Practical Optional Types for Clojure

Ambrose Bonnaire-Sergeant†, Rowan Davies*, Sam Tobin-Hochstadt†

Indiana University†; Omnia Team, Commonwealth Bank of Australia*

Abstract. Typed Clojure is an optional type system for Clojure, a dynamic language in the Lisp family that targets the JVM. Typed Clojure enables Clojure programmers to gain greater confidence in the correctness of their code via static type checking while remaining in the Clojure world, and has acquired significant adoption in the Clojure community. Typed Clojure repurposes Typed Racket's *occurrence typing*, an approach to statically reasoning about predicate tests, and also includes several new type system features to handle existing Clojure idioms.
In this paper, we describe Typed Clojure and present these type system extensions, focusing on three features widely used in Clojure. First, multimethods provide extensible operations, and their Clojure semantics turns out to have a surprising synergy with the underlying occurrence typing framework. Second, 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. Third, Clojure programmers idiomatically use immutable dictionaries for data structures; Typed Clojure handles this with multiple forms of heterogeneous dictionary types. 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 present a quantitative analysis on the use of type system features in two substantial code bases.

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 TypeScript [15] and Flow [9] for JavaScript, Hack [8] for PHP, and mypy [18] for Python among the optional systems, and Typed Racket [20], Reticulated Python [22], and GradualTalk [1] among gradually-typed systems.[1]

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

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

Fig. 1. A simple Typed Clojure program

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

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

Since Clojure is a language in the Lisp family, we apply the lessons of Typed Racket, an existing gradual type system for Racket, to the core of Typed Clojure, consisting of an extended λ-calculus over a variety of base types shared between all Lisp systems. Furthermore, Typed Racket's *occurrence typing* has proved necessary for type checking realistic Clojure programs.

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

But multimethods are not the only Clojure feature crucial to type checking existing programs. As a language built on the Java Virtual Machine, Clojure provides flexible and transparent access to existing Java libraries, and **Clojure/Java interoperation** is found in almost every significant Clojure code base. Typed Clojure therefore builds in an understanding of the Java type sys-

tem and handles interoperation appropriately. Notably, `null` is a distinct type in Typed Clojure, designed to automatically rule out null-pointer exceptions.

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

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

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

Our main contributions are as follows:

1. We motivate and describe Typed Clojure, an optional type system for Clojure that understands existing Clojure idioms.
2. We present a sound formal model for three crucial type system features: multi-methods, Java interoperability, and heterogenous maps.
3. We evaluate the use of Typed Clojure features on existing Typed Clojure code, including both open source and in-house systems.

The remainder of this paper begins 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 present an empirical analysis of significant code bases written in `core.typed`—the full implementation of Typed Clojure—in Section 5. Finally, we discuss related work and conclude.

2 Overview of Typed Clojure

We now begin a tour of the central features of Typed Clojure, beginning with Clojure itself. Our presentation uses the full Typed Clojure system to illustrate key type system ideas, before studying the core features in detail in Section 3.

2.1 Clojure

Clojure [11] is a Lisp that runs on the Java Virtual Machine with support for concurrent programming and immutable data structures in a mostly-functional

style. Clojure provides easy interoperation with existing Java libraries, with Java values being like 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 Typed Clojure

Here is a simple one-argument function `greet` that takes and returns strings.

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

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

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

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

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

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

Flow analysis Occurrence typing [21] models type-based control flow. In `greetings`, a branch ensures `repeat` is never passed `nil`.

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

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

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

2.3 Java interoperability

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

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

Example 1

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

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

Function arguments default to `Any`, the most permissive type. Ascribing a parameter type allows Typed Clojure to find a specific method.

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

Example 2

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

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

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

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

Since Java-level types lack nullability information, Typed Clojure conservatively assumes Java methods and constructor arguments cannot be passed `nil`—this restriction can be adjusted per method, but method targets can never be `nil`.

In contrast, JVM invariants guarantee constructors return non-null.[2]

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

Example 3

2.4 Multimethods

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

Value-based dispatch This simple multimethod says hello in different languages.

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

Example 4

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

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

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

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

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

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

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

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

2.5 Heterogeneous hash-maps

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

Example 5

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

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

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

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

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

Example 6

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

HMaps in practice The next example is extracted from a production system at CircleCI, a company with a large production Typed Clojure system (Section 5.1 presents a case study and empirical result from this code base).

Example 7

```
(defalias RawKeyPair ; extra keys disallowed
  (HMap :mandatory {:pub RawKey, :priv RawKey},
        :complete? true))
(defalias EncKeyPair ; extra keys disallowed
  (HMap :mandatory {:pub RawKey, :enc-priv EncKey}, :complete? true))

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

As `EncKeyPair` is fully specified, we must remove extra keys like `:priv`.

```
(defn enc-keypair-bad [kp] ; Type error: :priv disallowed
  (assoc kp :enc-priv (encrypt (:priv kp))))
```

2.6 HMaps and multimethods, joined at the hip

HMaps and multimethods are the primary ways for representing and dispatching on data respectively, and so are intrinsically linked. As type system designers, we must search for a compositional approach that can anticipate any combination of these features.

Thankfully, occurrence typing, originally designed for reasoning about `if` tests, provides the compositional approach we need. By extending the system with a handful of rules based on HMaps and other functions, we can automatically cover both easy cases and those that compose rules in arbitrary ways.

Futhermore, this approach extends to multimethod dispatch by reusing occurrence typing's approach to conditionals and encoding a small number of rules to handle the `isa?`-based dispatch. In practice, conditional-based control flow typing extends to multimethod dispatch, and vice-versa.

We first demonstrate a very common, simple dispatch style, then move on to deeper structural dispatching where occurrence typing's compositionality shines.

HMaps and unions Partially specified HMap's with a common dispatch key combine naturally with ad-hoc unions. An `Order` is one of three kinds of HMaps.

```
(defalias Order "A meal order, tracking dessert quantities."
  (U '{:Meal ':lunch, :desserts Int} '{:Meal ':dinner :desserts Int}
     '{:Meal ':combo :meal1 Order :meal2 Order}))
```

The `:Meal` entry is common to each HMap, always mapped to a known keyword singleton type. It's natural to dispatch on the `class` of an instance—it's similarly natural to dispatch on a known entry like `:Meal`.

Example 8

```
(ann desserts [Order -> Int])
(defmulti desserts :Meal)  ; dispatch on :Meal entry
(defmethod desserts :lunch [o] (:desserts d))
(defmethod desserts :dinner [o] (:desserts d))
(defmethod desserts :combo [o]
  (+ (desserts (:meal1 o)) (desserts (:meal2 o))))

(desserts {:Meal :combo, :meal1 {:Meal :lunch :desserts 1},
           :meal2 {:Meal :dinner :desserts 2}}) ;=> 3
```

The `:combo` method is verified to only structurally recur on `Order`s. This is achieved because we learn the argument `o` must be of type `'{:Meal :combo}` since `(isa? (:Meal o) :combo)` is true. Combining this with the fact that `o` is an `Order` eliminates possibility of `:lunch` and `:dinner` orders, simplifying `o` to `'{:Meal ':combo :meal1 Order :meal2 Order}` which contains appropriate arguments for both recursive calls.

Nested dispatch A more exotic dispatch mechanism for `desserts` might be on the `class` of the `:desserts` key. If the result is a number, then we know the `:desserts` key is a number, otherwise the input is a `:combo` meal. We have already seen dispatch on `class` and on keywords in isolation—occurrence typing automatically understands control flow that combines its simple building blocks.

The first method has dispatch value `Long`, a subtype of `Int`, and the second method has `nil`, the sentinel value for a failed map lookup. In practice, `:lunch` and `:dinner` meals will dispatch to the `Long` method, but Typed Clojure infers a slightly more general type due to the definition of `:combo` meals.

Example 9

