Practical Optional Types for Clojure

Ambrose Bonnaire-Sergeant
Indiana University
abonnair@indiana.edu

Rowan Davies
University of Western Australia
rowan.davies@uwa.edu.au

Sam Tobin-Hochstadt
Indiana University
samth@indiana.edu

Abstract

Typed Clojure is an optional type system for Clojure, a dynamic language in the Lisp family that targets the JVM. Typed Clojure's type system build on the design of Typed Racket, repurposing in particular *occurrence typing*, an approach to statically reasoning about predicate tests. However, in adapting the type system to Clojure, changes and extensions are required to accommodate additional language features and idioms used by Clojure programmers.

In this paper, we describe Typed Clojure and present these type system extensions, focusing on three features widely used in Clojure. First, Java interoperability is central to Clojure's mission but introduces challenges such as ubiquitous `null`; Typed Clojure handles Java interoperability while ensuring the absence of null-pointer exceptions in typed programs. Second, Clojure programmers idiomatically use immutable dictionaries for data structures; Typed Clojure handles this in the type system with multiple forms of heterogeneous dictionary types. Third, multimethods provide extensible operations, and their Clojure semantics turns out to have a surprising synergy with the underlying occurrence typing framework.

We provide a formal model of the Typed Clojure type system incorporating these and other features, with a proof of soundness. Additionally, Typed Clojure is now in use by numerous corporations and developers working with Clojure, and we report on experience with the system and its lessons for the future.

1. Clojure with static typing

The popularity of dynamically-typed languages in software development, combined with a recognition that types often improve programmer productivity, software reliability, and performance, has led to the recent development of a wide variety of optional and gradual type systems aimed at checking existing programs written in existing languages. These include Microsoft's TypeScript for JavaScript, Facebook's Hack for PHP and Flow for JavaScript, and MyPy for Python among the optional systems, and Typed Racket, Reticulated Python, and GradualTalk among gradually-typed systems.[1]

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

We synthesize these lessons to present *Typed Clojure*, an optional type system for Clojure. Typed Clojure builds on the core type checking approach of Typed Racket, an existing gradual type system for Racket. However, Typed Clojure extends this basic framework in multiple ways to accommodate the unique idioms and features of Clojure, producing an expressive synthesis of ideas and demonstrating a surprising coincidence between multiple dispatch in Clojure and Typed Racket's occurrence typing framework.

```
(ann parent ['{:file (U nil File)} -> (U nil Str)])
(defn parent [{^File f :file}]
  (if f (.getParent f) nil))
```

Figure 1. A simple Typed Clojure program

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

Figure 1 presents a simple program demonstrating many aspects of our system, from simple type annotations to explicit handling of Java's `null` (written `nil`) in interoperation, as well as an extended form of occurrence typing and Clojure's *type hints*, which are central to Typed Clojure's approach to interoperability.

The `parent` function has the type

```
['{:file (U nil File)} -> (U nil Str)]
```

which means that it takes a hash table whose `:file` key maps to either `nil` or a `File`, and it produces either `nil` or a `String`. The `parent` function uses the `:file` keyword as an accessor to get the file, checks that it isn't `nil`, and then obtains the parent by making a Java method call. The annotation `^File f` is a type hint on `f`, which instructs the Clojure compiler (running prior to Typed Clojure typechecking) to statically resolve the `getParent` call to `File`'s `getParent` method with signature `String getParent();`, rather than using reflection at runtime.

In the remainder of this paper, we describe how Typed Clojure's central innovations, including Java interoperability, multimethods, and heterogeneously-typed immutable maps, enable this example and many others. We begin with an example-driven presentation of the main type system features in Section 2. We then incrementally present a core calculus for Typed Clojure covering all of these features together in Section 3 and prove type soundness (Section 4). We then discuss the full implementation of Typed Clojure, dubbed `core.typed`, which extends the formal model in many ways, and the experience gained from its use in Section 5. Finally, we discuss related work and conclude.

[1] We reserve the term "gradual typing" for systems such as Typed Racket which soundly interoperate between typed and untyped code; systems like Typed Clojure or TypeScript which do not enforce type invariants we describe as "optionally typed".

2. Overview of Typed Clojure

We now begin a tour of the central features of Typed Clojure, beginning with Clojure itself. In our presentation, we will make use of the full Typed Clojure system to illustrate the key type system ideas, before studying the core features in detail in section 3.

2.1 Clojure

Clojure (Hickey 2008) is a Lisp built to run on the Java Virtual Machine with exemplary support for concurrent programming and immutable data structures. It emphasizes mostly-functional programming, restricting imperative updates to a limited set of structures which have specific thread synchronization behaviour. By default, it provides fast implementations of immutable lists, vectors, and hash tables, which are used for most data structures, although it also provides means for defining new records.

One of Clojure's primary advantages is easy interoperation with existing Java libraries. It automatically generates appropriate JVM bytecode to make Java method and constructor calls, and treats Java values as any other Clojure value. However, this smooth interoperability comes at the cost of pervasive `null`, which leads to the possibility of null pointer exceptions—a drawback we address in Typed Clojure.

2.2 Clojure Syntax

We describe new syntax as they appear in each example, but we also include the essential basics of Clojure syntax.

`nil` is exactly Java's `null`. Parentheses indicate *applications*, brackets delimit *vectors*, braces delimit *hash-maps* and double quotes delimit *Java strings*. *Symbols* begin with an alphabetic character, and a colon prefixed symbol like `:a` is a *keyword*.

Commas are always *whitespace*.

2.3 Typed Racket and occurrence typing

Tobin-Hochstadt and Felleisen (2010) presented Typed Racket with occurrence typing, a technique for deriving type information from conditional control flow. They introduced the concept of occurrence typing with the following example.

```
#lang typed/racket

(lambda ([x : (U #f Number)])
  (if (number? x) (add1 x) 0))
```

This function takes a value that is either *#f* or a number, represented by an *untagged* union type. The 'then' branch has an implicit invariant that `x` is a number, which is automatically inferred with occurrence typing and type checked without further annotations.

We chose to build on the ideas and implementation of Typed Racket to implement a type system targeting Clojure for several reasons. Initially, the similarities between Racket and Clojure drew us to investigate the effectiveness of repurposing occurrence typing for a Clojure type system—both languages share a Lisp heritage, similar standard functions (for instance `map` in both languages is variable-arity) and idioms. While Typed Racket is gradually typed and has sophisticated dynamic semantics for cross-language interaction, we chose to first implement the static semantics with the hope to extend Typed Clojure to be gradually typed at a future date. Finally, Typed Racket's combination of bidirectional checking and occurrence typing presents a successful model for type checking dynamically typed programs without compromising soundness, which is appealing over success typing (Lindahl and Sagonas 2006) which cannot prove strong properties about programs and soft typing (Cartwright and Fagan 1991) which has proved too complicated in practice.

Here is the above program in Typed Clojure.

Example 1

```
(ns demo.eg1
  (:refer-clojure :exclude [fn])
  (:require [clojure.core.typed :refer [fn U Num]]))

(fn [x :- (U nil Num)]
  (if (number? x) (inc x) 0))
```

This is a regular Clojure file compiled with the Clojure compiler, with the `ns` form declaring a *namespace* for managing var and class imports. Here `:require` declares a runtime dependency on `clojure.core.typed`, Typed Clojure's core namespace, and `:refer` brings a collection of vars into scope by name. The `:refer-clojure :exclude` option unmaps core vars from the current namespace—here we unmap `clojure.core/fn`, which creates a function from a parameter vector and a body expression, and import a typed variant of `fn` that supports type annotations.

The typed `fn` supports optional annotations by adding `:-` and a type after a parameter position or binding vector to annotate parameter types and return types respectively. Typed Clojure provides a `check-ns` function to type checks the current namespace. `number?` is a Java `instanceof` test of `java.lang.Number`. As in Typed Racket, `U` creates an *untagged union* type, which can take any number of types.

Typed Clojure can already check all of the examples in Tobin-Hochstadt and Felleisen (2010)—the rest of this section describes the extensions necessary to check Clojure code.

2.4 Exceptional control flow

Along with conditional control flow, Clojure programmers rely on *exceptions* to assert type-related invariants.

Example 2

```
(fn [x :- (U nil Num)]
  (do (if (number? x) nil (throw (Exception.)))
      (inc x)))
```

The `do` form sequences two expressions returning the latter, `throw` corresponds to Java's `throw` and `(class. args*)` is the syntax for Java constructors—that is a class name with a dot suffix as the operator followed the arguments to the constructor.

In this example a `throw` expression guards `(inc x)`, the increment function for numbers, from being evaluated if `x` is `nil`, preventing a possible null-pointer exception.

To check this example, occurrence typing automatically assumes `x` is a number when checking the second `do` subexpression based on the first subexpression. [2] We model this formally (section 3.1) and prove null-pointer exceptions are impossible in typed code (section 4).

2.5 Heterogeneous hash-maps

Hash-maps with keyword keys play a major role in Clojure programming. HMap types model the most common usages of keyword maps.

Example 3

```
(defalias Expr
  (U '{:op ':if, :test Expr, :then Expr, :else Expr}
     '{:op ':do, :left Expr, :right Expr}
     '{:op ':const, :val Num}))

(defn an-exp [] :- Expr
  (let [v {:op :const, :val 1}]
    {:op :do, :left v, :right v}))
```

[2] See `https://github.com/typedclojure/examples` for full examples. From here we omit `ns` forms.

The `defn` macro defines a top-level function, with syntax like the typed `fn`. The function `an-exp` is verified to return an `Expr`.

The `defalias` macro defines a type abbreviation which can reference itself recursively. Here `defalias` defines `Expr` that describes the structure of a recursively-defined AST as a union of HMaps. A quoted keyword in a type, such as `':if`, is a singleton type that contains just the keyword. A type that is a quoted map like `'{:op ':if}` is a HMap type with a fixed number of keyword entries of the specified types known to be *present*, zero entries known to absolutely be *absent*, and an infinite number of *unknown* entries entries. Since only keyword keys are allowed, they do not require quoting.

HMaps in Practice The next example is extracted from a production system at CircleCI, a company with a large production Typed Clojure system (section 5.3 presents a case study).

Example 4

```
(defalias RawKeyPair
  (HMap :mandatory {:public-key RawKey,
                    :private-key RawKey},
        :complete? true))
(defalias EncKeyPair
  (HMap :mandatory {:public-key RawKey,
                    :enc-private-key EncKey},
        :complete? true))

(ann enc-keypair [RawKeyPair -> EncKeyPair])
(defn enc-keypair [kp]
  (assoc (dissoc kp :private-key)
    :enc-private-key (encrypt (:private-key kp))))
```

`enc-keypair` takes an unencrypted keypair and returns an encrypted keypair by dissociating the raw `:private-key` entry with `dissoc` and associating an encrypted private key as `:enc-private-key` on an immutable map with `assoc`. The expression `(:private-key kp)` shows that keywords are also functions that look themselves up in a map returning the associated value or `nil` if the key is missing. Since `EncKeyPair` is `:complete?`, Typed Clojure enforces the return type does not contain an entry `:private-key`, and would complain if the `dissoc` operation forgot to remove it.

