective hosts. Notable implementations target the Java Virtual Machine (via Clojure), the Common Language Runtime (via ClojureCLR), and Javascript virtual machines (via ClojureScript). Host interoperability is the main source of incompatibility between Clojure implementations. Aside from syntactic matters, Clojure makes no attempt at reconciling host interoperability differences: it is up to the programmer to decide how to best write portable Clojure code.

Typed Clojure is designed for Clojure, which runs on the Java Virtual Machine (JVM). Clojure embraces the JVM as a host by sharing its runtime type system and providing direct interaction with libraries written in other JVM languages.

Most commonly, Clojure programmers use libraries written in Java. Java is a statically typed language, and it follows that any interaction with Java will already have an annotated Java type. Typed Clojure uses Java's static type annotations to help statically type check interactions with Java from Clojure.

This section details where Typed Clojure attempts to improve some shortcomings of Java's static type system.

3.2.1 Primitive Arrays

Primitive arrays in Java are covariant, which is well-known to be statically unsound [OW97; BK99]. This means that at compile time we cannot completely trust the type signature of any Java method or field that involves an array.

We handle this by introducing a type `(Array c t)`, the type of the array that has the Java component type `c` and that can read and write values of type `t`. Internally, `Array` types expand to `(Array3 c w r)`, the type of the array that has the Java component type `c`, that can write values of type `w` (which is contravariant in `Array3`), and that can read values of type `r` (which is covariant in `Array3`). Concretely, `(Array c t)` expands to `(Array3 c t t)`.

`Array3` is useful when an array is passed to/from Java or some untyped Clojure code. As soon as an array leaves the safety of Typed Clojure, it can be considered to have type `(Array3 c Nothing t)`. It is illegal to write to an array of this type in Typed Clojure without first checking its component type, but reads remain legal.

This scheme should be considered experimental, and is not completely fleshed out or implemented. Notably, the component type `c` seems redundant, and Typed Clojure currently does not provide syntax to "cast" an unwriteable array to be writeable.

3.2.2 Interaction with *null*

As of Java 7, Java does not provide a type-safe construct for eliminating occurrences of *null*. Instead, Java programmers rely on ad-hoc approaches, like testing for the presence of *null* (like Listing 3.4) or prior domain knowledge. One solution to this problem is an Option type (also known a Maybe type). Scala provides *scala.Option* for this purpose. The programmer provides two branches of code, (respectively to execute if an expression is *null*, and if it is not) and the appropriate branch is chosen at runtime. There is no such construct in idiomatic

Clojure, so this solution is inappropriate: a satisfactory solution should reuse current Clojure idioms.

Listing 3.4: null elimination in Java

```
...
Object a = nextObject();
if (a != null)
  parseNextObject(a);
else
  throw new Exception("null Pointer Found!);
...
```

It is a goal of Typed Clojure to statically prevent misuses of Java's *null*. At runtime, Clojure represents *null* as the value `nil`; Typed Clojure assigns this value the static type `nil` (a singleton type containing just the value `nil`). This enables more expressive types that explicitly communicate where `nil`, and therefore *null*), is allowed. Section 1.4.9 gives an example of how this could be useful when using Java interoperability.

The crucial aspect of this approach is that misusing *null* results in a *static type error*. This differs significantly from Java: according to Java's static type system, it is implicit that almost every operation could involve *null*! Occurrence typing [THF10] plays a crucial role in making this a practical design choice, by understanding existing Clojure idioms related to *null* elimination (see Section 2.11). Eliminating occurrences of *null* in Java is the programmer's responsibility. Instead, Typed Clojure helps the programmer by statically proving many misuses of *null* are impossible.

This decision, however, needs further work to be completely satisfying. In particular, type inference should be improved in certain common situations. For example, we currently do not know how to infer `(filter identity coll)` as returning a sequence with the elements of `coll` that are not `nil` or `false`, for any sequence `coll` (see Section 4.6). Also, Typed Clojure lets the programmer define a set of unenforced *assumptions* where *null* is allowed in Java fields and methods (see Section 1.4.9), which is potentially brittle: there is nothing stopping the Java code from introducing unexpected *null* values at runtime. Adding runtime assertions that enforce these assumptions would allow more accurate and earlier errors, an approach similar to that taken by Typed Racket [TH10] when interfacing with untyped Racket code.

3.3 Clojure type hierarchy

The Clojure type hierarchy is a set of Java classes and interfaces, mostly in the *clojure.lang* package, that define Clojure's core interfaces and data structures. Typed Clojure parameterises some of these classes in a way that resembles Java or Scala generics.

Clojure's type hierarchy was accommodating to this process. For example, `clojure.lang.Seqable` is the main interface behind Clojure's sequence abstraction. Sequences are immutable collections, so it was natural to parameterise `Seqable` with one covariant argument (covariant means `(Seqable Integer)` is a subtype of `(Seqable Number)` because `Integer` is a subtype of `Number`). All "seqable" classes in the Clojure hierarchy implement `Seqable`, so updating them to use the parameterised `Seqable` was a matter of searching for classes that had `Seqable` as an ancestor and replacing the unparameterised `Seqable` with the corresponding parameterised version.

Defining new "parameterised" versions of existing classes and interfaces do not affect anything outside Typed Clojure. They are only for use by Typed Clojure.

3.4 Protocols and Datatypes

Protocols and datatypes are two of Clojure's main means of abstraction. Protocols define a set of unimplemented methods, and each must be assigned types. Datatype definitions are type checked by ensuring the methods they implement match the types defined by the Java interface or assigned to the protocol method they are extending.

Common usages of datatypes are checkable, like datatype constructors, field accesses, and protocol method invocation. These are described in Section 1.4.3. Mutable fields in datatypes are not yet supported, but is also a rarely used feature.

3.5 Local Type Inference

Type inference in Typed Clojure is based on Local Type Inference by Pierce and Turner [PT00]. The main extension to Local Type Inference in Typed Clojure is support for *dotted variables* for variable-arity polymorphism. This implementation is ported directly from Typed Racket.

3.6 F-bounded Polymorphism

Typed Clojure includes support for F-bounds on type variables[Can+89], as an extension to Local Type Inference. Type variables can have upper and lower bounds, and F-bounds allow type variable bounds to recursively refer to the type variable being bounded. This is similar to Java and Scala, which both support F-bounded polymorphism.

3.7 Heterogeneous Maps

Heterogeneous map types in Typed Clojure are simple enough to avoid the complications associated with concatenable records (see Section 2.4) while still supporting many Clojure idioms. They support adding new entries with keyword keys, removing entries, and lookup of keyword keys with the type associated with each key in a map tracked separately, similar to type systems for record types (see work by Harper and Pierce [HP91]). When using these operations with non-keyword keys, heterogeneous maps are treated like plain hash-maps.

Heterogeneous map types do not support concatenation (merging), as they only hold positive information on the keys with entries, not which keys are *absent*. This makes concatenation unsafe, as keys to the right of a concatenation can "overwrite" keys to the left with different types. With no information on the absence of keys, concatenation becomes an unsound operation.

We formalised some aspects of the behaviour of heterogeneous map types in Appendix A. This also serves as a prototype for formalising other aspects in the future. Here we give a formal syntax, operational semantics and type system using inference rules, as is standard in the programming language community (see, eg., Pierce [Pie02]).