```
(ann desserts' [Order -> Int])
(defmulti desserts'
  (fn [o :- Order] (class (:desserts o))))
(defmethod desserts' Long [o]
; o :- (U '{:Meal (U ':dinner ':lunch), :desserts Int}
;         '{:Meal ':combo, :desserts Int, :meal1 Order, :meal2 Order})
  (:desserts o))
(defmethod desserts' nil [o]
  ; o :- '{:Meal ':combo, :meal1 Order, :meal2 Order}
  (+ (desserts' (:meal1 o)) (desserts' (:meal2 o))))
```

In the `Long` method, Typed Clojure learns that its argument is at least of type `'{:desserts Long}`—since `(isa? (class (:desserts o)) Long)` must be true. Here the `:desserts` entry *must* be present and mapped to a `Long`—even in a `:combo` meal, which does not specify `:desserts` as present or absent.

In the `nil` method, `(isa? (class (:desserts o)) nil)` must be true—which implies `(class (:desserts o))` is `nil`. Since lookups on missing keys return `nil`, either

- `o` has a `:desserts` entry to `nil`, like `{:desserts nil}`, or
- `o` is missing a `:desserts` entry.

We can express this type with the `:absent-keys` HMap option

```
(U '{:desserts nil} (HMap :absent-keys #{:desserts}))
```

This eliminates non-`:combo` meals since their `'{:desserts Int}` type does not agree with this new information (because `:desserts` is neither `nil` or absent).

From multiple to arbitrary dispatch Clojure multimethod dispatch, and Typed Clojure's handling of it, goes even further, supporting dispatch on multiple arguments via vectors. Dispatch on multiple arguments is beyond the scope of this paper, but the same intuition applies—adding support for multiple dispatch admits arbitrary combinations and nestings of it and previous dispatch rules.

3 A Formal Model of λ_{TC}

After demonstrating the core features of Typed Clojure, we link them together in a formal model called λ_{TC}. Building on occurrence typing, we incrementally add each novel feature of Typed Clojure to the formalism, interleaving presentation of syntax, typing rules, operational semantics, and subtyping.

3.1 Core type system

We start with a review of occurrence typing [21], the foundation of λ_{TC}.

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

Types Types σ or τ include the top type $\top$, *untagged* unions ($\bigcup\ \overrightarrow{\tau}$), singletons ($\mathbf{Val}\, l$), and class instances C. We abbreviate the classes **Boolean** to **B**, **Keyword** to **K**, **Nat** to **N**, **String** to **S**, and **File** to **F**. We also abbreviate the types ($\bigcup$) to $\bot$, ($\mathbf{Val}\,\mathsf{nil}$) to **nil**, ($\mathbf{Val}\,\mathsf{true}$) to **true**, and ($\mathbf{Val}\,\mathsf{false}$) to **false**. The difference between the types ($\mathbf{Val}\, C$) and C is subtle. The former is inhabited by class literals like **K** and the result of (class :a)—the latter by *instances* of classes, like a keyword literal :a, an instance of the type **K**. Function types $x{:}\sigma \xrightarrow[o]{\psi|\psi} \tau$ contain *latent* (terminology from [13]) propositions ψ, object o, and return type τ, which may refer to the function argument x. They are instantiated with the actual object of the argument in applications.

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

Propositions with a logical system In standard type systems, association lists often track the types of variables, like in LC-Let and LC-Local.

$$\text{LC-Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x \mapsto \sigma \vdash e_2 : \tau}{\Gamma \vdash (\text{let}\ [x\ e_1]\ e_2) : \tau} \qquad \text{LC-Local}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

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

Fig. 2. Syntax of Terms, Types, Propositions and Objects

Occurrence typing instead pairs *logical formulas*, that can reason about arbitrary non-empty objects, with a *proof system*. The logical statement σ_x says variable x is of type σ.

$$
\textsc{T0-Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, \sigma_x \vdash e_2 : \tau}{\Gamma \vdash (\text{let } [x\ e_1]\ e_2) : \tau} \qquad\qquad \textsc{T0-Local}\ \frac{\Gamma \vdash \tau_x}{\Gamma \vdash x : \tau}
$$

In T0-Local, $\Gamma \vdash \tau_x$ appeals to the proof system to solve for τ.

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

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

To illustrate, recall Example 8. The parameter o is of type **Order**, written $\mathbf{Order}_{\mathsf{o}}$ as a proposition. In the :combo method, we know (:Meal o) is :combo, based on multimethod dispatch rules. This is written $(\mathbf{Val}\,\mathsf{:combo})_{\mathbf{key}_{\mathsf{:Meal}}(\mathsf{o})}$, pronounced "the :Meal path of variable o is of type ($\mathbf{Val}$:combo)".

To attain the type of o, we must solve for τ in $\Gamma \vdash \tau_{\mathsf{o}}$, under proposition environment $\Gamma = \mathbf{Order}_{\mathsf{o}}, (\mathbf{Val}\,\mathsf{:combo})_{\mathbf{key}_{\mathsf{:Meal}}(\mathsf{o})}$ which deduces τ to be a :combo

meal. The logical system *combines* pieces of type information to deduce more accurate types for lexical bindings—this is explained in Section 3.6.

$$\textsc{T-Local}\ \frac{\Gamma \vdash \tau_x \qquad \sigma = (\cup\ \mathbf{nil}\ \mathbf{false})}{\Gamma \vdash x : \tau\ ;\ \overline{\sigma}_x|\sigma_x\ ;\ x}$$

$$\textsc{T-Abs}\ \frac{\Gamma, \sigma_x \vdash e \Rightarrow e' : \sigma'\ ;\ \psi_+|\psi_-\ ;\ o \qquad \tau = x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \sigma'}{\Gamma \vdash \lambda x^{\sigma}.e \Rightarrow \lambda x^{\sigma}.e' : \tau\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset}$$

$$\textsc{T-If}\ \frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow e_1' : \tau_1\ ;\ {\psi_1}_+|{\psi_1}_-\ ;\ o_1 \\ \Gamma, {\psi_1}_+ \vdash e_2 \Rightarrow e_2' : \tau\ ;\ \psi_+|\psi_-\ ;\ o \\ \Gamma, {\psi_1}_- \vdash e_3 \Rightarrow e_3' : \tau\ ;\ \psi_+|\psi_-\ ;\ o \\ e' = (\text{if}\ e_1'\ e_2'\ e_3')\end{array}}{\Gamma \vdash (\text{if}\ e_1\ e_2\ e_3) \Rightarrow e' : \tau\ ;\ \psi_+|\psi_-\ ;\ o}$$

$$\textsc{T-Kw}\ \frac{}{\Gamma \vdash k : (\mathbf{Val}\, k)\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset} \qquad \textsc{T-Num}\ \frac{}{\Gamma \vdash n : \mathbf{N}\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset}$$

$$\textsc{T-Nil}\ \frac{}{\Gamma \vdash \mathsf{nil} : \mathbf{nil}\ ;\ \mathtt{ff}|\mathtt{tt}\ ;\ \emptyset} \qquad \textsc{T-False}\ \frac{}{\Gamma \vdash \mathsf{false} : \mathbf{false}\ ;\ \mathtt{ff}|\mathtt{tt}\ ;\ \emptyset}$$

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