Example 5 is like Example 4 except the `:absent-keys` HMap option is used instead of `:complete?`, which takes a *set literal* of keywords that do not appear in the map, written with #-prefixed braces. The syntax `(fn [{pkey :private-key, :as kp}] ...)` aliases `kp` to the first argument and `pkey` to `(:private-key m)` in the function body.

Example 5

```
(defalias RawKeyPair
  (HMap :mandatory {:public-key RawKey,
                    :private-key RawKey}))
(defalias EncKeyPair
  (HMap :mandatory {:public-key RawKey,
                    :enc-private-key EncKey},
        :absent-keys #{:private-key}))

(ann enc-keypair [RawKeyPair -> EncKeyPair])
(defn enc-keypair [{pkey :private-key, :as kp}]
  (assoc (dissoc kp :private-key)
    :enc-private-key (encrypt pkey)))
```

Since this example enforces that `:private-key` must not appear in a `EncKeyPair` Typed Clojure would still complain if we forgot to `dissoc :private-key` from the return value. Now, however we could stash the raw private key in another entry like `:secret-key` which is not mentioned by the partial HMap `EncKeyPair` without Typed Clojure noticing.

Branching on HMaps Finally, testing on HMap properties allows us to refine its type down branches. `dec-map` takes an `Expr`, traverses to its nodes and decrements their values by `dec`, then builds the `Expr` back up with the decremented nodes.

Example 6

```
 1  (ann dec-leaf [Expr -> Expr])
 2  (defn dec-leaf [m]
 3    (if (= (:op m) :if)
 4      {:op :if,
 5       :test (dec-leaf (:test m)),
 6       :then (dec-leaf (:then m)),
 7       :else (dec-leaf (:else m))}
 8      (if (= (:op m) :do)
 9        {:op :do,
10         :left  (dec-leaf (:left m)),
11         :right (dec-leaf (:right m))}
12        {:op :const,
13         :val (dec (:val m))})))
```

If we go down the then branch (line 4), since `(= (:op m) :if)` is true we remove the `:do` and `:const` Expr's from the type of `m` (because their respective `:op` entries disagrees with `(= (:op m) :if)`) and we are left with an `:if` Expr. On line 8, we instead strike out the `:if` Expr since it contradicts `(= (:op m) :if)` being false. Line 9 know we can remove the `:const` Expr from the type of `m` because it contradicts `(= (:op m) :do)` being true, and we know `m` is a `:do` Expr. Line 12 we strike out `:do` because `(= (:op m) :do)` is false, so we are left with `m` being a `:const` Expr.

Section 3.3 discusses how this automatic reasoning is achieved.

2.6 Java interoperability

Clojure supports interoperability with Java, including the ability to call constructors, methods and access fields.

```
(fn [f] (.getParent f))
```

Calls to Java methods and fields have prefix notation like `(.method target args*)` and `(.field target)` respectively, with method and field names prefixed with a dot and methods taking some number of arguments.

Unlike Java, Clojure is dynamically typed. We have no type information about `f` but we still need to pick a method to call. The Clojure compiler delegates the choice to runtime using *Java reflection*. Unfortunately reflection is slow and unpredictable, so Clojure supports *type hints* to help eliminate it where possible,

```
(fn [^File f] (.getParent f))
```

Symbols support *metadata*—the syntax `^File f` is a single expression that is a symbol `f` with metadata `{:tag File}`. In binding positions like `(fn [^File f] ...)` syntactic occurrences preserve metadata.

The Clojure compiler uses the type hint to statically resolve the method call to the `public String getParent()` method of `java.io.File`. The call to `getParent` is unambiguous at runtime but type checking fails—Typed Clojure considers `f` to be of type `Any`, which is unsafe to use as the target of even a resolved method. If instead we annotate the function parameter with type `File`, we get a static type error.

```
(fn [f :- File] (.getParent f)) ;; type error
```

Typed Clojure disallows reflection in typed code so we must add back the type hint to obtain a well-typed expression.

Example 7

```
(fn [^File f :- File] (.getParent f))
```

A type hint does not correspond to a static property so they are ignored by Typed Clojure. Type hints are still required, however, as Typed Clojure has no mechanism to provide optimisations (discussed in Section 5).

Example 8

```
(defn parent [^File f :- (U nil File)]
  (if f (.getParent f) nil))
```

Typed Clojure and Java treat `null` differently. In Clojure, where it is known as `nil`, Typed Clojure assigns it an explicit type called `nil`. In Java `null` is implicitly a member of any reference type. This means the Java static type `String` is equivalent to `(U nil String)` in Typed Clojure.

Reference types in Java are nullable, so to guarantee a method call does not leak `null` into a Typed Clojure program we must assume methods can return `nil`.

Example 9

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

In contrast, JVM invariants guarantee that constructors cannot return `null`, so we are safe to assume constructors are non-nullable.

Example 10

```
(fn [^String s :- String] :- File
  (File. s))
```

By default Typed Clojure conservatively assumes method and constructor arguments to be *non-nullable*, but can be configured globally for particular positions if needed.

2.7 Multimethods

A multimethod in Clojure is a function with a *dispatch function* and a *dispatch table* of methods. Multimethods are created with `defmulti`.

```
(ann path [Any -> (U nil String)])
(defmulti path class)
```

The multimethod `path` has type `[Any -> (U nil String)]`, an initially empty *dispatch table* and *dispatch function* `class`, a function that returns the class of its argument or `nil` if passed `nil`.

We can use `defmethod` to install a method to `path`.

```
(defmethod path String [x] x)
```

Now the dispatch table maps the *dispatch value* `String` to the function `(fn [x] x)`. We add another method which maps `File` to the function `(fn [^File x] (.getPath x))` in the dispatch table.

```
(defmethod path File [^File x] (.getPath x))
```

After installing both methods, the call

```
(path (File. "dir/a"))
```

dispatches to the second method we installed because

```
(isa? (class "dir/a") String)
```

is true, and finally returns

```
((fn [^File x] (.getPath x)) "dir/a").
```

The `isa?` function first tries an equality check on its arguments, then if that fails and both arguments are classes a subclassing check is returned.

```
(isa? :a :a) ;=> true
(isa? Keyword Object) ;=> true
```

We include the above sequence of definitions as Example 11.

Example 11

```
(ann path [Any -> (U nil String)])
(defmulti path class)
(defmethod path String [x] x)
(defmethod path File [^File x] (.getPath x))

(path "dir/a") ;=> "a"
```

Typed Clojure does not predict if a runtime dispatch will be successful—`(path :a)` type checks because `:a` agrees with the parameter type `Any`, but throws an error at runtime.

HMap dispatch The flexibility of `isa?` is key to the generality of multimethods. In Example 12 we dispatch on the `:op` key of our HMap AST `Expr`. Since keywords are functions that look themselves up in their argument, we simply use `:op` as the dispatch function.

Example 12

```
(ann inc-leaf [Expr -> Expr])
(defmulti inc-leaf :op)
(defmethod inc-leaf :if [{tt :test, t :then, e :else}]
  {:op :if,
   :test (inc-leaf tt),
   :then (inc-leaf t),
   :else (inc-leaf e)})
(defmethod inc-leaf :do [{l :left, r :right}]
  {:op :do,
   :left  (inc-leaf l),
   :right (inc-leaf r)})
(defmethod inc-leaf :const [{v :val}]
  {:op :const,
   :val (inc v)})
```

`inc-map` is like Example 6 except the nodes are incremented. The reasoning is similar, except we only consider one branch (the current method) by locally considering the current *dispatch value* and reasoning about how it relates to the *dispatch function*. For example, in the `:do` method we learn the `:op` key is a `:do`, which narrows our argument type to the `:do` Expr, and similarly for the `:if` and `:const` methods.

Multiple dispatch `isa?` is special with vectors—vectors of the same length recursively call `isa?` on the elements pairwise.

```
(isa? [Keyword Keyword] [Object Object]) ;=> true
```

Example 13 simulates multiple dispatch by dispatching on a vector containing the class of both arguments. `open` takes two arguments which can be strings or files and returns a new file that concatenates their paths.

We call three different `File` constructors, each known at compile-time via type hints. Multiple dispatch follows the same kind of reasoning as Example 12, except we update multiple bindings simultaneously.

2.8 Final example

Example 14 combines everything we will cover for the rest of the paper: multimethod dispatch, reflection resolution via type hints, Java method and constructor calls, conditional and exceptional flow reasoning, and HMaps.

We dispatch on `:p` to distinguish the two cases of `FSM`—for example on `:F` we know the `:file` is a file. The body of the first method uses type hints to resolve reflection and conditional control

Example 13

```
(defalias FS (U File String))

(ann open [FS FS -> File])
(defmulti open (fn [l r]
                 [(class l) (class r)]))
(defmethod open [File File] [^File f1, ^File f2]
  (let [s (.getPath f2)]
    (do (if (string? s) nil (throw (Exception.)))
        (File. f1 s))))
(defmethod open [String String] [s1 s2]
  (File. (str s1 "/" s2)))
(defmethod open [File String] [^File s1, ^String s2]
  (File. s1 s2))

(open (File. "dir") "a")         ;=> #<File dir/a>
(open "dir" "a/b")               ;=> #<File dir/a/b>
(open (File. "a/b") (File. "c")) ;=> #<File a/b/c>
```

Example 14

```
(defalias FSM
  (U '{:p ':F :file (U nil File)}
     '{:p ':S :str (U nil String)}))

(ann maybe-parent [FSM -> (U nil Str)])
(defmulti maybe-parent :p)
(defmethod maybe-parent :F [{file :file :as m}]
  (if (:file m) (.getParent ^File file) nil))
(defmethod maybe-parent :S [{^String str :str}]
  (do (if str nil (throw (Exception.)))
      (.getParent (File. str))))

(maybe-parent {:p :S :str "dir/a"}) ;=> "dir"
(maybe-parent {:p :F :file (File. "dir/a")});=> "dir"
(maybe-parent {:p :F :file nil}) ;=> nil
```

Figure 2. Multimethod Examples

flow to prove null-pointer exceptions are impossible. The second method is similar except it uses exceptional control flow.

3. A Formal Model of λ_{TC}

Now that we have demonstrated the core features Typed Clojure provides, we link them together in a formal model called λ_{TC}. Our presentation will start with a review of occurrence typing (Tobin-Hochstadt and Felleisen 2010). Then for the rest of the section we incrementally add each novel feature of Typed Clojure to the formalism, interleaving presentation of syntax, typing rules, operational semantics and subtyping.