CHAPTER 4

Implementation

This chapter discusses the implementation of the Typed Clojure prototype type system (we refer to this implementation as *Typed Clojure* for the remainder of this chapter) in some detail concentrating on significant challenges that were identified and overcome. Many aspects of Typed Clojure's design follow the implementation of Typed Racket, which is reflected in this chapter. (The code is available on Github at `https://github.com/frenchy64/typed-clojure`).

4.1 Type Checking Procedure

Typed Clojure and Typed Racket differ significantly in how type checking is integrated into their programming environments. Typed Racket is implemented as a language on the Racket platform, which provides highly sophisticated and extensible macro facilities. Interestingly, this allows Typed Racket to be entirely implemented with macros. Instead, Typed Clojure is implemented as a library that utilises abstract syntax trees (AST) generated by analyze [BSc12], a library I developed for this project. This strategy follows common practice for Clojure projects, which favours providing modular libraries over modifying the language.

This section goes into details on the implementation of Typed Clojure. First, a high-level overview is given on the type checking procedure. Then the interfaces to particular high-level functions are discussed.

4.1.1 General Overview

There are several stages to type checking in Typed Clojure. Type checking is typically initiated at the read-eval-print-loop prompt (REPL), for example `(check-ns 'my.ns)` checks the namespace `my.ns`. Before type checking begins, all global type definitions in the namespace are added to the global type environment by compiling the namespace. These type definitions include

- type alias definitions
- global variable, protocol, datatype, and Java Class annotations
- Java method annotations, such as `nilable-param` and `non-nil-return`

An AST is then generated from the code contained in the target namespace. This AST is then recursively descended and is type checked using local type inference. Currently only one error is reported at a time, and type checking stops if a type error is found.

4.1.2 Bidirectional Checking

The interface to the bidirectional checking algorithm is `typed.core/check`, which takes an expression, represented as an AST generated from *analyze*, and an optional expected type for the given expression. If the expected type is present, the bidirectional algorithm *checks* that the expected type matches the actual type of the expression. If the expected type is omitted, a type is instead *synthesised* for the expression. The algorithm is based on Pierce and Turner's Local Type Inference [PT00] and the implementation is similar in form to Typed Racket's [THF08], in that one function with an extra "expected type" argument is preferred over two complementary functions, one for checking and one for synthesis.

The main difference between Typed Clojure's and Typed Racket's bidirectional checking algorithm is the representation for expressions. Typed Racket relies on pre-existing Racket features like syntax objects for expression representation. Clojure instead leans towards abstract syntax tree representation, despite its Lisp heritage. In terms of the bidirectional checking, the difference is mostly cosmetic.

4.1.3 Occurrence typing

Typed Clojure's implementation of occurrence typing is ported and extended from Typed Racket. Occurrence typing plays several roles in Typed Clojure. First, occurrence typing is used to update the type environment at every conditional branch. Second, it is used to calculate whether branches are reachable. Third, paths are used extensively in Typed Clojure. The implementation of these features are discussed in this section.

The basic idea of occurrence typing involves keeping a separate environment of *propositions* that relate bindings to types. These propositions are then used

to update the type environment. In Typed Clojure there are several types of propositions, referred to as *filters*. They are based on the theory by Tobin-Hochstadt and Felleisen [THF10], and are ported directly from Typed Racket.

- TopFilter and BotFilter represent the trivially true and trivially false propositions respectively.
- TypeFilter and NotTypeFilter represent a positive or negative association of a binding name to a type. The syntax `(is t name)` means a proposition that the binding called `name` is of type `t` (corresponding to TypeFilter). The syntax `(! t name)` means a proposition that the binding called `name` is *not* of type `t` (corresponding to NotTypeFilter).
- AndFilter and OrFilter represent logical conjunction of propositions (written `(&` $\overrightarrow{p}$`)`), and logical disjunction of propositions (written `(|` $\overrightarrow{p}$`)`) for propositions `p`.

Propositions can optionally carry *path* information represented by a sequence of *path elements*, which signify which part of the binding's type to update. For example, Typed Racket uses *car* and *cdr* path elements to track which component of a cons type to update. Paths are particularly useful in Typed Clojure. There are path elements for traversing heterogeneous map types (KeyPE), inferring length information (CountPE), and `first` and `rest` paths for sequences. These additions do not appear to introduce any major new complexities related to paths.

4.2 Polymorphic Type Inference

The polymorphic type inference algorithm is directly ported from Typed Racket with slight extensions for bounded variables, and is directly based on Pierce and Turner's Local Type Inference [PT00]. A common entry point for inferring type variables for polymorphic function invocations is `typed.core/infer`.

`infer` is invoked like `(infer X Y S T R expected)`, where

- `X` is a map from type variable names to their bounds (representing the type variables in scope),
- `Y` is a map from type variable names to their bounds (representing the *dotted* type variables in scope),

- `S` and `T` are sequences of types of equal length, (usually the types of the actual arguments provided and the types of the parameters of the polymorphic function),
- `R` is a result type (usually the return type of the polymorphic function),
- `expected` is the expected type for R, or the value `nil`,

and returns a *substitution* that satisfies the following conditions:

- Pairwise, each `S` is a subtype of `T`,
- `R` is below `expected`, if `expected` is provided.

A substitution maps type variables to types. It is valid to replace all occurrences of the type variables named in the substitution with their associated type. For example, substitution are often applied to `R` by the caller of `infer` to eliminate the type variables in `X` and `Y`.

`infer` is almost always used when invoking polymorphic functions like, for example, `constantly`, which has type `(All [x y] [x -> [y * -> x]])` (read as a function taking `x` and returning a function that takes any number of `y`'s and returns `x`). For instance, `((constantly true) 'any 'number)`, results in the value `true`.

Type checking the invocation `(constantly true)` calls `infer` roughly like

```
(infer {'x no-bounds 'y no-bounds}
       {}
       [(parse-type 'true)]
       [(make-F 'x)]
       (with-frees [(make-F 'x) (make-F 'y)]
         (parse-type '[y * -> x]))
       nil)
```

where the internal Typed Clojure bindings

- `no-bounds` is the type variable bounds with upper bound as `Any` and lower bound as `Nothing`,
- `parse-type` is a function that takes type *syntax* and converts it to a type (its argument must be quoted),
- `make-F` is a function that takes a name symbol and returns a type variable type of that name, and

- `with-frees` is a macro that brings the type variables named in its first argument into scope in its second argument.

This returns a substitution that replaces occurrences of `x` with `true` and `y` with `Any`. This helps infer the type of the expression `(constantly true)` as `[Any * -> true]` (where `true` is the singleton type containing just the value `true`).

4.3 F-Bounded Polymorphism

A feature not present in Typed Racket is bounded polymorphism. Several changes were needed to support bounded polymorphism. In every position where a set of type variables was required, it was replaced by a map of type variables to bounds.

Bounds consist of an upper and lower type bound, or a *kind bound*. Kind bounds are experimental following the inclusion of user definable type constructors (motivated in section 5.3). They are a stub for a more comprehensive treatment of higher-kinded operators such as that described by Moors, Piessens, and Odersky for Scala [MPO08]. At present, a type variable can only be instantiated to a type between its upper and lower bounds, or, if a kind bound is defined instead, to a kind below the kind bound.

F-bounded polymorphism allows type variables to refer to themselves in their type bounds. Bounds are checked after a substitution is generated, guaranteeing no substitution can violate type variable bounds. To support F-bounds, the substitution being checked is applied to the lower and upper bounds for each type variable, and the type associated with the type variable in the substitution is checked to be between these bounds.