$$\textsc{T-Str}\ \frac{}{\Gamma \vdash s : \mathbf{S}\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset} \qquad \textsc{T-Class}\ \frac{}{\Gamma \vdash C : (\mathbf{Val}\, C)\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset} \qquad \textsc{T-True}\ \frac{}{\Gamma \vdash \mathsf{true} : \mathbf{true}\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset}$$

$$\textsc{T-Let}\ \frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow e_1' : \sigma\ ;\ {\psi_1}_+|{\psi_1}_-\ ;\ o_1 \qquad \psi' = \overline{(\cup\ \mathbf{nil}\ \mathbf{false})}_x \supset {\psi_1}_+ \\ \psi'' = (\cup\ \mathbf{nil}\ \mathbf{false})_x \supset {\psi_1}_- \qquad \Gamma, \sigma_x, \psi', \psi'' \vdash e_2 \Rightarrow e_2' : \tau\ ;\ \psi_+|\psi_-\ ;\ o\end{array}}{\Gamma \vdash (\text{let}\ [x\ e_1]\ e_2) \Rightarrow (\text{let}\ [x\ e_1']\ e_2') : \tau[o_1/x]\ ;\ \psi_+|\psi_-[o_1/x]\ ;\ o[o_1/x]}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash e \Rightarrow e_1 : x{:}\sigma \xrightarrow[o_f]{{\psi_f}_+|{\psi_f}_-} \tau\ ;\ \psi_+|\psi_-\ ;\ o \qquad \Gamma \vdash e' \Rightarrow e_1' : \sigma\ ;\ {\psi'}_+|{\psi'}_-\ ;\ o'}{\Gamma \vdash (e\ e') \Rightarrow (e_1\ e_1') : \tau[o'/x]\ ;\ {\psi_f}_+|{\psi_f}_-[o'/x]\ ;\ o_f[o'/x]}$$

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

Fig. 3. Core typing rules

Typing judgment We formalize our system following Tobin-Hochstadt and Felleisen [21]. The typing judgment $\Gamma \vdash e \Rightarrow e' : \tau\ ;\ \psi_+|\psi_-\ ;\ o$ says expression e rewrites to e', which is of type τ in the proposition environment Γ, with 'then' proposition ψ_+, 'else' proposition ψ_- and object o.

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

Typing rules The core typing rules are given as Figure 3. We introduce the interesting rules with the complement number predicate as a running example.

$$\lambda \mathsf{d}^{\top}.(\text{if}\ (n?\ \mathsf{d})\ \mathsf{false}\ \mathsf{true}) \tag{1}$$

$$\frac{\exists i.\ \vdash \tau <: \sigma_i}{\vdash \tau <: (\bigcup\ \overrightarrow{\sigma}^i)}\ \text{S-UnionSuper} \qquad \frac{\overrightarrow{\vdash \tau_i <: \sigma}^i}{\vdash (\bigcup\ \overrightarrow{\tau}^i) <: \sigma}\ \text{S-UnionSub} \qquad \vdash x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau <: \mathbf{Fn}\ \text{S-FunMono}$$

$$\vdash C <: \mathbf{Object}\ \text{S-Object} \qquad \vdash (\mathbf{Val}\, C) <: \mathbf{Class}\ \text{S-SClass} \qquad \vdash (\mathbf{Val}\, b) <: \mathbf{B}\ \text{S-SBool}$$

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

$$\vdash \tau <: \tau\ \text{S-Refl} \qquad \vdash \tau <: \top\ \text{S-Top} \qquad \vdash (\mathbf{Val}\, k) <: \mathbf{K}\ \text{S-SKw}$$

Fig. 4. Core subtyping rules

$$\frac{\rho \vdash e_1 \Downarrow v_1 \qquad v_1 \neq \mathsf{false} \qquad v_1 \neq \mathsf{nil} \qquad \rho \vdash e_2 \Downarrow v}{\rho \vdash (\text{if } e_1\ e_2\ e_3) \Downarrow v}\ \text{B-IfTrue}$$

$$\frac{\rho \vdash e_1 \Downarrow \mathsf{false} \ \text{ or } \ \rho \vdash e_1 \Downarrow \mathsf{nil} \qquad \rho \vdash e_3 \Downarrow v}{\rho \vdash (\text{if } e_1\ e_2\ e_3) \Downarrow v}\ \text{B-IfFalse}$$

Fig. 5. Select core semantics

The lambda rule T-Abs introduces $\sigma_x = \top_{\mathsf{d}}$ to check the body. With $\Gamma = \top_{\mathsf{d}}$, T-If first checks the test $e_1 = (n?\ \mathsf{d})$ via the T-App rule, with three steps.

First, in T-App the operator $e = n?$ is checked with T-Const, which uses δ_τ (Figure 7, dynamic semantics in the supplemental material) to type constants. $n?$ is a predicate over numbers, and *class* returns its argument's class.

Resuming $(n?\ \mathsf{d})$, in T-App the operand $e' = \mathsf{d}$ is checked with T-Local as

$$\Gamma \vdash \mathsf{d} : \top\ ;\ \overline{(\cup\ \mathbf{nil}\ \mathbf{false})}_{\mathsf{d}} | (\cup\ \mathbf{nil}\ \mathbf{false})_{\mathsf{d}}\ ;\ \mathsf{d} \tag{2}$$

which encodes the type, proposition, and object information about variables.

Finally, the T-App rule substitutes the operand's object o' for the parameter x in the latent type, propositions, and object.

$$\Gamma \vdash (n?\ \mathsf{d}) : \mathbf{B}\ ;\ \mathbf{N}_{\mathsf{d}} | \overline{\mathbf{N}}_{\mathsf{d}}\ ;\ \emptyset \tag{3}$$

To demonstrate, the 'then' proposition—in T-App $\psi_+[o'/x]$—substitutes the latent 'then' proposition of $\delta_\tau(n?)$ with d, giving $\mathbf{N}_x[\mathsf{d}/x] = \mathbf{N}_{\mathsf{d}}$.

To check the branches of (if $(n?\ \mathsf{d})$ false true), T-If introduces $\psi_{1+} = \mathbf{N}_{\mathsf{d}}$ to check $e_2 = \mathsf{false}$, and $\psi_{1-} = \overline{\mathbf{N}}_{\mathsf{d}}$ to check $e_3 = \mathsf{true}$. The branches are first checked with T-False and T-True respectively, the T-Subsume premises $\Gamma, \psi_+ \vdash {\psi'}_+$ and $\Gamma, \psi_- \vdash {\psi'}_-$ allow us to pick compatible propositions for both branches.

$$\Gamma, \mathbf{N}_{\mathsf{d}} \vdash \mathsf{false} : \mathbf{B}\ ;\ \overline{\mathbf{N}}_{\mathsf{d}} | \mathbf{N}_{\mathsf{d}}\ ;\ \emptyset$$
$$\Gamma, \overline{\mathbf{N}}_{\mathsf{d}} \vdash \mathsf{true} : \mathbf{B}\ ;\ \overline{\mathbf{N}}_{\mathsf{d}} | \mathbf{N}_{\mathsf{d}}\ ;\ \emptyset$$

Finally T-Abs assigns a type to the overall function:

$$\vdash \lambda \mathsf{d}^{\top}.(\text{if } (n? \mathsf{d}) \text{ false true}) : \mathsf{d}{:}\top \xrightarrow[\emptyset]{\overline{\mathbf{N}}_{\mathsf{d}}|\mathbf{N}_{\mathsf{d}}} \mathbf{B} \ ; \ \mathbb{tt}|\mathbb{ff} \ ; \ \emptyset$$

Subtyping Figure 4 presents subtyping as 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**. Function types are subtypes of **Fn**. All types except for **nil** are subtypes of **Object**, so $\top$ is similar to $(\bigcup$ **nil Object**$)$. Function subtyping is contravariant left of the arrow—latent propositions, object and result type are covariant. Subtyping for untagged unions is standard.

Operational semantics We define the dynamic semantics for λ_{TC} in a big-step style using an environment, following [21]. 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.14) however a notable difference is nil is a false value, which affects the semantics of if (Figure 5).

3.2 Java Interoperability

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

We extend the syntax in Figure 6 with Java field lookups and calls to methods and constructors. To prevent ambiguity, field accesses are written $(.\ e\ fld)$ and method calls $(.\ e\ (mth\ \overrightarrow{e}))$.

In Example 1, `(.getParent (new File "a/b"))` translates to

$$(.\ (\text{new}\ \ \mathbf{F}\ \text{“a/b”})\ (\text{getParent})) \tag{4}$$

But both the constructor and method are unresolved. We introduce *non-reflective* expressions for specifying exact Java overloads.

$$(.\ (\text{new}_{[\mathbf{S}]}\ \ \mathbf{F}\ \ \text{“a/b”})\ (\text{getParent}^{\mathbf{F}}_{[[],\mathbf{S}]})) \tag{5}$$