The first insight about occurrence typing is that logical formulas can be used to represent type information about our programs by relating parts of the runtime environment to types via propositional logic. *Type Propositions* ψ make assertions like "variable x is of type **Number**" or "variable x is not nil"—in our logical system we write these as $\mathbf{Number}_x$ and $\overline{\mathbf{nil}}_x$. The other propositions are standard logical connectives: implications, conjunctions, disjunctions, and the trivial ($\mathbb{tt}$) and impossible ($\mathbb{ff}$) propositions (Figure 3).

The particular part of the runtime environment we reference in a type proposition is called the *object*. The typing judgment relates an object to every expression in the language. An object is either *empty*, written $\emptyset$, which says this expression is not known to evaluate to a particular part of the current runtime environment, or a variable with some *path*, written $\pi(x)$, that exactly indicates how the value of this expression can be derived from the current runtime environment. Type propositions can only reference non-empty objects.

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

Figure 3. Syntax of Terms, Types, Propositions and Objects

The second insight is that we can replace the traditional representation of a type environment (eg., a map from variables to types) with a set of propositions, written Γ. Instead of mapping x to the type **Number**, we use the proposition $\mathbf{Number}_x$.

Given a set of propositions, we can use logical reasoning to derive new information about our programs with the judgment $\Gamma \vdash \psi$. In addition to the standard rules for the logical connectives, the key rule is L-Update, which combines multiple propositions about the same variable, allowing us to refine its type.

$$\frac{\Gamma \vdash \tau_{\pi'(x)} \qquad \Gamma \vdash \nu_{\pi(\pi'(x))}}{\Gamma \vdash \mathsf{update}(\tau, \nu, \pi)_{\pi'(x)}}\ \text{L-UPDATE}$$

For example, with L-Update we can use the knowledge of $\Gamma \vdash (\bigcup\ \mathbf{nil}\ \mathbf{N})_x$ and $\Gamma \vdash \overline{\mathbf{nil}}_x$ to derive $\Gamma \vdash \mathbf{N}_x$. (The metavariable ν ranges over τ and $\overline{\tau}$ (without variables).) We cover L-Update in more detail in Section 3.3.

Finally, this approach allows the type system to track programming idioms from dynamic languages using implicit type-based reasoning based on the result of conditional tests. For instance, Example 1 only utilizes `x` once the programmer is convinced it is safe to do so based whether `(number? x)` is true or false. To express this in the type system, every expression is described by two propositions: a 'then' proposition for when it reduces to a true value, and an 'else' proposition when it reduces to a false value—for `(number? x)` the then proposition is $\mathbf{Number}_x$ and the else proposition is $\overline{\mathbf{Number}}_x$.

We formalize our type system following Tobin-Hochstadt and Felleisen (2010) (with differences highlighted in blue). The typing judgment

$$\Gamma \vdash e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$$

says expression e is of type τ in the proposition environment Γ, with 'then' proposition ψ_+, 'else' proposition ψ_- and object o. We write $\Gamma \vdash e : \tau$ if we are only interested in the type.

Syntax is given in Figure 3. Expressions include variables, values, application, abstractions, conditionals and let expressions. All binding forms introduce fresh variables. Values include booleans, nil, class literals, keywords, numbers, constants and closures. Value environments map local bindings to values.

Types include the top type, *untagged* unions, functions, singleton types and class instances. We abbreviate **Boolean** as **B**, **Keyword** as **K** and **Number** and **N**. The type (**Val K**) is inhabited by the class literal **K** and `:a` is of type **K**. We abbreviate $(\bigcup)$ as

$\bot$, (**Val** nil) as **nil**, (**Val** true) as **true** and (**Val** false) as **false**. Function types contain *latent* (terminology from (Lucassen and Gifford 1988)) propositions and object, which, along with the return type, may refer to the function argument. They are latent because they are instantiated with the actual object of the argument in applications before they are used in the proposition environment.

Figure 4 contains the core typing rules. The key rule for reasoning about conditional control flow is T-If.

T-If

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \ ; \ \psi_{1+}|\psi_{1-} \ ; \ o_1 \\ \Gamma, \psi_{1+} \vdash e_2 : \tau \ ; \ \psi_{2+}|\psi_{2-} \ ; \ o \\ \Gamma, \psi_{1-} \vdash e_3 : \tau \ ; \ \psi_{3+}|\psi_{3-} \ ; \ o\end{array}}{\Gamma \vdash (\text{if } e_1\ e_2\ e_3) : \tau \ ; \ \psi_{2+} \vee \psi_{3+}|\psi_{2-} \vee \psi_{3-} \ ; \ o}$$

The propositions of the test expression e_1, ψ_{1+} and ψ_{1-}, are used as assumptions in the then and else branch respectively. If the result of the if is a true value, then it either came from e_2, in which case ψ_{2+} is true, or from e_3, which implies ψ_{3+} is true. The else proposition is $\psi_{2-} \vee \psi_{3-}$ similarly. The T-Local rule connects the type system to the proof system over type propositions via $\Gamma \vdash \tau_x$ to derive a type for a variable. Using this rule, the type system can then appeal to L-Update to refine the type assigned to x.

We are now equipped to type check Example 1, starting at body:

```
... (if (number? x) (inc x) 0) ...
```

We know Γ = $(\bigcup\ \textbf{nil}\ \ \textbf{N})_x$. The test expression uses T-App,

$$\Gamma \vdash (\textit{number? } x) : \textbf{B} \ ; \ \textbf{N}_x|\overline{\textbf{N}}_x \ ; \ \emptyset$$

since *number?* has type $x{:}\top \xrightarrow[\emptyset]{\textbf{N}_x|\overline{\textbf{N}}_x} \textbf{B}$ and x has object x.

Finally we check both branches using the extended proposition environment as specified by T-If. Going down the then branch, our new assumption $\textbf{N}_x$ is crucial to check

$$\Gamma, \textbf{N}_x \vdash x : \textbf{N} \ ; \ \overline{(\cup\ \textbf{nil}\ \ \textbf{false})}_x|(\cup\ \textbf{nil}\ \ \textbf{false})_x \ ; \ \emptyset$$

because we can now satisfy the premise of T-Local:

$$\Gamma, \textbf{N}_x \vdash \textbf{N}_x .$$

Operational semantics We define the dynamic semantics for λ_{TC} in a big-step style using an environment, following Tobin-Hochstadt and Felleisen (2010). We include both errors and a wrong value, which is provably ruled out by the type system. The main judgment is $\rho \vdash e \Downarrow \alpha$ which states that e evaluates to answer α in environment ρ. We chose to omit the core rules (see Figure A.17) however a notable difference is nil is a false value, which affects the semantics of if:

B-IfTrue

$$\frac{\begin{array}{c}\rho \vdash e_1 \Downarrow v_1 \\ v_1 \neq \mathsf{false} \qquad v_1 \neq \mathsf{nil} \\ \rho \vdash e_2 \Downarrow v\end{array}}{\rho \vdash (\text{if } e_1\ e_2\ e_3) \Downarrow v}$$

B-IfFalse

$$\frac{\begin{array}{c}\rho \vdash e_1 \Downarrow \mathsf{false} \ \text{ or } \ \rho \vdash e_1 \Downarrow \mathsf{nil} \\ \rho \vdash e_3 \Downarrow v\end{array}}{\rho \vdash (\text{if } e_1\ e_2\ e_3) \Downarrow v}$$

Subtyping (Figure 5) is a reflexive and transitive relation with top type $\top$. Singleton types are instances of their respective classes—boolean singleton types are of type **B**, class literals are instances of **Class** and keywords are instances of **K**. Instances of classes C are subtypes of **Object**. Since function types are subtypes of **Fn**, all types except for **nil** are subtypes of **Object**, so $\top$ = $(\bigcup\ \textbf{nil}\ \ \textbf{Object})$. Function subtyping is contravariant left of the arrow—latent propositions, object and the result type are covariant. Subtyping for untagged unions is standard.

S-Refl

$$\vdash \tau <: \tau$$

S-Top

$$\vdash \tau <: \top$$

S-UnionSuper

$$\frac{\exists i. \vdash \tau <: \sigma_i}{\vdash \tau <: (\bigcup\ \overrightarrow{\sigma}^{i})}$$

S-UnionSub

$$\frac{\overrightarrow{\vdash \tau_i <: \sigma}^{i}}{\vdash (\bigcup\ \overrightarrow{\tau}^{i}) <: \sigma}$$

S-FunMono

$$\vdash x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau <: \textbf{Fn}$$

S-Object

$$\vdash C <: \textbf{Object}$$

S-SClass

$$\vdash (\textbf{Val}\, C) <: \textbf{Class}$$

S-SBool

$$\vdash (\textbf{Val}\, b) <: \textbf{B}$$

S-SKw

$$\vdash (\textbf{Val}\, k) <: \textbf{K}$$

S-Fun

$$\frac{\begin{array}{cc}\vdash \sigma' <: \sigma & \vdash \tau <: \tau' \\ \psi_+ \vdash \psi'_+ & \psi_- \vdash \psi'_- \\ \multicolumn{2}{c}{\vdash o <: o'}\end{array}}{\vdash x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau <: x{:}\sigma' \xrightarrow[o']{\psi'_+|\psi'_-} \tau'}$$

Figure 5. Core Subtyping rules

3.1 Reasoning about Exceptional Control Flow

We extend our model with sequencing expressions and errors, where err models the result of calling Clojure's `throw` special form with some `Throwable`.

$$e ::= \ldots \ | \ \mathsf{err} \ | \ (\text{do } e\ e) \qquad \text{Expressions}$$

Our main insight is as follows: if the first subexpression in a sequence reduces to a value, then it is either true or false. If we learn some proposition in both cases then we can use that proposition as an assumption to check the second subexpression. T-Do formalizes this intuition.

T-Do

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \ ; \ \psi_{1+}|\psi_{1-} \ ; \ o_1 \\ \Gamma, \psi_{1+} \vee \psi_{1-} \vdash e : \tau \ ; \ \psi_+|\psi_- \ ; \ o\end{array}}{\Gamma \vdash (\text{do } e_1\ e) : \tau \ ; \ \psi_+|\psi_- \ ; \ o}$$

The introduction of errors, which do not evaluate to either a true or false value, makes our insight interesting.

T-Error

$$\Gamma \vdash \mathsf{err} : \bot \ ; \ \mathbb{ff}|\mathbb{ff} \ ; \ \emptyset$$

Recall Example 2.

```
...  (do (if (number? x) nil (throw (Exception.)))
         (inc x)) ...
```

As before, checking $(\textit{number? } x)$ allows us to use the proposition $\textbf{N}_x$ when checking the then branch.