4.4 Variable-arity Polymorphism

Variable-arity polymorphism in Typed Clojure is directly ported from Typed Racket. This was the most complicated part of the prototype. At the center of the implementation is manipulating dotted type variables, which can represent a sequence of types.

It also required changes to the polymorphic type inference, where each reference to a type variable required a special case for a dotted type variable. For example, the constraint-generation algorithm for Local Type Inference features extra kinds of constraints for dotted variables. Strickland, Tobin-Hochstadt, and

Felleisen elaborate on the particular changes required for the Typed Racket implementation [STHF09].

Porting Typed Racket's variable-arity polymorphism implementation was tedious because some of the relevant internal functions interact in strange ways with the rest of Typed Racket. My impression was that Typed Racket was initially designed without variable-arity polymorphism and was added without major changes to other components. Typed Clojure was developed with the same design so full variable-arity polymorphism implementation could be ported without change.

4.5 Portability to other Clojure Dialects

Typed Clojure was built for the Clojure programming language, whose compiler and data structures are implemented in Java. ClojureScript is the first major Clojure dialect to be written in Clojure, and it is likely future dialects of Clojure will follow this example. Where Clojure uses Java Classes and Interfaces, ClojureScript's compiler and data structures are written in terms of Clojure's two core abstractions: protocols and datatypes. It would be desirable to port Typed Clojure to such Clojure dialects while keeping the core of the implementation constant.

A significant portion of Typed Clojure is theoretically platform independent but there are challenges to targeting new dialects, including incompatible host interoperability and non-standardised abstract syntax trees. The first issue is predictable due to a core philosophy of Clojure dialects: host interoperability is non-portable[1]. Each dialect of Clojure has a unique host interoperability story and Typed Clojure should cater for them separately. The second issue is potentially resolvable either by enforcing a standard representation for abstract syntax trees across Clojure implementations, or developing a library that provided a common interface to abstract syntax trees for each Clojure implementation.

4.6 Proposition Inference for Functions

A feature not yet implemented in Typed Racket is the ability to infer new propositions based on existing propositions of a function. This feature was added to Typed Clojure to support filtering a sequence based on negative information, such as filtering values that are *not* `nil`.

[1]See the complete Clojure rationale: http://clojure.org/rationale

Listing 4.1: Type annotation for filter

```
(ann clojure.core/filter
  (All [x y]
    [[x -> Any :filters {:then (is y 0)}] (U nil (Seqable x)) ->
       (Seqable y)]))
```

To better understand the problem, Listing 4.1 presents the type of `filter`, which takes a function `f` and a sequence `s` as arguments, and returns a sequence that contains each element in `s` such that applying `f` to the element returns a true value. The `:filters` syntax requires some explanation. Function types support an optional *filter set* attached to the return type, written as a map with `:then` and/or `:else` keys (if omitted, they default to the trivially true proposition which has no effect). The "then-proposition" and "else-proposition" are added to the type environment when the return value is a true and false value respectively. For example, the filter set `{:then (is y 0)}` is read "if the return value is a true value, then the first argument must be of type y, otherwise if it is a false value, nothing interesting is enforced" The type given for `filter` works because the type variable `y` occurs in both the "then-proposition" of the first argument and the return type `(Seqable y)`.

Listing 4.2: Troublesome filter

```
(filter (fn [a] (not (nil? a))) coll)
```

The difficulty starts with something like Listing 4.2, where the inferred filter set for the first argument to `filter` is `{:then (! nil 0) :else (is nil 0)}`[2]. The "then-proposition" `(! nil 0)` does not fit with `(is y 0)` that `filter` expects.

We can sometimes get around this if we already have a predicate with a positive "then-proposition". For example, if we are filtering out `nil` values from a sequence of type `(Seqable (U Number nil))`, we can replace Listing 4.2 with `(filter number? coll)`, where `number?` has the filter set `{:then (is Number 0) :else (! Number 0)}`. This does not work, however, when filtering a sequence of type like `(Seqable (U x nil))` for some unspecified `x` because there is no built-in predicate with "then-proposition" `(is x 0)`.

Listing 4.3: Filtering with negative propositions

```
(filter (ann-form
          #(not (nil? %))
          [(U nil x) -> boolean :filters {:then (is x 0)}])
        mvs)
```

[2](! nil 0) is the proposition that the first argument is *not* of type `nil`.

Instead, we generate new propositions using a technique suggested by Tobin-Hochstadt [TH12], that follows from the occurrence typing calculus defined by Tobin-Hochstadt and Felleisen [THF10]. First the filter set for functions are inferred as usual. To collect new propositions, the "then-proposition" is applied to the type environment (which maps local bindings to types). Any types associated with bindings that are changed after this are represented as new propositions, which are added to the "then-proposition" for this filter set. The same procedure is followed for the "else-proposition".

Using this technique, the anonymous function in Listing 4.3[3] has the filter set `{:then (& (! nil 0) (is x 0)) :else (is nil 0)}` which is good enough to infer the `filter`ed result as `(Seqable x)`.

Further work in this area is needed when filtering on a non-anonymous function. For example, it is not clear how to infer the common idiom `(filter identity coll)` as returning a sequence of non-nil elements, for any sequence `coll`. Inferring new propositions for already existing functions like `identity` does not fit with Tobin-Hochstadt and Felleisen's calculus [THF10], confirmed by Tobin-Hochstadt [TH12] as future work in this area.

[3]The Clojure syntax `#(not (nil? %))` is equivalent to `(fn [a] (not (nil? a)))`

CHAPTER 5

Experiments

In this chapter we work through several examples of using Typed Clojure and gauge how well the current prototype handles them. We intentionally chose examples that could be challenging to type check, or were particularly useful to the everyday Clojure programmer.

5.1 Java Interoperability

It is idiomatic and common in Clojure to interface with existing Java code via Java interoperability. Typed Clojure is intended to be useful for practical purposes, so it is important to understand this common feature. To test Typed Clojure's approach, existing code utilising Java interoperability was ported. My porting effort attempted to follow how a real programmer might port code to Typed Clojure; I describe this process step-by-step.

This section describes porting a function from *clojure.contrib.reflect* [1] [2], a Clojure library that relies heavily on Java interoperability. I chose to port `call-method` for several reasons: it chains several Java calls together, it uses primitive arrays, and `null` is a valid value in one place.

Before showing the implementation, there is a brief explanation of the relevant syntax. Java methods are called using the *dot* operator. If `o` is an object, then `(. o m` $\overrightarrow{a}$`)` calls its method named `m`, passing $\overrightarrow{a}$ as arguments. Syntactic sugar allows the method to be named first: `(.m o` $\overrightarrow{a}$`)` is equivalent to `(. o m` $\overrightarrow{a}$`)`. Also, `doto` is convenient notation for chaining multiple method calls to the same object, presumably for side effects, and returns the original object. For example, `(doto o (.m1` $\overrightarrow{a}$`) (.m2` $\overrightarrow{b}$`))` calls methods `m1` (passing $\overrightarrow{a}$ as arguments) and `m2` (passing $\overrightarrow{b}$ as arguments) on `o` in order, and returns `o` with any side effects

[1] *clojure.contrib* Github project: https://github.com/richhickey/clojure-contrib

[2] Note that while the *clojure.contrib* library as a whole should be considered deprecated, useful code still exists within it which apparently has not been migrated elsewhere.

applied to it.