From the left, the one-argument constructor for **F** takes a **S**, and the getParent method of **F** takes zero arguments and returns a **S**.

We now walk through this conversion.

Constructors First we check and convert (new **F** “a/b”) to $(\text{new}_{[\mathbf{S}]}$ **F** “a/b”$)$. The T-New typing rule checks and rewrites constructors. To check (new **F** “a/b”) we first resolve the constructor overload in the class table—there is at most one to simplify presentation. With $C_1 = \mathbf{S}$, we convert to a nilable type the argument with $\tau_1 = (\bigcup$ **nil S**$)$ and type check “a/b” against τ_1. Typed Clojure defaults to

$$
\begin{array}{lll}
e & ::= \ldots (.\ e\ fld) \mid (.\ e\ (mth\ \overrightarrow{e})) \mid (\text{new}\ C\ \overrightarrow{e}) & \text{Expressions} \\
 & \mid (.\ e\ fld^{C}_{C}) \mid (.\ e\ (mth^{C}_{[[\overrightarrow{C}],C]}\ \overrightarrow{e})) \mid (\text{new}_{[\overrightarrow{C}]}\ C\ \overrightarrow{e}) & \text{Non-reflective Expressions} \\
v & ::= \ldots \mid C\ \{\overrightarrow{fld : v}\} & \text{Values} \\
ce & ::= \{\mathsf{m} \mapsto \{\overrightarrow{mth \mapsto [[\overrightarrow{C}], C]}\}, \mathsf{f} \mapsto \{\overrightarrow{fld \mapsto C}\}, \mathsf{c} \mapsto \{[\overrightarrow{C}]\}\} & \text{Class descriptors} \\
\mathcal{CT} & ::= \{\overrightarrow{C \mapsto ce}\} & \text{Class Table}
\end{array}
$$

T-New
$$\frac{[\overrightarrow{C_i}] \in \mathcal{CT}[C][\mathsf{c}] \qquad \overrightarrow{\mathsf{JT}_{\mathsf{nil}}(C_i) = \tau_i} \qquad \overrightarrow{\Gamma \vdash e_i \Rightarrow e_i' : \tau_i} \qquad \mathsf{JT}(C) = \tau}{\Gamma \vdash (\text{new}\ C\overrightarrow{e_i}) \Rightarrow (\text{new}_{[\overrightarrow{C_i}]}\ C\ \overrightarrow{e_i}) : \tau\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset}$$

T-Method
$$\frac{\Gamma \vdash e \Rightarrow e' : \sigma \qquad \mathsf{TJ}(\sigma) = C_1 \qquad mth \mapsto [[\overrightarrow{C_i}], C_2] \in \mathcal{CT}[C_1][\mathsf{m}] \qquad \overrightarrow{\mathsf{JT}_{\mathsf{nil}}(C_i) = \tau_i} \qquad \overrightarrow{\Gamma \vdash e_i \Rightarrow e_i' : \tau_i} \qquad \mathsf{JT}_{\mathsf{nil}}(C_2) = \tau \qquad \vdash \sigma <: \mathbf{Object}}{\Gamma \vdash (.\ e\ (mth\overrightarrow{e_i})) \Rightarrow (.\ e'\ (mth^{C_1}_{[[\overrightarrow{C_i}],C_2]}\ \overrightarrow{e_i})) : \tau\ ;\ \mathtt{tt}|\mathtt{tt}\ ;\ \emptyset}$$

T-Field
$$\frac{\Gamma \vdash e \Rightarrow e' : \sigma \quad \vdash \sigma <: \mathbf{Object} \quad \mathsf{TJ}(\sigma) = C_1 \quad fld \mapsto C_2 \in \mathcal{CT}[C_1][\mathsf{f}] \quad \mathsf{JT}_{\mathsf{nil}}(C_2) = \tau}{\Gamma \vdash (.\ e\ fld) \Rightarrow (.\ e'\ fld^{C_1}_{C_2}) : \tau\ ;\ \mathtt{tt}|\mathtt{tt}\ ;\ \emptyset}$$

$$
\begin{array}{llll}
\mathsf{JT}_{\mathsf{nil}}(\mathbf{Void}) = \mathbf{nil} & & \mathsf{JT}(\mathbf{Void}) = \mathbf{nil} & \mathsf{TJ}(\tau) = C \quad \text{if} \vdash \tau <: \mathsf{JT}_{\mathsf{nil}}(C) \\
\mathsf{JT}_{\mathsf{nil}}(C) = (\bigcup\ \mathbf{nil}\ C) & & \mathsf{JT}(C) = C &
\end{array}
$$

B-Field
$$\frac{\rho \vdash e \Downarrow v \qquad \mathsf{JVM}_{\mathsf{getstatic}}[C_1, v_1, fld, C_2] = v}{\rho \vdash (.\ e\ fld^{C_1}_{C_2}) \Downarrow v}$$

B-New
$$\frac{\overrightarrow{\rho \vdash e_i \Downarrow v_i} \qquad \mathsf{JVM}_{\mathsf{new}}[C_1, [\overrightarrow{C_i}], [\overrightarrow{v_i}]] = v}{\rho \vdash (\text{new}_{[\overrightarrow{C_i}]}\ C\ \overrightarrow{e_i}) \Downarrow v}$$

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

Fig. 6. Java Interoperability Syntax, Typing and Operational Semantics

$$
\begin{array}{ll}
\delta_\tau(class) = x{:}\top \xrightarrow[\mathbf{class}(x)]{\mathtt{tt}|\mathtt{tt}} (\bigcup\ \mathsf{nil}\ \mathbf{Class}) \\
\delta_\tau(n?) = x{:}\top \xrightarrow[\emptyset]{\mathbf{N}_x|\overline{\mathbf{N}}_x} \mathbf{B}
\end{array}
$$

Fig. 7. Constant typing

allowing non-nilable arguments, but this can be overridden, so we model the more general case. The return Java type $\mathbf{F}$ is converted to a non-nil Typed Clojure type $\tau = \mathbf{F}$ for the return type, and the propositions say constructors can never be false—constructors can never produce the internal boolean value that Clojure uses for false, or nil. Finally, the constructor rewrites to $(\text{new}_{[\mathbf{S}]}\ \mathbf{F}\ \text{“a/b”})$.

Methods Next we convert $(.\ (\text{new}_{[\mathbf{S}]}\ \mathbf{F}\ \text{“a/b”})\ (\text{getParent}))$ to the non-reflective expression $(.\ (\text{new}_{[\mathbf{S}]}\ \mathbf{F}\ \text{“a/b”})\ (\text{getParent}^{\mathbf{F}}_{[[],\mathbf{S}]}))$. We use T-Method to check $(.\ (\text{new}_{[\mathbf{S}]}\ \mathbf{F}\ \text{“a/b”})\ (\text{getParent}))$, which checks unresolved methods. We verify the target type $\sigma = \mathbf{F}$ is non-nil before erasing **nil** by converting to a Java type $C_1 = \mathbf{F}$. The specific overload is chosen from the class table based on C_1—there is at most one. Then $(\text{new}_{[\mathbf{S}]}\ \mathbf{F}\ \text{“a/b”})$ is checked against a nilable conversion of C_1, $\tau_1 = (\bigcup\ \mathbf{nil}\ \mathbf{F})$, which succeeds by the previous rule. The nilable return type $\tau = (\bigcup\ \mathbf{nil}\ \mathbf{S})$ is given, and—finally—the entire expression rewrites to expression 5.

The T-Field rule is included in Figure 6, and is like T-Method, but without arguments.

The evaluation rules B-Field, B-New and B-Method (Figure 6) simply evaluate their arguments and call the relevant JVM operation, which we do not model—Section 4 states our exact assumptions. There are no evaluation rules for reflective Java interoperability, since there are no typing rules that rewrite to reflective calls.

3.3 Multimethod preliminaries: isa?

We now consider the isa? operation, a core part of the dispatch mechanism for multimethods. Recalling the Section 2.4 examples, isa? is a subclassing test for classes, otherwise an equality test. The key component of the T-IsA rule (Figure 8) is the $\mathsf{IsAProps}$ metafunction, used to calculate the propositions for isa? tests.

As an example, the expression (isa? $(\mathit{class}\ x)\ \mathbf{K}$) 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 8). IsA explicitly handles classes in the second case.

3.4 Multimethods

Figure 8 presents *immutable* multimethods without default methods to ease presentation. Figure 9 is the translation of Example 4 to λ_{TC}.

To check (defmulti $x{:}\mathbf{K} \to \mathbf{S}\ \lambda x^{\mathbf{K}}.x$), we note (defmulti $\sigma\ e$) creates a multimethod with *interface type* σ, and dispatch function e of type σ', producing a value of type $(\mathbf{Multi}\ \sigma\ \sigma')$. The T-DefMulti typing rule checks the dispatch function, and verifies both the interface and dispatch type's domain agree. Our