By T-Nil and subsumption we can propagate this information to both propositions.

$$\textbf{N}_x \vdash \mathsf{nil} : \textbf{nil} \ ; \ \textbf{N}_x|\textbf{N}_x \ ; \ \emptyset$$

Furthermore, using T-Error and subsumption we can conclude anything in the else branch.

$$\overline{\textbf{N}}_x \vdash \mathsf{err} : \bot \ ; \ \textbf{N}_x|\textbf{N}_x \ ; \ \emptyset$$

Using the above as premises to T-If we conclude that if the first expression in the do evaluates successfully, $\textbf{N}_x$ must be true.

$$(\bigcup\ \textbf{nil}\ \ \textbf{N})_x \vdash (\text{if } (\textit{number? } x)\ \mathsf{nil}\ \mathsf{err}) : \textbf{B} \ ; \ \textbf{N}_x|\textbf{N}_x \ ; \ \emptyset$$

We can now use $\textbf{N}_x$ in the environment to check the second subexpression $(\textit{inc } x)$, completing the example.

3.2 Precise Types for Heterogeneous maps

Figure 6 presents syntax, typing rules and dynamic semantics in detail. The type $(\textbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})$ includes $\mathcal{M}$, a map of *present* entries (mapping keywords to types), $\mathcal{A}$, a set of keyword keys that

T-ABS
$$\frac{\Gamma, \sigma_x \vdash e : \tau \,;\, \psi_+|\psi_- \,;\, o}{\Gamma \vdash \lambda x^\sigma . e : x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau \,;\, \mathtt{tt}|\mathtt{ff} \,;\, \emptyset}$$

T-SUBSUME
$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \,;\, \psi_+|\psi_- \,;\, o \\ \Gamma, \psi_+ \vdash {\psi'}_+ \qquad \Gamma, \psi_- \vdash {\psi'}_- \\ \vdash \tau <: \tau' \qquad \vdash o <: o'\end{array}}{\Gamma \vdash e : \tau' \,;\, {\psi'}_+|{\psi'}_- \,;\, o'}$$

T-NUM
$$\Gamma \vdash n : \mathbf{N} \,;\, \mathtt{tt}|\mathtt{ff} \,;\, \emptyset$$

T-CONST
$$\Gamma \vdash c : \delta_\tau(c) \,;\, \mathtt{tt}|\mathtt{ff} \,;\, \emptyset$$

T-TRUE
$$\Gamma \vdash \mathsf{true} : \mathbf{true} \,;\, \mathtt{tt}|\mathtt{ff} \,;\, \emptyset$$

T-KW
$$\Gamma \vdash k : (\mathbf{Val}\, k) \,;\, \mathtt{tt}|\mathtt{ff} \,;\, \emptyset$$

T-FALSE
$$\Gamma \vdash \mathsf{false} : \mathbf{false} \,;\, \mathtt{ff}|\mathtt{tt} \,;\, \emptyset$$

T-CLASS
$$\Gamma \vdash C : (\mathbf{Val}\, C) \,;\, \mathtt{tt}|\mathtt{ff} \,;\, \emptyset$$

T-NIL
$$\Gamma \vdash \mathsf{nil} : \mathbf{nil} \,;\, \mathtt{ff}|\mathtt{tt} \,;\, \emptyset$$

T-LET
$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma \,;\, \psi_{1+}|\psi_{1-} \,;\, o_1 \\ \psi' = \overline{(\cup\ \mathbf{nil}\ \mathbf{false})}_x \supset \psi_{1+} \\ \psi'' = (\cup\ \mathbf{nil}\ \mathbf{false})_x \supset \psi_{1-} \\ \Gamma, \sigma_x, \psi', \psi'' \vdash e : \tau \,;\, \psi_+|\psi_- \,;\, o\end{array}}{\Gamma \vdash (\mathrm{let}\ [x\ e_1]\ e) : \tau \,;\, \psi_+|\psi_-[o_1/x] \,;\, o[o_1/x]}$$

T-LOCAL
$$\frac{\begin{array}{c}\Gamma \vdash \tau_x \\ \sigma = (\cup\ \mathbf{nil}\ \mathbf{false})\end{array}}{\Gamma \vdash x : \tau \,;\, \overline{\sigma}_x|\sigma_x \,;\, x}$$

T-APP
$$\frac{\begin{array}{c}\Gamma \vdash e : x{:}\sigma \xrightarrow[o_f]{\psi_{f+}|\psi_{f-}} \tau \,;\, \psi_+|\psi_- \,;\, o \\ \Gamma \vdash e' : \sigma \,;\, {\psi'}_+|{\psi'}_- \,;\, o'\end{array}}{\Gamma \vdash (e\ e') : \tau[o'/x] \,;\, \psi_{f+}|\psi_{f-}[o'/x] \,;\, o_f[o'/x]}$$

Figure 4. Typing rules

$$\begin{array}{lll}
e & ::= \ldots \mid (\mathrm{get}\ e\ e) \mid (\mathrm{assoc}\ e\ e\ e) & \text{Expressions} \\
v & ::= \ldots \mid \{\} & \text{Values} \\
\tau & ::= \ldots \mid (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}) & \text{Types} \\
\mathcal{M} & ::= \{\overrightarrow{k \mapsto \tau}\} & \text{HMap mandatory entries} \\
\mathcal{A} & ::= \{\overrightarrow{k}\} & \text{HMap absent entries} \\
\mathcal{E} & ::= \mathcal{C} \mid \mathcal{P} & \text{HMap completeness tags} \\
pe & ::= \ldots \mid \mathbf{key}_k & \text{Path Elements}
\end{array}$$

T-GETHMAP
$$\frac{\begin{array}{c}\Gamma \vdash e : (\bigcup\ \overrightarrow{(\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})}^i\ ) \,;\, \psi_{1+}|\psi_{1-} \,;\, o \\ \Gamma \vdash e_k : (\mathbf{Val}\, k) \qquad \overrightarrow{\mathcal{M}[k] = \tau}^i\end{array}}{\Gamma \vdash (\mathrm{get}\ e\ e_k) : (\bigcup\ \overrightarrow{\tau}^i) \,;\, \mathtt{tt}|\mathtt{tt} \,;\, \mathbf{key}_k(x)[o/x]}$$

T-GETHMAPABSENT
$$\frac{\begin{array}{c}\Gamma \vdash e : (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}) \,;\, \psi_{1+}|\psi_{1-} \,;\, o \\ \Gamma \vdash e_k : (\mathbf{Val}\, k) \qquad k \in \mathcal{A}\end{array}}{\Gamma \vdash (\mathrm{get}\ e\ e_k) : \mathbf{nil} \,;\, \mathtt{tt}|\mathtt{tt} \,;\, \mathbf{key}_k(x)[o/x]}$$

T-GETHMAPPARTIALDEFAULT
$$\frac{\begin{array}{c}\Gamma \vdash e : (\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}\ \mathcal{A}) \,;\, \psi_{1+}|\psi_{1-} \,;\, o \\ \Gamma \vdash e_k : (\mathbf{Val}\, k) \qquad k \notin dom(\mathcal{M}) \qquad k \notin \mathcal{A}\end{array}}{\Gamma \vdash (\mathrm{get}\ e\ e_k) : \top \,;\, \mathtt{tt}|\mathtt{tt} \,;\, \mathbf{key}_k(x)[o/x]}$$

T-ASSOCHMAP
$$\frac{\begin{array}{c}\Gamma \vdash e : (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}) \\ \Gamma \vdash e_k : (\mathbf{Val}\, k) \qquad \Gamma \vdash e_v : \tau \qquad k \notin \mathcal{A}\end{array}}{\Gamma \vdash (\mathrm{assoc}\ e\ e_k\ e_v) : (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}[k \mapsto \tau]\ \mathcal{A}) \,;\, \mathtt{tt}|\mathtt{ff} \,;\, \emptyset}$$

S-HMAP
$$\frac{\forall i.\ \mathcal{M}[k_i] = \sigma_i \text{ and} \vdash \sigma_i <: \tau_i \qquad \mathcal{A}_1 \supseteq \mathcal{A}_2}{\vdash (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}_1) <: (\mathbf{HMap}^{\mathcal{E}}\ \{\overrightarrow{k \mapsto \tau}\}^i\ \mathcal{A}_2)}$$

S-HMAPP
$$\frac{\forall i.\ \mathcal{M}[k_i] = \sigma_i \text{ and} \vdash \sigma_i <: \tau_i}{\vdash (\mathbf{HMap}^{\mathcal{C}}\ \mathcal{M}\ \mathcal{A}') <: (\mathbf{HMap}^{\mathcal{P}}\ \{\overrightarrow{k \mapsto \tau}\}^i\ \mathcal{A})}$$

S-HMAPMONO
$$\vdash (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}) <: \mathbf{Map}$$

B-ASSOC
$$\frac{\begin{array}{c}\rho \vdash e \Downarrow m \\ \rho \vdash e_k \Downarrow k \\ \rho \vdash e_v \Downarrow v_v \\ v = m[k \mapsto v_v]\end{array}}{\rho \vdash (\mathrm{assoc}\ e\ e_k\ e_v) \Downarrow v}$$

B-GET
$$\frac{\begin{array}{c}\rho \vdash e \Downarrow m \\ \rho \vdash e' \Downarrow k \\ k \in dom(m) \\ m[k] = v\end{array}}{\rho \vdash (\mathrm{get}\ e\ e') \Downarrow v}$$

B-GETMISSING
$$\frac{\begin{array}{c}\rho \vdash e \Downarrow m \\ \rho \vdash e' \Downarrow k \\ k \notin dom(m)\end{array}}{\rho \vdash (\mathrm{get}\ e\ e') \Downarrow \mathsf{nil}}$$

Figure 6. HMap Syntax, Typing and Operational Semantics

are known to be *absent* and tag $\mathcal{E}$ which is either $\mathcal{C}$ ("complete") if the map is fully specified by $\mathcal{M}$, and $\mathcal{P}$ ("partial") if there are *unknown* entries. To ease presentation, if an HMap has completeness tag $\mathcal{C}$ then $\mathcal{A}$ implicitly contains all keywords not in the domain of $\mathcal{M}$. Keys cannot be both present and absent.

The expressions `(get m :a)` and `(:a m)` are semantically identical, though we only model the former to avoid the added complexity of keywords being functions. To simplify presentation, we only provide syntax for the empty map literal and restrict lookup and extension to keyword keys. The metavariable m ranges over the runtime value of maps $\{\overrightarrow{k \mapsto v}\}$, usually written $\{\overrightarrow{k\ v}\}$.

Subtyping for HMaps designate **Map** as a common supertype for all HMaps. S-HMap says that an HMap is a subtype of another HMap if they agree on $\mathcal{E}$, agree on mandatory entries with subtyping and at least cover the absent keys of the supertype. Complete maps are subtypes of partial maps as long as they agree on the mandatory entries of the partial map via subtyping (S-HMapP).

The typing rules for get consider three possible cases. T-GetHMap models a lookup that will certainly succeed, T-GetHMapAbsent a lookup that will certainly fail and T-GetHMapPartialDefault a lookup with unknown results. Lookups on unions of HMaps are only supported in T-GetHMap, in particular to support looking up `:op` on a map of type `Expr` (Example 3) where every element in the union contains the key we are looking up. The objects in the T-Get rules are more complicated than those in T-Local—the next section discusses this in detail. Finally T-AssocHMap extends an HMap with a mandatory entry while preserving completeness and absent entries, and enforcing $k \notin \mathcal{A}$ to prevent badly formed types.