Listing 5.1: call-method

```
;; call-method pulled from clojure.contrib.reflect, (c) 2010 Stuart
   Halloway & Contributors
(defn call-method
  "Calls a private or protected method.

  params is a vector of classes which correspond to the arguments to
  the method e

  obj is nil for static methods, the instance object otherwise.

  The method-name is given a symbol or a keyword (something Named)."
  [^Class klass method-name params obj & args]
  (let [method (doto (.getDeclaredMethod klass
                                         (name method-name)
                                         (into-array Class params))
                 (.setAccessible true))]
    (.invoke method obj (into-array Object args))))
```

The original function is modified slightly for readability and is presented in Listing 5.1.

Listing 5.2: call-method Type Annotation

```
(ann call-method
     [Class Named (IPersistentVector Class) (U nil Object) (U nil
        Object) * -> (U nil Object)])
```

Thankfully this function has up-to-date documentation, and from it we can derive an expected type (Listing 5.2).

Before running the type checker we must convert our array constructors into ones that Typed Clojure can understand. Array types in Typed Clojure are represented by `(Array c t)`, where the Java class `c` is the Java component type, and the Typed Clojure type `t` is the Typed Clojure component type. We can pass this array to Java methods accepting type `c[]`, and we can read and write type `t` to the array from Typed Clojure.

In particular we change two invocations of `into-array`.

- `(into-array Class params)`, which creates an array of type `Class[]` and populates it with the elements of `params`, becomes `(into-array> Class Class params)`, which is of type `(Array Class Class)`.

- `(into-array Object args)`, which creates an array of type `Object[]` and populates it with the elements of `args`, becomes `(into-array> Object (U nil Object) args)` which is of type `(Array Object (U nil Object))`.

The second place in the `Array` type constructor allows fine grained control over what is allowed in the array. The first point above must be type `(Array Class Class)` because the `getDeclaredMethod` method on `java.lang.Class` instances requires an array of non-null `Class` objects. On the other hand, the `invoke` method takes an array that allows *null* members, so its type is `(Array Object (U nil Object))`.

Now we run the type checker, which produces a type error.

```
#<Exception java.lang.Exception: 29: Cannot call instance method
    java.lang.reflect.AccessibleObject/setAccessible on type (U nil
    java.lang.reflect.Method)>
```

Because Typed Clojure assumes all methods return nilable Objects, the call to `getDeclaredMethod` has return type `(U nil java.lang.reflect.Method)`. It is not type safe to call `setAccessible` on this type, so we get a type error.

In this case, Typed Clojure is too conservative: according to its documentation `getDeclaredMethod` never returns *null*. We add this rule with `non-nil-return`.

```
(non-nil-return java.lang.Class/getDeclaredMethod :all)
```

Running the type checker produces a different type error.

```
#<Exception java.lang.Exception: Type Error, REPL:32 - (U
    java.lang.Object nil) is not a subtype of: java.lang.Object>
```

This concerns passing `obj` as the first argument to the `invoke` method. Typed Clojure conservatively defaults method parameter types as non-nullable. Therefore the first parameter of `invoke` is `Object` by default; `obj` is `(U java.lang.Object nil)`. Again, this is too conservative as the first argument can be *null* for static methods, and we use `nilable-param` to specify the first argument of `invoke` may be nil, for the arity of two parameters.

```
(nilable-param java.lang.reflect.Method/invoke {2 #{0}})
```

The final successfully type checked code is presented in Listing 5.3.

Listing 5.3: Type Annotated code for call-method

```
(non-nil-return java.lang.Class/getDeclaredMethod :all)
(nilable-param java.lang.reflect.Method/invoke {2 #{0}})

(ann call-method [Class Named (IPersistentVector Class) (U nil Object)
   (U nil Object) * -> (U nil Object)])

;; call-method pulled from clojure.contrib.reflect, (c) 2010 Stuart
   Halloway & Contributors
(defn call-method
  "Calls a private or protected method.

  params is a vector of classes which correspond to the arguments to
  the method e

  obj is nil for static methods, the instance object otherwise.

  The method-name is given a symbol or a keyword (something Named)."
  [^Class klass method-name params obj & args]
  (let [method (doto (.getDeclaredMethod klass
                                         (name method-name)
                                         (into-array> Class Class
                                            params))
                 (.setAccessible true))]
    (.invoke method obj (into-array> Object (U nil Object) args))))
```

5.2 Red-black trees

This experiment involved porting an implementation of red-black trees used as an experiment for SML CIDRE by Davies [Dav05]. SML CIDRE is a sort-checker for Standard ML that supports refinement types. The red-black tree implementation was ported to work with SML CIDRE to statically check red-black tree invariants. It is a particularly good experiment for SML CIDRE because it generates an unusually large number of intersection types.

Intersection types in SML CIDRE are heavily optimised via memoisation. Because of this, SML CIDRE is able to check this experiment with little trouble. The current Typed Clojure prototype is not as successful, and appears to hang during type checking. As this experiment was particularly ambitious, diagnosing and fixing this issue is delegated to future work.

One interesting point was noticed when porting the experiment. The SML CIDRE version of the red-black tree experiment uses SML CIDRE datasorts to represent the red-black tree invariants. The Typed Clojure version entirely uses plain hash-maps, a common Clojure idiom. It was noticed that the heterogeneous map types that Typed Clojure provides would be sufficient to represent the red-black tree invariants.

5.3 Monads

Monads are an interesting control structure used in functional programming languages, as described by Wadler [Wad95]. They are most recognisable from its inclusion in the statically-typed language Haskell [Sim10], where monads are relied on for many features including global state, file output, and exceptions.

I chose to port the Clojure Contrib library *algo.monads* [Hc12] to Typed Clojure. The library provides several kinds of monads, monad transformers, and monadic functions. Macros are used to provide pleasant syntax for consumers of the library.

5.3.1 Monad Definitions

This library represents a monad as a hash-map with four keys: `:m-bind`, `:m-result`, `:m-zero`, and `:m-plus`. A valid monad must provide the first two, and the latter two may optionally be mapped to the keyword `::undefined`[3].

Listing 5.4: Untyped definition for the identity monad

```
(defmonad identity-m
   "Monad describing plain computations. This monad does in fact
      nothing
    at all. It is useful for testing, for combination with monad
    transformers, and for code that is parameterized with a monad."
  [m-result identity
   m-bind   (fn m-result-id [mv f]
              (f mv))
  ])
```

A monad is defined using the macro `defmonad`. The macro expands to code that binds a var to a hash-map with the aforementioned keys.

[3]Keywords prefixed with `::` are qualified in the current namespace.

Listing 5.5: Type for identity monad

```
(def-alias Undefined ':undefined)
(ann identity-m
     '{:m-bind (All [x y]
                 [x [x -> y] -> y])
       :m-result (All [x]
                   [x -> x])
       :m-zero Undefined
       :m-plus Undefined})
```

A direct typing of the identity monad would look like Listing 5.5. This signature assigns the expected type for the var `identity-m` to be a heterogeneous map type with the minimum entries for monads: `:m-bind` and `:m-result`. The syntax for heterogeneous map types requires a `'` prefix, and any number of key-value pairs are given between curly braces. Heterogeneous map keys must be keywords, so the syntax is made more convenient by omitting the usually required `'` prefix for keyword types.