$$
\begin{array}{lll}
e & ::= \ldots \mid (\text{defmulti } \tau\ e) \mid (\text{defmethod } e\ e\ e) \mid (\text{isa? } e\ e) & \text{Expressions} \\
v & ::= \ldots \mid [v,t]_{\mathsf{m}} & \text{Values} \\
t & ::= \{\overrightarrow{v \mapsto v}\} & \text{Dispatch tables} \\
\sigma,\tau & ::= \ldots \mid (\mathbf{Multi}\,\tau\ \tau) & \text{Types}
\end{array}
$$

T-DefMulti

$$\frac{\sigma = x{:}\tau \xrightarrow[o]{\psi_+|\psi_-} \tau' \qquad \sigma' = x{:}\tau \xrightarrow[o']{\psi'_+|\psi'_-} \tau'' \qquad \Gamma \vdash e \Rightarrow e' : \sigma'}{\Gamma \vdash (\text{defmulti } \sigma\ e) \Rightarrow (\text{defmulti } \sigma\ e') : (\mathbf{Multi}\,\sigma\ \sigma')\ ;\ \mathbb{tt}|\mathbb{ff}\ ;\ \emptyset}$$

T-DefMethod

$$\frac{\begin{array}{c}\tau_m = x{:}\tau \xrightarrow[o]{\psi_+|\psi_-} \sigma \qquad \tau_d = x{:}\tau \xrightarrow[o']{\psi'_+|\psi'_-} \sigma' \qquad \Gamma \vdash e_m \Rightarrow e'_m : (\mathbf{Multi}\,\tau_m\ \tau_d) \\ \Gamma \vdash e_v \Rightarrow e'_v : \tau_v \qquad \mathsf{IsAProps}(o', \tau_v) = \psi''_+|\psi''_- \\ \Gamma, \tau_x, \psi''_+ \vdash e_b \Rightarrow e'_b : \sigma\ ;\ \psi_+|\psi_-\ ;\ o \qquad e' = (\text{defmethod } e'_m\ e'_v\ \lambda x^\tau.e'_b)\end{array}}{\Gamma \vdash (\text{defmethod } e_m\ e_v\ \lambda x^\tau.e_b) \Rightarrow e' : (\mathbf{Multi}\,\tau_m\ \tau_d)\ ;\ \mathbb{tt}|\mathbb{ff}\ ;\ \emptyset}$$

T-IsA

$$\frac{\Gamma \vdash e \Rightarrow e_1 : \sigma\ ;\ \psi'_+|\psi'_-\ ;\ o \qquad \Gamma \vdash e' \Rightarrow e'_1 : \tau \qquad \mathsf{IsAProps}(o,\tau) = \psi_+|\psi_-}{\Gamma \vdash (\text{isa? } e\ e') \Rightarrow (\text{isa? } e_1\ e'_1) : \mathbf{B}\ ;\ \psi_+|\psi_-\ ;\ \emptyset}$$

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

S-PMultiFn

$$\frac{\vdash \sigma_t <: x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau \qquad \vdash \sigma_d <: x{:}\sigma \xrightarrow[o']{\psi'_+|\psi'_-} \tau'}{\vdash (\mathbf{Multi}\,\sigma_t\ \sigma_d) <: x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau}$$

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-DefMulti

$$\frac{\rho \vdash e \Downarrow v_d \qquad v = [v_d, \{\}]_{\mathsf{m}}}{\rho \vdash (\text{defmulti } \tau\ e) \Downarrow v}$$

B-DefMethod

$$\frac{\rho \vdash e \Downarrow [v_d, t]_{\mathsf{m}} \qquad \rho \vdash e' \Downarrow v_v \qquad \rho \vdash e_f \Downarrow v_f \qquad v = [v_d, t[v_v \mapsto v_f]]_{\mathsf{m}}}{\rho \vdash (\text{defmethod } e\ e'\ e_f) \Downarrow v}$$

$\mathsf{GM}(t, v_e) = v_f$ if $\overrightarrow{v_{fs}} = \{v_f\}$ where $\overrightarrow{v_{fs}} = \{v_f | v_k \mapsto v_f \in t \text{ and } \mathsf{IsA}(v_e, v_k) = \mathsf{true}\}$
$\mathsf{GM}(t, v_e) = \mathsf{err}$ otherwise

B-IsA

$$\frac{\rho \vdash e_1 \Downarrow v_1 \qquad \rho \vdash e_2 \Downarrow v_2 \qquad \mathsf{IsA}(v_1, v_2) = v}{\rho \vdash (\text{isa? } e_1\ e_2) \Downarrow v}$$

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

B-BetaMulti

$$\frac{\rho \vdash e \Downarrow [v_d, t]_{\mathsf{m}} \quad \rho \vdash e' \Downarrow v' \quad \rho \vdash (v_d\ v') \Downarrow v_e \quad \mathsf{GM}(t, v_e) = v_f \quad \rho \vdash (v_f\ v') \Downarrow v}{\rho \vdash (e\ e') \Downarrow v}$$

Fig. 8. Multimethod Syntax, Typing and Operational Semantics

$$
\begin{array}{l}
(\text{let } [hi_0 \ (\text{defmulti } x{:}\,\mathbf{K} \xrightarrow[\emptyset]{\mathtt{tt}\,|\,\mathtt{tt}} \mathbf{S} \ \ \lambda x^{\mathbf{K}}.x)] \\
\quad (\text{let } [hi_1 \ (\text{defmethod } hi_0 \ \mathsf{:en} \ \lambda x^{\mathbf{K}}.\text{“}\mathsf{hello}\text{”})] \\
\quad\quad (\text{let } [hi_2 \ (\text{defmethod } hi_1 \ \mathsf{:fr} \ \lambda x^{\mathbf{K}}.\text{“}\mathsf{bonjour}\text{”})] \\
\quad\quad\quad (hi_2 \ \mathsf{:en}))))
\end{array}
$$

Fig. 9. Multimethod example

example checks with $\tau = \mathbf{K}$, interface type $\sigma = x{:}\,\mathbf{K} \to \mathbf{S}$, dispatch function type $\sigma' = x{:}\,\mathbf{K} \xrightarrow[x]{\mathtt{tt}\,|\,\mathtt{tt}} \mathbf{K}$, and overall type $(\mathbf{Multi}\, x{:}\,\mathbf{K} \to \mathbf{S}\ x{:}\,\mathbf{K} \xrightarrow[x]{\mathtt{tt}\,|\,\mathtt{tt}} \mathbf{K})$.

Next, show how to check (defmethod hi_0 :en $\lambda x^{\mathbf{K}}$.“hello”). The expression (defmethod $e_m\ e_v\ e_f$) creates a new multimethod that extends multimethod e_m's dispatch table, mapping dispatch value e_v to method e_f. The T-DefMulti typing rule checks e_m is a multimethod with dispatch function type τ_d, then calculates the extra information we know based on the current dispatch value ${\psi''}_+$, which is assumed when checking the method body. Our example checks with e_m being of type $(\mathbf{Multi}\, x{:}\,\mathbf{K} \to \mathbf{S}\ x{:}\,\mathbf{K} \xrightarrow[x]{\mathtt{tt}\,|\,\mathtt{tt}} \mathbf{K})$ with $o' = x$ and $\tau_v = (\mathbf{Val}\,\mathsf{:en})$. Then ${\psi''}_+ = (\mathbf{Val}\,\mathsf{:en})_x$ by $\mathsf{IsAProps}(x, (\mathbf{Val}\,\mathsf{:en})) = (\mathbf{Val}\,\mathsf{:en})_x|\overline{(\mathbf{Val}\,\mathsf{:en})}_x$. Since $\tau = \mathbf{K}$, we check the method body with $\mathbf{K}_x, (\mathbf{Val}\,\mathsf{:en})_x \vdash$ “hello” $:\mathbf{S}\ ;\ \mathtt{tt}\,|\,\mathtt{tt}\ ;\ \emptyset$. Finally from the interface type τ_m, we know $\psi_+ = \psi_- = \mathtt{tt}$, and $o = \emptyset$, which also agrees with the method body, above.

Subtyping Multimethods are functions, via S-PMultiFn, which says a multimethod can be upcast to its interface type. Multimethod call sites are then handled by T-App via T-Subsume. Other rules are given in Figure 8.

Semantics Multimethod definition semantics are also given in Figure 8. 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.

The overall dispatch mechanism is summarised by B-BetaMulti. First the dispatch function v_d is applied to the argument v' to obtain the dispatch value v_e. Based on v_e, the GM metafunction (Figure 8) extracts a method v_f from the method table t and applies it to the original argument for the final result.

3.5 Precise Types for Heterogeneous maps

Figure 10 presents heterogeneous map types. The type $(\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})$ contains $\mathcal{M}$, a map of *present* entries (mapping keywords to types), $\mathcal{A}$, a set of keyword keys that 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. The partially specified map of `lunch` in Example 6 is written $(\mathbf{HMap}^{\mathcal{P}}\{(\mathbf{Val}\,\mathsf{:en})\ \mathbf{S}, (\mathbf{Val}\,\mathsf{:fr})\ \mathbf{S}\}\ \{\})$. We abbreviate this type as **Lu** in this section. The type of the fully specified map `breakfast` in Example 5 elides the