The semantics for get and assoc are straightforward. If the entry is missing, B-GetMissing produces nil.

3.3 Paths

Recall the first insight of occurrence typing—we can reason about specific *parts* of the runtime environment using propositions. We refer to parts of the runtime environment via a *path* that consists of a series of *path elements* applied right-to-left to a variable written $\pi(x)$. Tobin-Hochstadt and Felleisen (2010) introduce the path elements **car** and **cdr** to reason about selector operations on cons cells. We instead want to reason about HMap lookups and calls to *class*.

Key path element We introduce our first path element $\mathbf{key}_k$, which represents the operation of looking up a key k. We directly relate this to our typing rule T-GetHMap (Figure 6) by checking the then branch of the first conditional test is checked in an equivalent version of Example 3.

```
(fn [m :- Expr]
  (if (= (get m :op) :if)
    {:op :if, ...}
    (if ...)))
```

We do not specifically support $=$ in our calculus, but on keyword arguments it works identically to `isa?` which we model in Section 3.5. Intuitively, if $\Gamma \vdash e : \tau \; ; \; \psi_+|\psi_- \; ; \; o$ then $(= e\ \text{:if})$ has the true and false propositions

$$(\mathbf{Val}\ \text{:if})_x | \overline{(\mathbf{Val}\ \text{:if})_x} [o/x]$$

where substitution reduces to $\mathtt{tt}$ if $o = \emptyset$.

We start with proposition environment $\Gamma = \mathbf{Expr}_{\ m}$. Since **Expr** is a union of HMaps, each with the entry :op, we can use T-GetHMap.

$$\Gamma \vdash (\text{get } m \text{ :op}) : \mathbf{K} \; ; \; \mathtt{tt}|\mathtt{tt} \; ; \; \mathbf{key}_{\text{:op}}(m)$$

Using our intuitive definition of $=$ above, we know

$$\Gamma \vdash (= \ (\text{get } m \text{ :op}) \text{ :if}) {:} \mathbf{B} \; ; \; (\mathbf{Val}\ \text{:if})_{\mathbf{key}_{\text{:op}}(m)} | \overline{(\mathbf{Val}\ \text{:if})_{\mathbf{key}_{\text{:op}}(m)}} \; ; \; \emptyset$$

Going down the then branch gives us the extended environment Γ' = $\mathbf{Expr}_{\ m}, (\mathbf{Val}\ \text{:if})_{\mathbf{key}_{\text{:op}}(m)}$. Using L-Update we can combine what we know about object m and object $\mathbf{key}_{\text{:op}}(m)$ to derive

$$\Gamma' \vdash (\mathbf{HMap}^{\mathcal{P}}\{\text{:op :if, :test } \mathbf{Expr}\text{ , :then } \mathbf{Expr}\text{ , :else } \mathbf{Expr}\ \} \ \{\})_m$$

The full definition of update is given in Figure 10 which considers both keys a path elements as well as the *class* path element described below. In the absence of paths, update simply performs set-theoretic operations on types; see Figure 11 for details.

Class path element Our second path element **class** is used in the latent object of the constant *class* function. Like Clojure's `class` function *class* returns the argument's class or nil if passed nil.

$$pe \quad ::= \quad \ldots \mid \textbf{class} \quad \text{Path Elements}$$

$$\delta_\tau(\mathit{class}) \quad = \quad x{:}\top \xrightarrow[\textbf{class}\,(x)]{\mathtt{tt}|\mathtt{tt}} (\bigcup \ \text{nil} \ \mathbf{Class}\ )$$

The dynamic semantics are given in Figure 8. The definition of update supports various idioms relating to **class** which we introduce in Section 3.5.

3.4 Java Interoperability and Type Hints

In Section 2.6 we discussed the role of type hints to help eliminate reflective calls. In this section, we introduce our model of Java and provide user-facing syntax corresponding to Clojure's Java interoperability forms and type hinted forms. Then we model the Clojure compiler's compile-time reflection resolution algorithm. To achieve this, first we define notation for *non-reflective* Java forms that unambiguously call a field, method or constructor. Then we define a rewrite relation that uses type hints to resolve reflection explicitly. Finally we give typing rules that model how Typed Clojure interacts with non-reflective calls.

We present Java interoperability in a restricted setting without class inheritance, overloading or Java Generics.

We extend the syntax in Figure 7 with type hinted expressions, reflective and non-reflective Java field lookups and calls to methods and constructors. We model the syntax after the 'dot' special form to prevent ambiguity—`(.fld e)` is now $(.\ e\ fld)$, `(.mth e es*)` is $(.\ e\ (mth\ \overrightarrow{es}))$ and `(.class es*)` is $(\text{new}\ C\ \overrightarrow{es})$. The reflective expressions come without typing rules because Typed Clojure only reasons about resolved reflection (as demonstrated in Section 2.6). The method call $(.\ e\ (mth^{C_1}_{[[\overrightarrow{C_i}],C_2]}\ \overrightarrow{e_i}))$ is a non-reflective call to the mth method on class C_1, with Java signature $C_2\ mth\ (\overrightarrow{C_i})$;. The field access $(.\ e\ fld^{C_1}_{C_2})$ calls the field on class C_1 with Java signature $C_2\ fld$;. The constructor invocation $(\text{new}_{[\overrightarrow{C_i}]}\ C\ \overrightarrow{e_i})$ calls the constructor with Java signature $C\ (\overrightarrow{C_i})$;.

$$\begin{array}{lll}
e & ::= \ldots\ \hat{}C\ x \mid (\text{let}\ [\hat{}C\ x\ e]\ e) & \text{Expressions} \\
 & \mid\ (.\ e\ fld) \mid (.\ e\ (mth\ \overrightarrow{e})) \mid (\text{new}\ C\ \overrightarrow{e}) & \\
 & \mid\ (.\ e\ (mth^{C}_{[[\overrightarrow{C}],C]}\ \overrightarrow{e})) & \text{Non-reflective Expressions} \\
 & \mid\ (.\ e\ fld^{C}_{C}) \mid (\text{new}_{[\overrightarrow{C}]}\ C\ \overrightarrow{e}) & \\
v & ::= \ldots \mid C\ \{\overrightarrow{fld : v}\} & \text{Values} \\
\gamma & ::= ?\ \mid C & \text{Type Hints} \\
\Sigma & ::= \{\overrightarrow{x : \gamma}\} & \text{Type Hint Environment} \\
ce & ::= \{\mathsf{mths} \mapsto \{\overrightarrow{[mth, [\overrightarrow{C}], C]}\}, & \text{Class descriptors} \\
 & \quad \mathsf{flds} \mapsto \{\overrightarrow{[fld, C]}\}, & \\
 & \quad \mathsf{ctors} \mapsto \{\overrightarrow{[\overrightarrow{C}]}\}\} & \\
\mathcal{CT} & ::= \{\overrightarrow{C \mapsto ce}\} & \text{Class Table}
\end{array}$$

T-NewStatic

$$\frac{\overrightarrow{\mathsf{Convert}(C_i) \ = \ \tau_i} \qquad \mathsf{Convert}(C) \ = \ \tau \qquad \overrightarrow{\Gamma \vdash e_i : \tau_i}}{\Gamma \vdash (\text{new}_{[\overrightarrow{C_i}]}\ C\ \overrightarrow{e_i}) : \tau \; ; \; \mathtt{tt}|\mathtt{ff} \; ; \; \emptyset}$$

T-MethodStatic

$$\frac{\begin{array}{c}\overrightarrow{\mathsf{Convert}(C_i) \ = \ \tau_i} \qquad \mathsf{Convert}(C_1) \ = \ \sigma \\ \mathsf{Convert}_{\mathsf{nil}}(C_2) \ = \ \tau \qquad \Gamma \vdash e : \sigma \qquad \overrightarrow{\Gamma \vdash e_i : \tau_i}\end{array}}{\Gamma \vdash (.\ e\ (mth^{C_1}_{[[\overrightarrow{C_i}],C_2]}\ \overrightarrow{e_i})) : \tau \; ; \; \mathtt{tt}|\mathtt{tt} \; ; \; \emptyset}$$

$$\begin{array}{llll}
\mathsf{Fld}(\mathcal{CT}, C, fld) & = & [C, C_f] & \text{if } [fld, C_f] \in \mathcal{CT}[C][\mathsf{flds}] \\
\mathsf{Ctor}(\mathcal{CT}, C, [\overrightarrow{C_p}]) & = & [\overrightarrow{C_p}] & \text{if } [\overrightarrow{C_p}] \in \mathcal{CT}[C][\mathsf{ctors}] \\
\mathsf{Mth}(\mathcal{CT}, C, mth, [\overrightarrow{C_p}]) & = & [C, [\overrightarrow{C_p}], C_r] & \text{if } [mth, [\overrightarrow{C_p}], C_r] \in \mathcal{CT}[C][\mathsf{mths}]
\end{array}$$

$$\begin{array}{lll}
\mathsf{Convert}_{\mathsf{nil}}(\mathbf{Void}\ ) & = & \mathbf{nil} \\
\mathsf{Convert}_{\mathsf{nil}}(C) & = & (\bigcup\ \mathbf{nil}\ \ C)
\end{array}
\qquad
\begin{array}{lll}
\mathsf{Convert}(\mathbf{Void}\ ) & = & \mathbf{nil} \\
\mathsf{Convert}(C) & = & C
\end{array}$$

B-Field

$$\frac{\begin{array}{c}\rho \vdash e \Downarrow v_1 \\ \mathsf{JVM}_{\mathsf{getstatic}}[C_1, v_1, fld, C_2] = v\end{array}}{\rho \vdash (.\ e\ fld^{C_1}_{C_2}) \Downarrow v}$$

B-New

$$\frac{\begin{array}{c}\overrightarrow{\rho \vdash e_i \Downarrow v_i} \\ \mathsf{JVM}_{\mathsf{new}}[C_1, [\overrightarrow{C_i}], [\overrightarrow{v_i}]] = v\end{array}}{\rho \vdash (\text{new}_{[\overrightarrow{C_i}]}\ C\ \overrightarrow{e_i}) \Downarrow v}$$

B-Method

$$\frac{\begin{array}{c}\rho \vdash e_m \Downarrow v_m \qquad \overrightarrow{\rho \vdash e_a \Downarrow v_a} \\ \mathsf{JVM}_{\mathsf{invokestatic}}[C_1, v_m, mth, [\overrightarrow{C_a}], [\overrightarrow{v_a}], C_2] = v\end{array}}{\rho \vdash (.\ e_m\ (mth^{C_1}_{[[\overrightarrow{C_a}],C_2]}\ \overrightarrow{e_a})) \Downarrow v}$$

Figure 7. Java Interoperability Syntax, Typing and Operational Semantics

We model Clojure's reflection resolution algorithm as a rewrite relation $\Sigma \ \vdash^{\mathcal{CT}}_{\mathsf{r}} \ e \ \Rightarrow \ e'$ which rewrites e to a possibly-less reflective expression e' with respect to type hint environment Σ and Java class table $\mathcal{CT}$. For example, R-FieldElimRefl emits a non-reflective field if it can find a field matching the type hint inferred on e'.