Monad types in languages with advanced type systems are often abstracted using type constructors. This allows reasoning about monadic code while keeping the particular monad abstract, which is a desirable result, so Typed Clojure was extended to support user definable type constructors.

Listing 5.6: An abstract definition of a monad.

```
(def-alias Monad
  (TFn [[m :kind (TFn [[x :variance :covariant]] Any)]]
    '{:m-bind (All [x y]
                [(m x) [x -> (m y)] -> (m y)])
      :m-result (All [x]
                  [x -> (m x)])
      :m-zero Undefined
      :m-plus Undefined}))
```

Listing 5.6 captures the abstract definition of a monad. A `Monad` is a type constructor parameterised by `m`, which is a type constructor taking a single argument `x` (a type) and returning a type (written `Any`)[4]. The body of the type constructor uses `m` abstractly. Typed Clojure ensures the correct number of arguments are passed and recognises the declared variances for each parameter. In this case the first argument of `m` is declared a covariant position. When instantiated, `m` must also be a type operator of one covariant argument returning a type.

[4]Haskell-like syntax is helpful here, the kind `(TFn [[x :variance :covariant]] Any)` is approximately `* -> *`. A more streamlined syntax is planned for future work.

Listing 5.7: Identity monad using user defined type constructors

```
(ann identity-m (Monad (TFn [[x :variance :covariant]] x)))
```

We can now express the type of the identity monad more abstractly (Listing 5.7). The purpose of the monad seems more apparent just from reading its type. In this case, the fact that identity monad has no effect is reflected by its type constructor returning exactly its argument.

Listing 5.8: Type checked identity monad definition

```
(defmonad identity-m
   "Monad describing plain computations. This monad does in fact
      nothing
    at all. It is useful for testing, for combination with monad
    transformers, and for code that is parameterized with a monad."
  [m-result identity
   m-bind   (ann-form
              (fn m-result-id [mv f]
                (f mv))
              (All [x y]
                [x [x -> y] -> y]))
  ])
```

The final type checked definition for the identity monad is given in Listing 5.8. In this case, just the monadic bind required annotation.

Listing 5.9: Several monad types

```
; Maybe monad
(ann maybe-m (MonadPlusZero
               (TFn [[x :variance :covariant]]
                 (U nil x))))
; Sequence monad (called "list monad" in Haskell)
(ann sequence-m (MonadPlusZero
                  (TFn [[x :variance :covariant]]
                    (Seqable x))))
; State monad
(def-alias State
  (TFn [[r :variance :covariant]
        [s :variance :invariant]]
    [s -> '[r s]]))
(ann state-m (All [s]
               (Monad (TFn [[x :variance :covariant]]
                        (State x s)))))
```

Listing 5.9 shows several monad types. The alias `MonadPlusZero` is identical to `Monad`, except both `:m-zero` and `:m-plus` are defined. The definitions were type checked by adding type annotations in appropriate places, in a similar fashion to the identity monad. A significant obstacle was discovered while type checking monad definitions, related to filtering sequences with negative type information (discussed in Section 4.6).

5.3.2 Monad Transformer Definitions

The initial motivation for adding user defined type constructors to Typed Clojure (as discussed in Section 5.3.1) was to type check monad transformer definitions. Monad transformers are always parameterised by a monad type constructor, and should work for all monad type constructors. Keeping the monad type constructor abstract allows us to determine whether a monad transformer works *for all* monads.

Finally, Listing 5.10 shows the type assigned to the maybe monad transformer definition. The most interesting aspect to notice here is that the monad `m` is kept abstract throughout the type.

Listing 5.10: Maybe monad transformer type

```
(ann maybe-t
     (All [[m :kind (TFn [[x :variance :covariant]] Any)]]
       (Fn
         [(AnyMonad m) -> (MonadPlusZero (TFn [[y :variance
            :covariant]]
                                              (m (U nil y))))]
         [(AnyMonad m) nil -> (MonadPlusZero (TFn [[y :variance
            :covariant]]
                                                   (m (U nil y))))]
         [(AnyMonad m) nil (U ':m-plus-default ':m-plus-from-base)
          -> (MonadPlusZero (TFn [[y :variance :covariant]]
                                 (m (U nil y))))])))
```

5.4 Conduit

Conduits are an advanced form of "pipes" using arrows, a generalisation of monads [Fra12]. *Conduit* is a Clojure library developed by Duey and contributors [5] that supports programming with conduits.

[5]Github home of Conduit: `https://github.com/jduey/conduit`

The library had to be modified and simplified considerably to be able to type check. Many existing conduits had troublesome variable-parameters, reminiscent of the issues of assigning types to `assoc` and `partial` (which is future work, see Section 6.1). In these cases, simplified versions of the conduits were created that took fixed arguments.

An interesting property of the types assigned to conduits (which are internally functions of a single argument) is that they are recursive types. This is not in itself interesting, but Typed Clojure implements conduit types as *type functions* with variance, which required the variance of the conduit types to be known in advance.

Listing 5.11 shows the type constructor for a conduit type, `==>`. Notice the body of `==>` includes a reference to itself in a position where variance must be known in advance. `declare-alias-kind` is used to declare the kind of an alias. Once an alias with an already declared kind is defined with `def-alias`, the defined kind must match the declared kind, otherwise a type error is thrown. (The specifics of the types in Listing 5.11 are not relevant to the rest of this discussion, so they are not explained).

Listing 5.11: Types for Conduits

```
(def-alias Result
  (TFn [[x :variance :covariant]]
    (U nil ;stream is closed
       '[] ;abort/skip
       '[x];consume/continue
       )))

(def-alias Cont
  (TFn [[in :variance :covariant]
        [out :variance :invariant]]
    [(U nil [(Result in) -> (Result out)]) -> (Result out)]))


(declare-alias-kind ==> (TFn [[in :variance :contravariant]
                              [out :variance :invariant]] Any))

(def-alias ==>
  (TFn [[in :variance :contravariant]
        [out :variance :invariant]]
    [in -> '[(U nil (==> in out)) (Cont out out)]]))
```

The resulting port of *Conduit* was unsatisfying for real world use. The simplifications made in order to type check the library reversed any effort that was

invested into *Conduit* to conform to common Clojure idioms. The future work on improving variable-argument types should reveal whether this library can be ported satisfactorily. It is desirable to port this library because implementations and usages of conduits result in very abstract code which static type systems can help verify as correct.

CHAPTER 6

Future Work

6.1 Variable-Arity Polymorphism

It is not clear how to handle several key Clojure functions that accept an even number of arguments. For example, valid usages of `assoc` are `(assoc m k v` $\overrightarrow{kv}$`)`, which takes three arguments and any number of paired arguments.

Strickland, Tobin-Hochstadt, and Felleisen's calculus [STHF09] is insufficient to express this pattern. Devising and integrating types that can express this pattern is set as future work.

6.2 Contracts and Blame

A key part of Typed Racket [TH10] is its contract and blame systems. They enable safe interoperability with untyped Racket by generating runtime contracts in key places based on static types. Typed Racket also includes a sophisticated blame system based on Wadler[WF09] to provide more accurate error reporting that ensures typed modules "can't be blamed".

There are two kinds of interoperability in Typed Clojure that could utilize these features. First, the set of "assumptions" given by the programmer about Java interoperability like those introduced by `non-nil-return` and `nilable-param` (see section 5.1) could be checked by adding runtime contracts. Secondly, when typed namespaces import untyped functions, the static type assigned to the untyped function could be enforced by wrapping the function in a runtime contract. Typed Racket uses this second approach when importing untyped Racket functions, and it results in better error messages.