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

T-AssocHMap

$$\frac{\Gamma \vdash e \Rightarrow (\text{assoc } e'\ e'_k\ e'_v) : (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}) \quad \Gamma \vdash e_k \Rightarrow e'_k : (\mathbf{Val}\, k) \quad \Gamma \vdash e_v \Rightarrow e'_v : \tau \quad k \notin \mathcal{A}}{\Gamma \vdash (\text{assoc } e\ e_k\ e_v) \Rightarrow (\text{assoc } e'\ e'_k\ e'_v) : (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}[k \mapsto \tau]\ \mathcal{A}) \ ;\ \mathtt{tt}|\mathtt{ff} \ ;\ \emptyset}$$

T-GetHMap

$$\frac{\Gamma \vdash e \Rightarrow e' : (\bigcup\ \overrightarrow{(\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})}^{i}) \ ;\ \psi_{1+}|\psi_{1-} \ ;\ o \quad \Gamma \vdash e_k \Rightarrow e'_k : (\mathbf{Val}\, k) \quad \overrightarrow{\mathcal{M}[k] = \tau}^{i}}{\Gamma \vdash (\text{get } e\ e_k) \Rightarrow (\text{get } e'\ e'_k) : (\bigcup\ \overrightarrow{\tau}^{i}) \ ;\ \mathtt{tt}|\mathtt{tt} \ ;\ \mathbf{key}_k(x)[o/x]}$$

T-GetHMapAbsent

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

T-GetHMapPartialDefault

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

S-HMapMono

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

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-HMap

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

B-Assoc

$$\frac{\begin{array}{c}\rho \vdash e \Downarrow m \qquad \rho \vdash e_k \Downarrow k \\ \rho \vdash e_v \Downarrow v_v\end{array}}{\rho \vdash (\text{assoc } e\ e_k\ e_v) \Downarrow m[k \mapsto v_v]}$$

B-Get

$$\frac{\rho \vdash e \Downarrow m \qquad \rho \vdash e' \Downarrow k \qquad k \in dom(m)}{\rho \vdash (\text{get } e\ e') \Downarrow m[k]}$$

B-GetMissing

$$\frac{\rho \vdash e \Downarrow m \qquad \rho \vdash e' \Downarrow k \qquad k \notin dom(m)}{\rho \vdash (\text{get } e\ e') \Downarrow \mathsf{nil}}$$

Fig. 10. HMap Syntax, Typing and Operational Semantics

$\mathsf{restrict}(\tau, \sigma) = \bot$ if $\nexists v.\ \vdash v : \tau \ ;\ \psi \ ;\ o$ and $\vdash v : \sigma \ ;\ \psi' \ ;\ o'$
$\mathsf{restrict}(\tau, \sigma) = \tau$ if $\vdash \tau <: \sigma$
$\mathsf{restrict}(\tau, \sigma) = \sigma$ otherwise

$\mathsf{remove}(\tau, \sigma) = \bot$ if $\vdash \tau <: \sigma$
$\mathsf{remove}(\tau, \sigma) = \tau$ otherwise

Fig. 11. Restrict and remove

absent entries, written $(\mathbf{HMap}^{\mathcal{C}}\{(\mathbf{Val}\,\mathsf{:en})\ \mathbf{S}, (\mathbf{Val}\,\mathsf{:fr})\ \mathbf{S}\})$. We abbreviate this type as **Bf** in this section. 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 metavariable m ranges over the runtime value of maps $\{\overrightarrow{k \mapsto v}\}$, usually written $\{\overrightarrow{k\ v}\}$. We only provide syntax for the empty map literal, however when convenient we abbreviate non-empty map literals to be a series of assoc operations on the empty map. We restrict lookup and extension to keyword keys.

How to check A mandatory lookup is checked by T-GetHMap.

$$\lambda \mathsf{b}^{\mathbf{Bf}}.(\text{get } \mathsf{b}\ \mathsf{:en})$$

The result type is **S**, and the return object is $\mathbf{key}_{\mathsf{:en}}(\mathsf{b})$. The object $\mathbf{key}_k(x)[o/x]$ is a symbolic representation for a keyword lookup of k in o. The substitution for x handles the case where o is empty.

$$\mathbf{key}_k(x)[y/x] = \mathbf{key}_k(y) \qquad \mathbf{key}_k(x)[\emptyset/x] = \emptyset$$

An absent lookup is checked by T-GetHMapAbsent.

$$\lambda \mathsf{b}^{\mathbf{Bf}}.(\text{get } \mathsf{b}\ \mathsf{:bocce})$$

The result type is **nil**, and the return object is $\mathbf{key}_{\mathsf{:bocce}}(\mathsf{b})$.

A lookup that is not present or absent is checked by T-GetHMapPartialDefault.

$$\lambda \mathsf{u}^{\mathbf{Lu}}.(\text{get } \mathsf{u}\ \mathsf{:bocce})$$

The result type is $\top$, and the return object is $\mathbf{key}_{\mathsf{:bocce}}(\mathsf{u})$. Notice propositions are erased once they enter a HMap type.

For presentational reasons, lookups on unions of HMaps are only supported in T-GetHMap and each element of the union must contain the relevant key.

$$\lambda \mathsf{u}^{(\bigcup\ \mathbf{Bf}\,\mathbf{Lu})}.(\text{get } \mathsf{u}\ \mathsf{:en})$$

The result type is **S**, and the return object is $\mathbf{key}_{\mathsf{:en}}(\mathsf{u})$. However, lookups of :bocce on $(\bigcup\ \mathbf{Bf}\,\mathbf{Lu})$ maps are unsupported. This restriction still allows us to check many of the examples in Section 2—in particular we can check Example 8, as :Meal is in common with both HMaps, but cannot check Example 9 because a :combo meal lacks a :desserts entry.

Extending a map with T-AssocHMap preserves its completeness.

$$\lambda \mathsf{b}^{\mathbf{Bf}}.(\text{assoc } \mathsf{b}\ \mathsf{:au}\ \text{“beans”})$$

The result type is $(\mathbf{HMap}^{\mathcal{C}}\{(\mathbf{Val}\,\mathsf{:en})\ \mathbf{S}, (\mathbf{Val}\,\mathsf{:fr})\ \mathbf{S}, (\mathbf{Val}\,\mathsf{:au})\ \mathbf{S}\})$, a complete map. T-AssocHMap also enforces $k \notin \mathcal{A}$ to prevent badly formed types.

Subtyping Subtyping for HMaps designate **Map** as a common supertype for all HMaps. S-HMap says that HMaps are subtypes 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 semantics for get and assoc are straightforward.

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

Fig. 12. Type update (the metavariable ν ranges over τ and $\overline{\tau}$ (without variables), $\vdash \mathbf{nil} \not<: \overline{\tau}$ when $\vdash \mathbf{nil} <: \tau$, see Figure 11 for restrict and remove.)

3.6 Proof system

The occurrence typing proof system uses standard propositional logic, except for where nested information is combined. This is handled by L-Update:

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

It says under Γ, if object $\pi'(x)$ is of type τ, and an extension $\pi(\pi'(x))$ is of possibly-negative type ν, then update $(\tau,\ \nu,\ \pi)$ is $\pi'(x)$'s type under Γ.

Recall Example 8. Solving $\mathbf{Order}_{\mathsf{o}}, (\mathbf{Val}\,{:}\mathsf{combo})_{\mathbf{key}_{:\mathsf{Meal}}(\mathsf{o})} \vdash \tau_{\mathsf{o}}$ uses L-Update, where $\pi = \epsilon$ and $\pi' = [\mathbf{key}_{:\mathsf{Meal}}]$.

$$\Gamma \vdash \mathsf{update}(\mathbf{Order}, (\mathbf{Val}\,{:}\mathsf{combo}), [\mathbf{key}_{:\mathsf{Meal}}])_{\mathsf{o}}$$

Since **Order** is a union of HMaps, we structurally recur on the first case of update (Figure 12), which preserves π. Each initial recursion hits the first HMap case, since there is some τ such that $\mathcal{M}[k] = \tau$ and $\mathcal{E}$ accepts partial maps $\mathcal{P}$.