R-FieldElimRefl

$$\frac{\begin{array}{c}\Sigma \ \vdash^{\mathcal{CT}}_{\mathsf{r}} \ e \Rightarrow e' \qquad \Sigma \vdash_{\mathsf{h}} e' : C \\ \mathsf{Fld}(\mathcal{CT}, C, fld) = [C_t, C_f]\end{array}}{\Sigma \ \vdash^{\mathcal{CT}}_{\mathsf{r}} \ (.\ e\ fld) \Rightarrow (.\ e'\ fld^{C_t}_{C_f})}$$

Type hint inference is given by the judgment $\Sigma \vdash_{\mathsf{h}} e : \gamma$ which infers the (possibly-unknown) type hint γ of expression e in type hint environment Σ. We defer the remainder of the rules for rewriting and the definition of type hint inference to the supplemental material.

As an example, we rewrite a simple field reference, with the assumptions that that the class **Point** has a field x of Java type **N** and $\Sigma(p) = \mathbf{Point}$. In rewriting the expression $(.\ p\ x)$, **Point** is inferred

$$\begin{array}{lcl@{\qquad}lcl}
\delta(\mathit{class}, C\ \{\overrightarrow{\mathit{fld} : v}\}) & = & C & & & \\
\delta(\mathit{class}, C) & = & \mathbf{Class} & \delta(\mathit{class}, \mathsf{true}) & = & \mathbf{B} \\
\delta(\mathit{class}, [\rho, \lambda x^{\tau}.e]_{\mathsf{c}}) & = & \mathbf{Fn} & \delta(\mathit{class}, \mathsf{false}) & = & \mathbf{B} \\
\delta(\mathit{class}, [v_d, t]_{\mathsf{m}}) & = & \mathbf{Multi} & \delta(\mathit{class}, k) & = & \mathbf{K} \\
\delta(\mathit{class}, m) & = & \mathbf{Map} & \delta(\mathit{class}, \mathsf{nil}) & = & \mathsf{nil}
\end{array}$$

Figure 8. Primitives

as the type hint of p and **N** is the field type by Fld (Figure 7). Now we just plug in the new information into our new expression

$$(.\ p\ x^{\mathbf{Point}}_{\mathbf{N}}).$$

Now we present the typing rules for resolved Java interoperability. T-FieldStatic checks a resolved field expression by ensuring the target has the correct static type, then returns a nilable type corresponding the Java type.

T-FieldStatic
$$\frac{\mathsf{Convert}(C_1)\ =\ \sigma \qquad \mathsf{Convert}_{\mathsf{nil}}(C_2)\ =\ \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash (.\ e\ \mathit{fld}^{C_1}_{C_2}) : \tau\ ;\ \mathtt{tt}|\mathtt{tt}\ ;\ \emptyset}$$

To continue our example, assume Γ = $\mathbf{Point}_p$. T-FieldStatic therefore produces the type ($\bigcup$ **nil** **N**) for the entire expression.

The rules T-MethodStatic and T-NewStatic work similarly (Figure 7), varying in the choice of nilability in the conversion function—methods can return nil but constructors cannot.

The evaluation rules B-Field, B-New and B-Method (Figure 7) simply evaluate their arguments and call the relevant JVM operation, which do not model—Section 4 states our exact assumptions.

3.5 Multimethod preliminaries: isa?

We now consider the isa? operation, a core part of the dispatch mechanism for multimethods. Recalling the examples in Section 2.7, isa? is a subclassing test for classes, otherwise an equality test—we do not model the semantics for vectors.

The key component of the T-IsA rule is the $\mathsf{IsAProps}$ metafunction (Figure 9), used to calculate the propositions for isa? tests.

T-IsA
$$\frac{\begin{array}{c}\Gamma \vdash e : \sigma\ ;\ {\psi'}_{+}|{\psi'}_{-}\ ;\ o \\ \Gamma \vdash e' : \tau \qquad \mathsf{IsAProps}(o, \tau) = \psi_{+}|\psi_{-}\end{array}}{\Gamma \vdash (\text{isa?}\ e\ e') : \mathbf{B}\ ;\ \psi_{+}|\psi_{-}\ ;\ \emptyset}$$

As an example, (isa? $(\mathit{class}\ x)$ **K**) has the true and false propositions $\mathsf{IsAProps}(\mathbf{class}(x), (\mathbf{Val}\,\mathbf{K})) = \mathbf{K}_x|\overline{\mathbf{K}}_x$, meaning that if this expression produces true, x is a keyword, otherwise it is not.

The operational behavior of isa? is given by B-IsA (Figure 9). IsA explicitly handles classes in the second case.

3.6 Multimethods

To ease presentation, we present *immutable* multimethods, with syntax and semantics given in Figure 9. defmethod returns a new extended multimethod without changing the original multimethod. Example 11 is now written

```
(let [path (defmulti [Any -> (U nil String)] class)]
 (let [path (defmethod path String [x] x)]
  (let [path (defmethod path File [^File x]
        (.getPath x))]
   (path "dir/a")))) ;=> "a"
```

The type (**Multi** σ σ') characterizes multimethods with *interface type* σ and *dispatch function type* σ'. The expression (defmulti σ e) defines a multimethod with interface type σ and dispatch function e. The expression (defmethod e_m e_v e_f) extends multimethod e_m and to map dispatch value e_v to e_f in an extended dispatch table. The value $[v, t]_{\mathsf{m}}$ is the runtime value of a multimethod with dispatch function v and dispatch table t.

$$\begin{array}{lll@{\qquad}l}
e & ::= \ldots \ |\ (\text{defmulti}\ \tau\ e) & & \text{Expressions} \\
 & \quad |\ \ (\text{defmethod}\ e\ e\ e)\ |\ (\text{isa?}\ e\ e) & & \\
v & ::= \ldots \ |\ [v, t]_{\mathsf{m}} & & \text{Values} \\
\sigma, \tau & ::= \ldots \ |\ (\mathbf{Multi}\ \tau\ \tau) & & \text{Types} \\
t & ::= \{\overrightarrow{v \mapsto v}\} & & \text{Multimethod dispatch table}
\end{array}$$

T-DefMulti
$$\frac{\sigma = x{:}\tau \xrightarrow[o]{\psi_{+}|\psi_{-}} \tau' \qquad \sigma' = x{:}\tau \xrightarrow[o']{{\psi'}_{+}|{\psi'}_{-}} \tau'' \qquad \Gamma \vdash e : \sigma'}{\Gamma \vdash (\text{defmulti}\ \sigma\ e) : (\mathbf{Multi}\ \sigma\ \sigma')\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset}$$

T-DefMethod
$$\frac{\begin{array}{c}\tau_m = x{:}\tau \xrightarrow[o]{\psi_{+}|\psi_{-}} \sigma \qquad \tau_d = x{:}\tau \xrightarrow[o']{{\psi'}_{+}|{\psi'}_{-}} \sigma' \\ \Gamma \vdash e_m : (\mathbf{Multi}\ \tau_m\ \tau_d) \qquad \mathsf{IsAProps}(o', \tau_v) = {\psi''}_{+}|{\psi''}_{-} \\ \Gamma \vdash e_v : \tau_v \qquad \Gamma, \tau_x, {\psi''}_{+} \vdash e_b : \sigma\ ;\ \psi_{+}|\psi_{-}\ ;\ o\end{array}}{\Gamma \vdash (\text{defmethod}\ e_m\ e_v\ \lambda x^{\tau}.e_b) : (\mathbf{Multi}\ \tau_m\ \tau_d)\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset}$$

$$\begin{array}{lcl}
\mathsf{IsAProps}(\mathbf{class}(\pi(x)), (\mathbf{Val}\,C)) & = & C_{\pi(x)}|\overline{C}_{\pi(x)} \\
\mathsf{IsAProps}(o, (\mathbf{Val}\,s)) & = & ((\mathbf{Val}\,s)_x|\overline{(\mathbf{Val}\,s)_x})[o/x] \\
 & & \text{if } s \neq C \\
\mathsf{IsAProps}(o, \tau) & = & \mathtt{tt}|\mathtt{tt} \qquad \text{otherwise}
\end{array}$$

S-PMultiFn
$$\frac{\begin{array}{c}\vdash \sigma_t <: x{:}\sigma \xrightarrow[o]{\psi_{+}|\psi_{-}} \tau \\ \vdash \sigma_d <: x{:}\sigma \xrightarrow[o']{{\psi'}_{+}|{\psi'}_{-}} \tau'\end{array}}{\vdash (\mathbf{Multi}\ \sigma_t\ \sigma_d) <: x{:}\sigma \xrightarrow[o]{\psi_{+}|\psi_{-}} \tau}$$

S-PMulti
$$\frac{\vdash \sigma <: \sigma' \qquad \vdash \tau <: \tau'}{\vdash (\mathbf{Multi}\ \sigma\ \tau) <: (\mathbf{Multi}\ \sigma'\ \tau')}$$

S-MultiMono
$$\vdash (\mathbf{Multi}\ x{:}\sigma \xrightarrow[o]{\psi_{+}|\psi_{-}} \tau\ x{:}\sigma \xrightarrow[o']{{\psi'}_{+}|{\psi'}_{-}} \tau') <: \mathbf{Multi}$$

B-DefMethod
$$\frac{\begin{array}{c}\rho \vdash e \Downarrow [v_d, t]_{\mathsf{m}} \\ \rho \vdash e' \Downarrow v_v \\ \rho \vdash e_f \Downarrow v_f \\ v = [v_d, t[v_v \mapsto v_f]]_{\mathsf{m}}\end{array}}{\rho \vdash (\text{defmethod}\ e\ e'\ e_f) \Downarrow v}$$

B-DefMulti
$$\frac{\begin{array}{c}\rho \vdash e \Downarrow v_d \\ v = [v_d, \{\}]_{\mathsf{m}}\end{array}}{\rho \vdash (\text{defmulti}\ \tau\ e) \Downarrow v}$$