6.3 Multimethods

Multimethods in Clojure are an idiomatic and often used feature. It would be essential for an optional static type system for Clojure to support multimethods if it is to be of practical use. The major hurdle for type checking multimethod definitions is the interaction between the multimethod dispatch mechanism and occurrence typing.

`clojure.core/isa?` is the core of multimethod dispatch. It is a function that takes two arguments and is `true` if the first argument is a member of the second argument, otherwise `false`. It gets quite complicated quickly even with common usages of `isa?`. Simple calls involving only *Class* objects return true if the left class includes the right class in a Java type relationship, eg. `(isa? Integer Number)` is `true`. For example, `(isa? [Integer Double] [Number Number])` is `true` because the left and right arguments are vectors of classes of the same length that are subtypes by `isa?` pairwise.

It is also unclear whether it is feasible to perform comprehensiveness and ambiguity prediction tests on multimethod usages. I predict that they will be very difficult to pull off satisfactorily because of the "openess" of multimethods, and we will fall back on the runtime errors that multimethods already throw.

6.4 Records

A Clojure record is a composite of datatypes and maps; it is a datatype with fields that can be treated like a map: records support map operations like `get` (lookup), `assoc` (add entries), and `dissoc` (remove entries). One interesting property from the perspective of static typing is that if a entry corresponding to a field of the underlying datatype is removed with `dissoc`, then the resulting type is a plain map: it loses its record type. Correspondingly, a record keeps its type if any other entry is removed.

Records are used frequently in Clojure code, so it is desirable to support them to some degree. Future work is planned to investigate solutions to satisfactorily capture their subtle semantics in a practical way.

6.5 Porting to other Clojure implementations

It would be desirable to port Typed Clojure to other Clojure implementations. Each implementation would bring its own set of challenges with interoperability.

The core of Typed Clojure would preserved with each port changing at least its interoperability with its host platform. Section 4.5 discusses potential problems with the porting process, and how they might be solved.

CHAPTER 7

Conclusion

Whether a language is dynamically typed or statically typed is not always a dividing classification. Recent languages have managed to support the advantages of both styles: the safety of a statically typed language while retaining the idioms found in dynamically typed languages.

This dissertation describes an optional static typing system that attempts to bring the advantages of static typing to the dynamically typed language Clojure, running on the Java Virtual Machine. By reusing many of the design choices made in similar projects like Typed Racket, we are able to design and implement a prototype type system *Typed Clojure* that can statically check many Clojure idioms.

Typed Clojure is intended to be of practical use to Clojure programmers. We show that Typed Clojure helps verify the absence of errors in Clojure code written in sophisticated programming styles by porting most of a wide-spread Clojure library for monadic programming. Similarly, we show that Typed Clojure helps verify code using Java interoperability is correct, and is a useful tool to show misuse of Java's *null*.

There is still significant future work in order to type check all Clojure idioms, but the work already carried out suggests that an optional type system for Clojure like Typed Clojure is both practical and useful tool.

References

[Aba+95] Martn Abadi et al. “Dynamic Typing in Polymorphic Languages”. In: *JOURNAL OF FUNCTIONAL PROGRAMMING* 5 (1995), pp. 92–103.

[AM91] Alexander Aiken and Brian R. Murphy. “Static Type Inference in a Dynamically Typed Language”. In: *In Eighteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, 1991, pp. 279–290.

[BK99] Nick Benton and Andrew Kennedy. “Interlanguage Working Without Tears: Blending SML with Java”. In: *In ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM Press, 1999, pp. 126–137.

[BSc12] Ambrose Bonnaire-Sergeant and contributors. *analyze Github Repository*. 2012. URL: `https://github.com/frenchy64/analyze`.

[Bra+98] Gilad Bracha et al. “Making the future safe for the past: adding genericity to the Java programming language”. In: *SIGPLAN Not.* 33.10 (Oct. 1998), pp. 183–200. ISSN: 0362-1340. DOI: `10.1145/286942.286957`. URL: `http://doi.acm.org/10.1145/286942.286957`.

[Can+89] Peter Canning et al. “F-bounded polymorphism for object-oriented programming”. In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. FPCA ’89. Imperial College, London, United Kingdom: ACM, 1989, pp. 273–280. ISBN: 0-89791-328-0. URL: `http://doi.acm.org/10.1145/99370.99392`.

[CM91] Luca Cardelli and John C. Mitchell. “Operations on records”. In: *Mathematical Structures in Computer Science*. 1991, pp. 3–48.

[CF91] Robert Cartwright and Mike Fagan. “Soft typing”. In: *SIGPLAN Not.* 26.6 (May 1991), pp. 278–292. ISSN: 0362-1340. DOI: `10.1145/113446.113469`. URL: `http://doi.acm.org/10.1145/113446.113469`.

[Chu40] Alonzo Church. “A formulation of the simple theory of types”. In: *Journal of Symbolic Logic* 5 (1940), pp. 56–68.

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. "Functional Characters of Solvable Terms". In: *Mathematical Logic Quarterly* 27.2-6 (1981), pp. 45–58. ISSN: 1521-3870. DOI: 10.1002/malq.19810270205. URL: http://dx.doi.org/10.1002/malq.19810270205.

[Cur34] H. B. Curry. "Functionality in combinatory logic". In: *Proceedings of the National Academy of Sciences* 20 (1934), pp. 584–590.

[Dav05] Rowan Davies. "Practical refinement-type checking". AAI3168521. PhD thesis. Pittsburgh, PA, USA, 2005. ISBN: 0-542-04587-7.

[Fag91] Mike Fagan. "Soft typing: an approach to type checking for dynamically typed languages". UMI Order No. GAX91-36021. PhD thesis. Houston, TX, USA, 1991.

[Fel09] Matthias Felleisen. *From Soft Scheme to Typed Scheme: 20 years of Scripts-to-Program conversion*. 2009. URL: http://www.ccs.neu.edu/home/matthias/Presentations/STOP/stop.pdf.

[FF97] Cormac Flanagan and Matthias Felleisen. "Componential set-based analysis". In: *ACM Transactions on Programming Languages and Systems*. ACM Press, 1997, pp. 235–248.

[Fra12] Yesod Web Framework. *Conduit*. 2012. URL: http://www.yesodweb.com/book/conduits.

[FP91] Tim Freeman and Frank Pfenning. "Refinement types for ML". In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. PLDI '91. Toronto, Ontario, Canada: ACM, 1991, pp. 268–277. ISBN: 0-89791-428-7. DOI: 10.1145/113445.113468. URL: http://doi.acm.org/10.1145/113445.113468.

[Goo12a] Google. *Dart Homepage*. 2012. URL: http://www.dartlang.org/.

[Goo12b] Google. *Typed Clojure Project Description for Google Summer of Code 2012*. 2012. URL: http://www.google-melange.com/gsoc/project/google/gsoc2012/ambrosebs/10001.

[Har12] Robert Harper. "Practical Foundations for Programming Languages". Draft available from http://www.cs.cmu.edu/~rwh/plbook/book.pdf. 2012.

[HP91] Robert Harper and Benjamin Pierce. "A Record Calculus Based on Symmetric Concatenation". In: *In Proc. of the ACM Symp. on Principles of Programming Languages*. ACM, 1991, pp. 131–142.