To demonstrate, :lunch meals are handled by the first HMap case and update to $(\mathbf{HMap}^{\mathcal{P}}\,\mathcal{M}[(\mathbf{Val}\,\text{:Meal}) \mapsto \sigma']\ \{\})$ where $\sigma' = \mathsf{update}\ ((\mathbf{Val}\,\text{:lunch}), (\mathbf{Val}\,\text{:combo}), \epsilon)$ and $\mathcal{M} = \{(\mathbf{Val}\,\text{:Meal}) \mapsto (\mathbf{Val}\,\text{:lunch}), (\mathbf{Val}\,\text{:desserts}) \mapsto \mathbf{N}\}$. σ' updates to $\bot$ via the penultimate update case, because restrict $((\mathbf{Val}\,\text{:lunch}), (\mathbf{Val}\,\text{:combo})) = \bot$ by the first restrict case. The same happens to :dinner meals, leaving just the :combo HMap.

In Example 9, $\Gamma \vdash \mathsf{update}(\mathbf{Order}, \mathbf{Long}, [\mathbf{class}, \mathbf{key}_{\text{:desserts}}])_{\mathsf{o}}$ updates the argument in the **Long** method. This recurs twice for each meal to handle the **class** path element.

We describe the other update cases. The first **class** case updates to C if *class* returns $(\mathbf{Val}\,C)$. The second $\mathbf{key}_k$ case detects contradictions in absent keys. The third $\mathbf{key}_k$ case updates unknown entries to be mapped to τ or absent. The fourth $\mathbf{key}_k$ case updates unknown entries to be *present* when they do not overlap with **nil**.

4 Metatheory

We prove type soundness following [21]. Our model is extended to include errors and a *wrong* value, and we prove well-typed programs do not go wrong; this is therefore a stronger theorem than proved by [21].

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

Assumption 1 ($\mathsf{JVM}_{\mathsf{new}}$)**.** *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*

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

For the purposes of our soundness proof, we require that all values are *consistent*. Consistency (defined in the supplemental material) states that the types of closures are well-scoped—they do not claim propositions about variables hidden in their closures.

We can now state our main lemma and soundness theorem. The metavariable α ranges over v, err and *wrong*. Proofs are deferred to A.8.

Lemma 1. *If* $\Gamma \vdash e' \Rightarrow e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$*,* $\rho \models \Gamma$*,* 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. $\vdash v \Rightarrow v : \tau\ ;\ {\psi'}_+|{\psi'}_-\ ;\ o'$ *for some* ${\psi'}_+$*,* ${\psi'}_-$ *and* o'*, and*

   4. v *is consistent with* ρ*, or*
- $\rho \vdash e \Downarrow \mathsf{err}$.

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

5 Experience

Typed Clojure is implemented as a Clojure library—`core.typed`. It provides preliminary integration with the Clojure compilation pipeline, primarily to resolve Java interoperability, however most usages are entirely optionally typed.

The `core.typed` implementation extends this paper in several key areas to handle checking real Clojure code, including an implementation of Typed Racket's variable-arity polymorphism [19], and support for other Clojure idioms like datatypes and protocols.

5.1 Evaluation

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

To evaluate this hypothesis, we analyzed two existing Typed Clojure code bases, one from the open-source community, and one from a company that uses Typed Clojure in production. For our data gathering, we instrumented the Typed Clojure type checker to record how often various features were used.

feeds2imap feeds2imap is an open source library written in Typed Clojure. It provides an RSS reader using the *java.mail* framework.

Of 11 typed namespaces containing 825 lines of code, there are 32 Java interactions. The majority are method calls, consisting of 20 (62%) instance methods and 5 (16%) static methods. The rest consists of 1 (3%) static field access, and 6 (19%) constructor calls—there are no instance field accesses.

There are 27 lookup operations on HMap types, of which 20 (74%) resolve to mandatory entries, 6 (22%) to optional entries, and 1 (4%) is an unresolved lookup. No lookups involved fully specified maps.

From 93 `def` expressions in typed code, 52 (56%) are checked, with a rate of 1 Java interaction for 1.6 checked top-level definitions, and 1 HMap lookup to 1.9 checked top-level definitions. That leaves 41 (44%) unchecked vars, mainly due to partially complete porting to Typed Clojure, but in some cases due to unannotated third-party libraries.

No typed multimethods are defined or used. Of 18 total type aliases, 7 (39%) contained one HMap type, and none contained unions of HMaps—on further inspection there was no HMap entry used to dictate control flow, often handled by multimethods. This is unusual in our experience, and is perhaps explained by feeds2imap mainly wrapping wrapping existing *javax.mail* functionality.

CircleCI CircleCI provides continuous integration services built with a mixture of open- and closed-source. Typed Clojure was used at CircleCI in production systems for two years [6], maintaining 87 namespaces and 19,000 lines of code.

The CircleCI code base contains 11 checked multimethods. All 11 dispatch functions are on a HMap key containing a keyword, in a similar style to Example 8. Correspondingly, all 89 methods are associated with a keyword dispatch value. The argument type was in all cases a single HMap type, however, rather than a union type. In our experience from porting other libraries, this is unusual.

Of 328 lookup operations on HMaps, 208 (64%) resolve to mandatory keys, 70 (21%) to optional keys, 20 (6%) to absent keys, and 30 (9%) lookups are unresolved. Of 95 total type aliases defined with `defalias`, 62 (65%) involved one or more HMap types. Out of 105 Java interactions, 26 (25%) are static methods, 36 (34%) are instance methods, 38 (36%) are constructors, and 5 (5%) are static fields. 35 methods are overriden to return non-nil, and 1 method overridden to accept nil—suggesting that `core.typed` disallowing `nil` as a method argument by default is justified.

Of 464 checked top-level definitions (which consists of 57 `defmethod` calls and 407 `def` expressions), 1 HMap lookup occurs per 1.4 top-level definitions, and 1 Java interaction occurs every 4.4 top-level definitions.

From 1834 `def` expressions in typed code, only 407 (22%) were checked. That leaves 1427 (78%) which have unchecked definitions, either by an explicit `:no-check` annotation or `tc-ignore` to suppress type checking, or the `warn-on-unannotated-vars` option, which skips `def` expressions that lack expected types via `ann`. From a brief investigation, reasons include unannotated third-party libraries, work-in-progress conversions to Typed Clojure, unsupported Clojure idioms, and hard-to-check code.