B-BetaMulti
$$\frac{\begin{array}{c}\rho \vdash e \Downarrow [v_d, t]_{\mathsf{m}} \\ \rho \vdash e' \Downarrow v' \\ \rho \vdash (v_d\ v') \Downarrow v_e \\ \mathsf{GM}(t, v_e) = v_f \\ \rho \vdash (v_f\ v') \Downarrow v\end{array}}{\rho \vdash (e\ e') \Downarrow v}$$

$$\begin{array}{l}
\mathsf{GM}(t, v_e) = v_f \quad \text{if } \overrightarrow{v_{fs}} = \{v_f\} \\
\qquad \text{where } \overrightarrow{v_{fs}} = \{v_f | (v_v, v_f) \in t \text{ and } \mathsf{IsA}(v_v, v_e) = \mathsf{true}\} \\
\mathsf{GM}(t, v_e) = \mathsf{err} \quad \text{otherwise}
\end{array}$$

B-IsA
$$\frac{\begin{array}{c}\rho \vdash e_1 \Downarrow v_1 \\ \rho \vdash e_2 \Downarrow v_2 \\ \mathsf{IsA}(v_1, v_2) = v\end{array}}{\rho \vdash (\text{isa?}\ e_1\ e_2) \Downarrow v}$$

$$\begin{array}{lcll}
\mathsf{IsA}(v, v) & = & \mathsf{true} & v \neq C \\
\mathsf{IsA}(C, C') & = & \mathsf{true} & \vdash C <: C' \\
\mathsf{IsA}(v, v') & = & \mathsf{false} & \text{otherwise}
\end{array}$$

Figure 9. Multimethod Syntax, Typing and Operational Semantics

The T-DefMulti rule ensures that the type of the dispatch function has at least as permissive a parameter type as the interface type. For example, we can check the definition from our translation above of Example 11 using T-DefMulti.

$$\vdash (\text{defmulti}\ \sigma\ \mathit{class}) : (\mathbf{Multi}\ \sigma\ \sigma')\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset$$

$$
\begin{array}{lcll}
\mathsf{restrict}(\tau, \sigma) & = & \bot & \\
 & & \text{if } \nexists v. \vdash v : \tau \ ; \ \psi_1 \ ; \ o_1 & \\
 & & \text{and} \vdash v : \sigma \ ; \ \psi_2 \ ; \ o_2 & \\
\mathsf{restrict}((\bigcup \ \overrightarrow{\tau}), \sigma) & = & (\bigcup \ \overrightarrow{\mathsf{restrict}(\tau, \sigma)}) & \\
\mathsf{restrict}(\tau, \sigma) & = & \tau & \text{if} \vdash \tau <: \sigma \\
\mathsf{restrict}(\tau, \sigma) & = & \sigma & \text{otherwise} \\
 & & & \\
\mathsf{remove}(\tau, \sigma) & = & \bot & \text{if} \vdash \tau <: \sigma \\
\mathsf{remove}((\bigcup \ \overrightarrow{\tau}), \sigma) & = & (\bigcup \ \overrightarrow{\mathsf{remove}(\tau, \sigma)}) & \\
\mathsf{remove}(\tau, \sigma) & = & \tau & \text{otherwise}
\end{array}
$$

Figure 11. Restrict and Remove

where $\sigma = x{:}\top \xrightarrow[\emptyset]{\mathtt{tt}|\mathtt{tt}} \tau$ and $\sigma' = x{:}\top \xrightarrow[\textbf{class}\,(x)]{\mathtt{tt}|\mathtt{tt}} (\bigcup\ \textbf{nil}\ \ \textbf{Class})$. Since the parameter types agree, this is well-typed.

The T-DefMethod rule is carefully constructed to ensure we have a syntactic lambda expression as the right-most subexpression. This way we can manually check the body of the lambda under an extended environment as sketched in Example 12. We use IsAProps to compute the proposition for this method, since isa? is used at runtime in multimethod dispatch.

We continue with the next line of the translation of Example 11. From the previous line we have $\Gamma = (\textbf{Multi}\, \sigma\ \sigma')_{path}$, so

$$\Gamma \vdash (\text{defmethod}\, \textit{prop}\ \textbf{String}\ \lambda x^{\top}.x) : (\textbf{Multi}\, \sigma\ \sigma')\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset$$

We know *prop* is a multimethod by Γ, so now we check the body of this method.

$$\Gamma, \top_x, \textbf{String}_x \vdash x : \textbf{String}\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset$$

The new proposition $\textbf{String}_x$ is derived by

$$\mathsf{IsAProps}(\textbf{class}\,(x), (\textbf{Val File}\,)) = \textbf{String}_x|\overline{\textbf{String}}_x.$$

The body of the `let` is checked by T-App because $(\textbf{Multi}\ \sigma\ \sigma')$ is a subtype of its interface type σ.

Multimethod definition semantics are straightforward. B-DefMulti creates a multimethod with the given dispatch function and an empty dispatch table. B-DefMethod produces a new multimethod with an extended dispatch table. B-BetaMulti invokes the dispatch function with the evaluated argument to obtain the dispatch value, and uses GM (which models Clojure's `get-method`) to extract the appropriate method. The call to GM only returns a value if there is *exactly one* method such that the corresponding dispatch value is compatible, using IsA, with the result of the dispatch function. Finally we return the result of applying the extracted method and the original argument.

4. Metatheory

We prove type soundness follow using the same technique as Tobin-Hochstadt and Felleisen (2010). We also include errors and a $wrong$ value and prove well-typed programs do not go wrong.

Rather than modeling Java's dynamic semantics, we instead make our assumptions about Java explicit. We concede that method and constructor calls may diverge or error, but we assume they can never go wrong. (Assumptions for other operations are given in the supplemental material).

Assumption 1 ($\mathsf{JVM_{new}}$)**.** *If* $\forall i.\ v_i = C_i\ \{\overrightarrow{fld_j : v_j}\}$ *or* $v_i = \mathsf{nil}$ *and* v_i *is consistent with* ρ *then either*

- $\mathsf{JVM_{new}}[C, [\overrightarrow{C_i}], [\overrightarrow{v_i}]] = C\ \{\overrightarrow{fld_k : v_k}\}$ *which is consistent with* ρ*,*
- $\mathsf{JVM_{new}}[C, [\overrightarrow{C_i}], [\overrightarrow{v_i}]] = \mathsf{err}$*, or*
- $\mathsf{JVM_{new}}[C, [\overrightarrow{C_i}], [\overrightarrow{v_i}]]$ *is undefined.*

For the purposes of our soundness proof, we require that all values are *consistent*. Consistency ensures that occurrence typing does not refer to variables hidden inside a closure.

Definition 1. v *is consistent with* ρ *iff* $\forall\ [\rho_1, \lambda x^{\sigma}.e]_{\mathsf{c}}$ *in* v*, if* $\vdash [\rho_1, \lambda x^{\sigma}.e]_{\mathsf{c}} : \tau\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset$ *and* $\forall\ o'$ *in* τ*, either* $o' = \emptyset$*, or* $o' = \pi'(x)$*, or* $\rho(o') = \rho_1(o')$*.*

Our main lemma says if there is a defined reduction, then the propositions, object and type are correct. The metavariable α ranges over v, err and $wrong$.

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

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

Proof. By induction on the derivation of the typing judgment. (Full proof given as lemma A.7). □

We can now state our soundness theorem.

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

Theorem 2 (Well-typed programs don't go wrong)**.** *If* $\vdash e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$ *then* $\not\vdash e \Downarrow wrong$*.*

5. Experience

Typed Clojure is implemented as a Clojure library named `core.typed`. In contrast to Racket, Clojure does not provide extension points to the macroexpander. To satisfy our goals of providing Typed Clojure as a library that works with the latest version of the Clojure compiler, `core.typed` is implemented as an external static analysis pass that must be explicitly invoked by the programmer. Therefore, `core.typed` is in a sense a linter.

This means that type checking is truly optional. On the positive side, `core.typed` is flexible to the needs of a dynamically typed programmer, encouraging experimentation with programs that may not type check. On the negative side, programmers must remember to type check their namespaces, though since type checking is a function call away, it is easily integrated as editor shortcuts or continuous integration. Also, programs cannot depend on the static semantics of Typed Clojure, meaning that type-based optimisation is impossible. If this were not the case, we could dispose of type-hints altogether, and simply use static types to resolve reflection.

5.1 Further Extensions

Datatypes, Records and Protocols Clojure features datatypes and protocols. Datatypes are Java classes declared final with public final fields. They can implement Java interfaces or protocols, which are similar to interfaces but already-defined classes and nil may extend protocols. Typed Clojure can reason about most of these features, including the ability to define polymorphic datatypes and protocols and utilising the Java type system to help check implemented interface methods.

Mutation and Polymorphism Clojure supports mutable references with software-transactional-memory which Typed Clojure defines *bivariantly*—with write and read type parameters as in the atomic reference (`Atom2 Int Int`) which can write and read

$$
\begin{array}{lcll}
\mathsf{update}((\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}), \nu, \pi :: \mathbf{key}_k) & = & (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}[k \mapsto \mathsf{update}(\tau, \nu, \pi)]\ \mathcal{A}) & \text{if } \mathcal{M}[k] = \tau \\
\mathsf{update}((\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}), \tau, \pi :: \mathbf{key}_k) & = & \bot & \text{if } \vdash \mathbf{nil} \not<: \tau \text{ and } k \in \mathcal{A} \\
\mathsf{update}((\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}), \overline{\tau}, \pi :: \mathbf{key}_k) & = & \bot & \text{if } \vdash \mathbf{nil} <: \tau \text{ and } k \in \mathcal{A} \\
\mathsf{update}((\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}), \nu, \pi :: \mathbf{key}_k) & = & (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}) & \text{if } k \in \mathcal{A} \\
\mathsf{update}((\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}\ \mathcal{A}), \tau, \pi :: \mathbf{key}_k) & = & (\cup\ (\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}[k \mapsto \tau]\ \mathcal{A}) & \text{if } \vdash \mathbf{nil} <: \tau, \\
 & & \quad (\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}\ (\mathcal{A} \cup \{k\}))) & k \notin dom(\mathcal{M}) \text{ and } k \notin \mathcal{A} \\
\mathsf{update}((\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}\ \mathcal{A}), \nu, \pi :: \mathbf{key}_k) & = & (\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}[k \mapsto \mathsf{update}(\top, \nu, \pi)]\ \mathcal{A}) & \text{if } k \notin dom(\mathcal{M}) \text{ and } k \notin \mathcal{A} \\
\mathsf{update}((\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}\ \mathcal{A}), \nu, \pi :: \mathbf{key}_k) & = & (\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}[k \mapsto \mathsf{update}(\top, \nu, \pi)]\ \mathcal{A}) & \\
\mathsf{update}((\bigcup\ \overrightarrow{(\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})}^{i}\ ), \nu, \pi :: \mathbf{key}_k) & = & (\bigcup\ \overrightarrow{\mathsf{update}((\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}), \nu, \pi :: \mathbf{key}_k)}^{i}\ ) & \\
\mathsf{update}(\tau, (\mathbf{Val}\, C), \pi :: \mathbf{class}) & = & \mathsf{update}(\tau, C, \pi) & \\
\mathsf{update}(\tau, \overline{(\mathbf{Val}\, C)}, \pi :: \mathbf{class}) & = & \mathsf{update}(\tau, \overline{C}, \pi) & \text{if } \ \nexists C' . \vdash C' <: C \text{ and } C' \neq C \\
\mathsf{update}(\tau, \sigma, \pi :: \mathbf{class}) & = & \mathsf{update}(\tau, \mathbf{Object}, \pi) & \text{if } \vdash \sigma <: \mathbf{Object} \\
\mathsf{update}(\tau, \overline{\sigma}, \pi :: \mathbf{class}) & = & \mathsf{update}(\tau, \mathbf{nil}\ , \pi) & \text{if } \vdash \mathbf{Object} <: \sigma \\
\mathsf{update}(\tau, \sigma, \pi :: \mathbf{class}) & = & \mathsf{update}(\tau, \mathbf{nil}, \pi) & \text{if } \vdash \sigma <: \mathbf{nil} \\
\mathsf{update}(\tau, \overline{\sigma}, \pi :: \mathbf{class}) & = & \mathsf{update}(\tau, \mathbf{Object}, \pi) & \text{if } \vdash \mathbf{nil} <: \sigma \\
\mathsf{update}(\tau, \nu, \pi :: \mathbf{class}) & = & \tau & \\
\mathsf{update}(\tau, \sigma, \epsilon) & = & \mathsf{restrict}(\tau, \sigma) & \\
\mathsf{update}(\tau, \overline{\sigma}, \epsilon) & = & \mathsf{remove}(\tau, \sigma) &
\end{array}
$$