[Hay91] Susumu Hayashi. "Singleton, union and intersection types for program extraction". In: *Theoretical Aspects of Computer Software*. Ed. by Takayasu Ito and AlbertR. Meyer. Vol. 526. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1991, pp. 701–730. ISBN: 978-3-540-54415-9. DOI: 10.1007/3-540-54415-1_71. URL: http://dx.doi.org/10.1007/3-540-54415-1_71.

[Her10] David Herman. "A Theory of Typed Hygienic Macros". PhD thesis. Boston, MA: Northeastern University, May 2010.

[Hic08] Rich Hickey. "The Clojure programming language". In: *Proceedings of the 2008 symposium on Dynamic languages*. DLS '08. Paphos, Cyprus: ACM, 2008, 1:1–1:1. ISBN: 978-1-60558-270-2. URL: http://doi.acm.org/10.1145/1408681.1408682.

[Hc12] Rich Hickey and contributors. *algo.monads*. 2012. URL: https://github.com/clojure/algo.monads.

[HP99] Haruo Hosoya and Benjamin C. Pierce. *How Good is Local Type Inference?* Tech. rep. University of Pennsylvania, 1999.

[Mey92] Bertrand Meyer. "Applying design by contract". In: *IEEE Computer* 25 (1992), pp. 40–51.

[Mic12] Microsoft. *Typescript Homepage*. 2012. URL: http://www.typescriptlang.org/.

[MC02] Todd Millstein and Craig Chambers. "Modular Statically Typed Multimethods". In: *Information and Computation*. Springer-Verlag, 2002, pp. 279–303.

[Mil+97] Robin Milner et al. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0262631814.

[MPO08] Adriaan Moors, Frank Piessens, and Martin Odersky. "Generics of a higher kind". In: *SIGPLAN Not.* 43.10 (Oct. 2008), pp. 423–438. ISSN: 0362-1340. DOI: 10.1145/1449955.1449798. URL: http://doi.acm.org/10.1145/1449955.1449798.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. 1st. USA: Artima Incorporation, 2008. ISBN: 0981531601, 9780981531601.

[OW97] Martin Odersky and Philip Wadler. "Pizza into Java: Translating theory into practice". In: *In Proc. 24th ACM Symposium on Principles of Programming Languages*. ACM Press, 1997, pp. 146–159.

[OZZ01] Martin Odersky, Christoph Zenger, and Matthias Zenger. "Colored local type inference". In: *SIGPLAN Not.* 36.3 (Jan. 2001), pp. 41–53. ISSN: 0362-1340. URL: http://doi.acm.org/10.1145/373243.360207.

[Ode+06] Martin Odersky et al. *An overview of the Scala programming language (second edition).* Tech. rep. EPFL Lausanne, Switzerland, 2006.

[Ora12] Oracle. *What Is an Interface?* 2012. URL: http://docs.oracle.com/javase/tutorial/java/concepts/interface.html.

[Pfe08] Frank Pfenning. "Church and Curry: Combining Intrinsic and Extrinsic Typing". In: *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday.* Vol. 17. College Publications, 2008.

[Pie97] Benjamin C. Pierce. *Bounded Quantification with Bottom.* Tech. rep. Computer Science Department, Indiana University, 1997.

[Pie02] Benjamin C. Pierce. *Types and programming languages.* Cambridge, MA, USA: MIT Press, 2002. ISBN: 0-262-16209-1.

[PT00] Benjamin C. Pierce and David N. Turner. "Local type inference". In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), pp. 1–44. ISSN: 0164-0925. URL: http://doi.acm.org/10.1145/345099.345100.

[Rey81] John C. Reynolds. "The essence of ALGOL". In: *Proceedings of the International Symposium on Algorithmic Languages.* 1981, pp. 345–372.

[Rey96] John C. Reynolds. *Design of the Programming Language Forsythe.* Tech. rep. Carnegie Mellon University, 1996.

[Rey02] John C. Reynolds. "What do types mean? - From intrinsic to extrinsic semantics". In: *Essays on Programming Methodology.* Springer-Verlag, 2002.

[Ros07] Andreas Rossberg. "Typed Open Programming - A higher-order, typed approach to dynamic modularity and distribution". PhD thesis. Saarland University, 2007.

[Sim10] Marlow Simon. *Haskell 2010 Language Report.* 2010. URL: http://www.haskell.org/onlinereport/haskell2010/.

[SA+12] Vincent St-Amour et al. “Typing the numeric tower”. In: *Proceedings of the 14th international conference on Practical Aspects of Declarative Languages*. PADL’12. Philadelphia, PA: Springer-Verlag, 2012, pp. 289–303. ISBN: 978-3-642-27693-4. DOI: 10.1007/978-3-642-27694-1_21. URL: http://dx.doi.org/10.1007/978-3-642-27694-1_21.

[STHF09] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. “Practical Variable-Arity Polymorphism”. In: *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. ESOP ’09. York, UK: Springer-Verlag, 2009, pp. 32–46. ISBN: 978-3-642-00589-3. URL: http://dx.doi.org/10.1007/978-3-642-00590-9_3.

[TH10] Sam Tobin-Hochstadt. “Typed Scheme: From Scripts to Programs”. PhD thesis. Northeastern University, 2010.

[TH12] Sam Tobin-Hochstadt. *Personal communication*. 2012.

[THF08] Sam Tobin-Hochstadt and Matthias Felleisen. “The design and implementation of typed scheme”. In: *SIGPLAN Not.* 43.1 (Jan. 2008), pp. 395–406. ISSN: 0362-1340. URL: http://doi.acm.org/10.1145/1328897.1328486.

[THF10] Sam Tobin-Hochstadt and Matthias Felleisen. “Logical types for untyped languages”. In: *SIGPLAN Not.* 45.9 (Sept. 2010), pp. 117–128. ISSN: 0362-1340. URL: http://doi.acm.org/10.1145/1932681.1863561.

[Wad95] Philip Wadler. “Monads for Functional Programming”. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK, UK: Springer-Verlag, 1995, pp. 24–52. ISBN: 3-540-59451-5. URL: http://dl.acm.org/citation.cfm?id=647698.734146.

[WF09] Philip Wadler and Robert Bruce Findler. “Well-Typed Programs Can’t Be Blamed”. In: *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. ESOP ’09. York, UK: Springer-Verlag, 2009, pp. 1–16. ISBN: 978-3-642-00589-3. DOI: 10.1007/978-3-642-00590-9_1. URL: http://dx.doi.org/10.1007/978-3-642-00590-9_1.

[Wan89] Mitchell Wand. “Type Inference for Record Concatenation and Multiple Inheritance”. In: *Fourth Annual Symposium on Logic in Computer Science*. 1989, pp. 92–97.

[WC97] Andrew K. Wright and Robert Cartwright. “A practical soft type system for scheme”. In: *ACM Trans. Program. Lang. Syst.* 19.1 (Jan. 1997), pp. 87–152. ISSN: 0164-0925. DOI: 10.1145/239912.239917. URL: http://doi.acm.org/10.1145/239912.239917.