Figure 10. Type Update

```
(ann clojure.core/swap!
  (All [w r b ...]
    [(Atom2 w r) [r b ... b -> w] b ... b -> w]))

(swap! (atom :- Num 1) + 2 3);=> 6 (atom contains 6)
```

Figure 12. Type annotation and example call of `swap!`

`Int`. Typed Clojure also supports parametric polymorphism, including Typed Racket's variable-arity polymorphism (Strickland et al. 2009), which enables us to assign a type to functions like `swap!` (figure 12), which takes a mutable *atom*, a function and extra arguments, and swaps into the atom the result of applying the function to the atom's current value and the extra arguments.

5.2 Limitations

Java Arrays Java arrays are known to be statically unsound. Bracha et al. (1998) summarises the approach taken to regain runtime soundness, which involves checking array writes at runtime.

Typed Clojure implements an experimental partial solution, making arrays *bivariant*, separating the write and read types into contravariant and covariant parameters. If the array originates from typed code, then we may track the write and read parameters statically. Currently arrays from foreign sources have their write parameter set to to $\bot$, protecting typed code from writing something of incorrect type. However there are currently no casting mechanisms to convince Typed Clojure the foreign array is writeable.

Array-backed sequences Typed Clojure assumes sequences are immutable. This is almost always true, however for performance reasons, sequences created from Java arrays (and Iterables) reflect future writes to the array in the 'immutable' sequence. While disturbing and a clear unsoundness in Typed Clojure, this has not yet been an issue in practice and is strongly discouraged as undefined behavior: "Robust programs should not mutate arrays or Iterables that have seqs on them." (Hickey 2015).

Gradual typing Gradual typing ensures sound interoperability between typed and untyped code by enforcing invariants of the type system via run-time contracts. Currently, interactions between typed and untyped Clojure code are unchecked which can violate the expectations of Typed Clojure. We hope to add support for gradually typing in the future.

5.3 Case Study

CircleCI provides continuous integration services built with a mixture of open- and closed-source. Typed Clojure has been used at CircleCI in production Clojure systems for at least two years.

CircleCI provided the first author access to the main closed-source backend system written in Clojure and Typed Clojure. We conducted a study of the effectiveness of Typed Clojure in practice. There is no clear metric for quantifying typed Clojure code, since untyped code can be freely mixed and some seemingly typed namespaces are not checked regularly. We manually type checked all namespaces that depend on `clojure.core.typed` and considered those with type errors as untyped. We then searched the remaining typed code for unsafe Typed Clojure operations like var annotations with `:no-check` and the `tc-ignore` macro, which instruct Typed Clojure to ignore the specified code, and also considered those untyped. Furthermore, we manually collected and inspected all top-level annotations and classified them.

We determined that CircleCI has a Clojure code base of approximately 50,000 lines, including around 10,000 lines of typed code. Out of 588 top-level var annotations, 270 (46%) were checked annotations of functions defined in typed code, 129 (22%) annotations assigned types to external libraries and the remaining 189 (32%) annotated 'unchecked' user code. HMaps were a valuable feature, with 38 (59%) out of 64 total type aliases featuring them; see example 4 for an instance.

Based on this and other interactions with Typed Clojure users, it is clear the main barrier to entry to Typed Clojure for large systems is the requirement to annotate functions outside the borders of typed code. We conjecture that this can be addressed by making annotations available for popular libraries.

6. Related Work

Multimethods Millstein and Chambers and collaborators present a sequence of systems (Chambers 1992; Chambers and Leavens 1994; Millstein and Chambers 2002) with statically-typed multimethods and modular type checking. In contrast to Typed Clojure, in these system methods declare the types of arguments that they expect which corresponds to exclusively using `class` as the dispatch function in Typed Clojure. However, Typed Clojure does not attempt to rule out failed dispatches at runtime.

Record Types Row polymorphism (Wand 1989; Cardelli and Mitchell 1991; Harper and Pierce 1991), used in systems such as the OCaml object system, provides many of the features of HMap types, but defined using universally-quantified row variables. HMaps in Typed Clojure are instead designed to be used with subtyping, but nonetheless provide similar expressiveness, including the ability to require presence and absence of certain keys.

Dependent JavaScript (Chugh et al. 2012) can track similar invariants as HMaps with types for JS objects. They must deal with mutable objects, they feature refinement types and strong updates to the heap to track changes to objects.

Typed Lua (Maidl et al. 2014) has *table types* which track entries in a mutable Lua table. Typed Lua changes the dynamic semantics of Lua to accommodate mutability: Typed Lua raises a runtime error for lookups on missing keys—HMaps consider lookups on missing keys normal.

The integration of completeness information, crucial for many examples in Typed Clojure, is not provided by any of these systems.

Java Interoperability in Statically Typed Languages Scala (Odersky et al. 2006) has nullable references for compatibility with Java. Programmers must manually check for `null` as in Java to avoid null-pointer exceptions.

Other optional and gradual type systems In addition to Typed Racket, several other gradual type systems have been developed recently, targeting existing dynamically-typed languages. Reticulated Python (Vitousek et al. 2014) is an experimental gradually typed system for Python, implemented as a source-to-source translation that inserts dynamic checks at language boundaries and supporting Python's first-class object system. Typed Clojure does not support a first-class object system because Java (and Clojure) have nominal classes, however HMaps offer an alternative to the structural objects offered by Reticulated. Similarly, GradualTalk (Allende et al. 2014) offers gradual typing for SmallTalk, with nominal classes.

Optional types, requiring less implementation effort and avoiding any runtime cost, have been widely adopted in industry, including Hack, an extension to PHP (Facebook 2014), and Flow (Facebook 2015) and TypeScript (Microsoft 2014), two extensions of JavaScript. These systems all support some form of occurrence typing, but not in the generality presented here, nor do they include the other features we have presented.

7. Conclusion

We have presented Typed Clojure, an optionally-typed version of Clojure whose type system works with a wide variety of distinctive Clojure idioms and features. Although based on the foundation of Typed Racket's occurrence typing approach, Typed Clojure both extends the fundamental control-flow based reasoning as well as applying it to handle seemingly unrelated features such as multimethods. In addition, Typed Clojure supports crucial features such as heterogeneous maps and Java interoperability while integrating these features into the core type system.

The result is a sound, expressive, and useful type system which, when implemented in `core.typed` with appropriate extensions, suitable for typechecking significant amount of existing Clojure programs. As a result, Typed Clojure is already successful: it is widely used in the Clojure community among both enthusiasts and professional programmers and receives contributions from many developers.

However, there is much more that Typed Clojure can provide. Most significantly, Typed Clojure currently does not provide *gradual typing*—interaction between typed and untyped code is unchecked and thus unsound. We hope to explore the possibilities of using existing mechanisms for contracts and proxies in Java and Clojure to enable sound gradual typing for Clojure.

Additionally, the Clojure compiler is unable to use Typed Clojure's wealth of static information to optimize programs. Addressing this requires not only first enabling sound gradual typing, but also integrating Typed Clojure into the Clojure tool chain more deeply, so that its information can be passed on to the compiler.

Finally, our case study and broader experience indicate that Clojure programmers still find themselves unable to use Typed Clojure on some of their programs for lack of expressiveness. This requires continued effort to analyze and understand the relevant features and idioms and develop new type checking approaches.

Acknowledgements

Thanks to Andrew Kent and Andre Kuhlenschmidt for comment on drafts of this paper.

References

E. Allende, O. Callau, J. Fabry, É. Tanter, and M. Denker. Gradual typing for smalltalk. *Science of Computer Programming*, 96:52–69, 2014.

G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *OOPSLA*, 1998.

L. Cardelli and J. C. Mitchell. Operations on records. In *Mathematical Structures in Computer Science*, pages 3–48, 1991.

R. Cartwright and M. Fagan. Soft typing. In *Proc. PLDI*, 1991.

C. Chambers. Object-oriented multi-methods in cecil. In *Proc. ECOOP*, 1992.

C. Chambers and G. T. Leavens. Typechecking and modules for multi-methods. In *Proc. OOPSLA*, 1994.

R. Chugh, D. Herman, and R. Jhala. Dependent types for javascript. In *Proc. OOPSLA*, 2012.

Facebook. Hack language specification. Technical report, 2014.

Facebook. Flow language specification. Technical report, 2015.

R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proc. POPL*, 1991.

R. Hickey. The clojure programming language. In *Proc. DLS*, 2008.

R. Hickey. Clojure sequence documentation, February 2015. URL `http://clojure.org/sequences`.

T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proc. PPDP*, 2006.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. POPL*, 1988.

A. M. Maidl, F. Mascarenhas, and R. Ierusalimschy. Typed lua: An optional type system for lua. In *Proc. Dyla*, 2014.

Microsoft. Typescript language specification. Technical Report Version 1.4, 2014.

T. Millstein and C. Chambers. Modular statically typed multimethods. In *Information and Computation*, pages 279–303. Springer-Verlag, 2002.

M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, M. Zenger, and et al. An overview of the scala programming language (second edition). Technical report, EPFL Lausanne, Switzerland, 2006.

T. S. Strickland, S. Tobin-Hochstadt, and M. Felleisen. Practical variable-arity polymorphism. In *Proc. ESOP*, 2009.

S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proc. ICFP*, ICFP '10, 2010.

M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and evaluation of gradual typing for python. In *Proc. DLS*, 2014.

M. Wand. Type inference for record concatenation and multiple inheritance, 1989.