APPENDIX A

Heterogeous map type theory prototype

A.1 Operational Semantics

Syntax of Terms

$e ::= (c\,\overrightarrow{e}) \mid \{\overrightarrow{e\ e}\}$	Expressions
$v ::= k \mid nil \mid \{\overrightarrow{v\ v}\}$	Values
$c ::= assoc \mid dissoc \mid get$	Constants

Evaluation Contexts

$E ::= [] \mid (c\ \overrightarrow{v}\ E\ \overrightarrow{e}) \mid \{\overrightarrow{v\ v}\ E\ e\ \overrightarrow{e\ e}\} \mid \{\overrightarrow{v\ v}\ v\ E\ \overrightarrow{e\ e}\}$	Evaluation Contexts

Operational Semantics

$$\frac{v_1 = \{\overrightarrow{v\ v}\} \qquad v_4 = v_1\ \textit{with entry}\ v_2\ \textit{to}\ v_3}{(\textit{assoc}\ v_1\ v_2\ v_3) \hookrightarrow v_4}\ \text{E-Assoc}$$

$$\frac{v_1 = \{\overrightarrow{v\ v}\} \qquad v_3 = v_1\ \textit{without entry indexed by}\ v_2}{(\textit{dissoc}\ v_1\ v_2) \hookrightarrow v_3}\ \text{E-Dissoc}$$

$$\frac{v_1 = \{\overrightarrow{v\ v}\} \qquad v_2\ v_3\ \textit{in}\ v_1}{(\textit{get}\ v_1\ v_2) \hookrightarrow v_3}\ \text{E-GetMapExist}$$

$$\frac{v_1 = \{\overrightarrow{v\ v}\} \qquad v_1\ \textit{has no entry with key}\ v_2 \qquad v_3 = \textit{nil}}{(\textit{get}\ v_1\ v_2) \hookrightarrow v_3}\ \text{E-GetMapNotExist}$$

Type Syntax

$$\begin{array}{rcl} T ::= & \textbf{Any} \mid \textbf{Nothing} \mid \textbf{nil} \mid k \mid \{\overrightarrow{k\ T}\} \mid (\textbf{IPersistentMap}\ T\ T) \\ & \mid (\vee\ \overrightarrow{T}) \mid (\wedge\ \overrightarrow{T}) \end{array}$$

Core Type Rules

T-AssocHMap

$$\frac{\Gamma \vdash e_m : T_m \qquad \Gamma \vdash e_k : k \qquad \Gamma \vdash e_v : T_v \qquad \vdash T_m <: \mathbf{\{\ \}}}{\Gamma \vdash (assoc\ e_m\ e_k\ e_v) : (update\_hmap\ T_m\ k\ T_v)}$$

T-AssocPromote

$$\frac{\Gamma \vdash e_m : T_m \qquad \Gamma \vdash e_k : T_k \qquad \Gamma \vdash e_v : T_v \qquad \vdash T_m <: \mathbf{(IPersistentMap\ Any\ Any)} \qquad T = (promote\_hmap\ T_m\ T_k\ T_v)}{\Gamma \vdash (assoc\ e_m\ e_k\ e_v) : T}$$

Type Metafunctions

$(update\_hmap\ \{\overrightarrow{T_k\ T_v}\}\ k_1\ T_1) = \{\overrightarrow{T_k\ T_v}\ k_1\ T_1\}$
$(update\_hmap\ (\wedge\ \overrightarrow{T})\ T_k\ T_v) = (\wedge\ \overrightarrow{(update\_hmap\ T\ T_k\ T_v)})$
$(update\_hmap\ (\vee\ \overrightarrow{T})\ T_k\ T_v) = (\vee\ \overrightarrow{(update\_hmap\ T\ T_k\ T_v)})$

$(promote\_hmap\ \{\overrightarrow{T_k\ T_v}\}\ T_{kn}\ T_{vn}) = (\mathbf{IPersistentMap}\ (\vee\ \overrightarrow{T_k}\ T_{kn})\ (\vee\ \overrightarrow{T_v}\ T_{vn}))$
$(promote\_hmap\ (\mathbf{IPersistentMap}\ T_k\ T_v)\ T_{kn}\ T_{vn}) = (\mathbf{IPersistentMap}\ (\vee\ T_k\ T_{kn})\ (\vee\ T_v\ T_{vn}))$

APPENDIX B

Dissertation Proposal

Background

Dynamically typed languages (also known as monotyped) are designed to be convenient for writing programs quickly, and aspire to get out of the programmer's way as much as possible. When programs grow large and stabilize, some features of static languages are missed, specifically static type checking.

Tobin-Hochstadt notes that "untyped scripts are difficult to maintain over the long run" [TH10] because types contain valuable design information. He developed Typed Scheme [TH10] to safely and incrementally port existing Scheme code, a dynamic language, to Typed Scheme, a static language.

Clojure is a dynamically typed, functional language with implementations for the Java Virtual Machine and Common Language Runtime, and compiling to Javascript. Clojure is also a Lisp, which makes it a good candidate to test the generality of ideas developed for Typed Scheme [TH10].

Aim

My goal is to develop a prototype optional static type system for Clojure, eventually intended for practical use.

It will be based on the lessons learnt throught the development of Typed Scheme, and as a response to Tobin-Hochstadt's [TH10] suggestion to add type systems to other existing dynamic languages.

There are several challenges to creating a satisfactory type system for Clojure.

Multimethods play a significant role in Clojure, and a satisfactory type system for Clojure should understand them to some degree. There is some experience statically typing multimethods [MC02], but this is a challenge and may not be

addressed in the timeframe proposed, beyond initial consideration of how it would fit with the rest of the system.

Clojure's core library includes variable-arity functions which are not easily typed with current static type systems. Providing satisfactory types for functions like Clojure's 'map', 'filter', and 'reduce' require support for non-uniform variable arity polymorphism. Strickland, Tobin-Hochstadt and Felleisen [STHF09]. describe their approach for non-uniform variable arity polymorphism, as used in Typed Racket. At a minimum, the proposed project will include broadly identifiying what new issues arise when adapting this approach to Clojure, with a prototype design if these issues are not too major.

Occurence typing is type checking technique developed for Typed Scheme [THF08] and improved for Typed Racket [THF10]. It helps the type checker understand common programming idioms with minimal type annotations. The proposed project will include a comparison of these approaches in the context of Clojure, and a prototype design if a satisfactory approach is designed and no major issues are identifed.

Ensuring vigorous type safety is an important aspect of Typed Scheme [TH10], especially when interacting between untyped and typed modules. I do not expect to concentrate on every combination of cross-module interaction. In particular, safely using typed code from untyped code will not be designed. Safe interaction between typed modules however is needed for a practical type checker. I will attempt to design a viable strategy or identify issues that require further consideration.

Method

The milestones for this project are broken into release milestones for the prototype library.

My goal is to finish all features listed in releases below and to begin work on the challenges, listed last.

For each of these it's not yet fully clear what novel issues may arise in adapting existing techniques to Clojure. For each item below the project will involve either implemention or identification of the novel issues that make implementation difficult. The classification below reflects the level of uncertainity regarding such issues.

0.1

- Union types

- Basic Local Type Inference algorithm
- Fixed arity function types
- Typed deftype (class definitions)
- Uniform variable-arity function types
- Enough functions annotated from clojure.core for proof-of-concept
- Bounded Type variables

Challenges
- Devise strategy for type inference (eg. occurence typing)
- Manage interactions between typed namespaces
- Mutable reference types
- Non-uniform variable arity polymorphism
- Typing Multimethods
- Fine Grained Hash-Map Types

Software and Hardware Requirements

Linux environment with Java, git, and maven installed. No issues are anticipated with regards to access to these.