Lessons Based on our empirical survey, HMaps and Java interoperability support are vital features used on average more than once per typed function. Multimethods are less common in our case studies. The CircleCI code base contains only 26 multimethods total in 55,000 lines of mixed untyped-typed Clojure code, a low number in our experience.

5.2 Further challenges

After a 2 year trial, the second case study decided to disabled type checking [7]. They were supportive of the fundamental ideas presented in this paper, but primarily cited issues with the checker implementation in practice and would reconsider type checking if they were resolved.

Performance Rechecking files with transitive dependencies is expensive since all dependencies must be rechecked. We conjecture caching type state will significantly improve re-checking performance, though preserving static soundness in the context of arbitrary code reloading is a largely unexplored area.

Library annotations Annotations for external code are rarely available, so a large part of the untyped-typed porting process is reverse engineering libraries.

Unsupported idioms While the current set of features is vital to checking Clojure code, as we have seen in Section 5.1 there is still much work to do. In practice, common Clojure functions are too polymorphic for the current implementation or theory to account for. For example, `comp` provides variable-arity function composition, and further research is required to check callsites.

6 Related Work

Multimethods [16] and collaborators present a sequence of systems [3, 4, 16] 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.

Record Types Row polymorphism [23, 2, 10], 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 [5] can track similar invariants as HMaps with types for JS objects. They must deal with mutable objects, they feature refinement types and strong updates to the heap to track changes to objects.

TeJaS [12], another type system for JavaScript, also supports similar HMaps, with the ability to record the presence and absence of entries, but lacks a compositional flow-checking approach like occurrence typing.

Typed Lua [14] has *table types* which track entries in a mutable Lua table. Typed Lua changes the dynamic semantics of Lua to accommodate mutability: Typed Lua raises a runtime error for lookups on missing keys—HMaps consider lookups on missing keys normal.

Java Interoperability in Statically Typed Languages Scala [17] has nullable references for compatibility with Java. Programmers must manually check for `null` as in Java to avoid null-pointer exceptions.

Other optional and gradual type systems Several other gradual type systems have been developed for existing dynamically-typed languages. Reticulated Python [22] is an experimental gradually typed system for Python, implemented as a source-to-source translation that inserts dynamic checks at language boundaries and supporting Python's first-class object system. Clojure's nominal classes avoids the need to support first-class object system in Typed Clojure, however HMaps offer an alternative to the structural objects offered by Reticulated. Similarly, GradualTalk [1] offers gradual typing for SmallTalk, with nominal classes.

Optional types have been adopted in industry, including Hack [8], and Flow [9] and TypeScript [15], two extensions of JavaScript. These systems support limited forms of occurrence typing, and do not include the other features we present.

7 Conclusion

Optional type systems must be designed with close attention to the language that they are intended to work for. We have therefore designed Typed Clojure, an optionally-typed version of Clojure, with a type system that 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 multi-methods. In addition, Typed Clojure supports crucial features such as heterogeneous maps and Java interoperability while integrating these features into the core type system. Not only are each of these features important in isolation to Clojure and Typed Clojure programmers, but they must fit together smoothly to ensure that existing Clojure programs are easy to convert to Typed Clojure.

The result is a sound, expressive, and useful type system which, as 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 used in the Clojure community among both enthusiasts and professional programmers and receives contributions from many developers.

Our empirical analysis of existing Typed Clojure programs bears out our design choices. Multimethods, Java interoperation, and heterogeneous maps are indeed common in both Clojure and Typed Clojure, meaning that our type system must accommodate them. Furthermore, they are commonly used together, and the features of each are mutually reinforcing. Additionally, the choice to make Java's `null` explicit in the type system is validated by the many Typed Clojure programs that specify non-nullable types.

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 enabling sound gradual typing, but also integrating Typed Clojure into the Clojure tool so that its information can be communicated to the compiler.

Finally, our case study, evaluation, 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 features and idioms and develop new type checking approaches.

References

1. Allende, E., Callau, O., Fabry, J., Tanter, É., Denker, M.: Gradual typing for smalltalk. Science of Computer Programming 96, 52–69 (2014)
2. Cardelli, L., Mitchell, J.C.: Operations on records. In: Mathematical Structures in Computer Science. pp. 3–48 (1991)
3. Chambers, C.: Object-oriented multi-methods in cecil. In: Proc. ECOOP (1992)
4. Chambers, C., Leavens, G.T.: Typechecking and modules for multi-methods. In: Proc. OOPSLA (1994)
5. Chugh, R., Herman, D., Jhala, R.: Dependent types for javascript. In: Proc. OOPSLA (2012)
6. CircleCI: Why were supporting typed clojure, and you should too! (September 2013), `http://blog.circleci.com/supporting-typed-clojure/`
7. CircleCI; O'Morain, M.: Why were no longer using core.typed (September 2015), `http://blog.circleci.com/why-were-no-longer-using-core-typed/`
8. Facebook: Hack language specification. Tech. rep. (2014)
9. Facebook: Flow language specification. Tech. rep. (2015)
10. Harper, R., Pierce, B.: A record calculus based on symmetric concatenation. In: Proc. POPL (1991)
11. Hickey, R.: The clojure programming language. In: Proc. DLS (2008)
12. Lerner, B.S., Politz, J.G., Guha, A., Krishnamurthi, S.: Tejas: Retrofitting type systems for javascript. In: Proceedings of the 9th Symposium on Dynamic Languages. pp. 1–16. DLS '13, ACM, New York, NY, USA (2013), `http://doi.acm.org/10.1145/2508168.2508170`
13. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proc. POPL (1988)
14. Maidl, A.M., Mascarenhas, F., Ierusalimschy, R.: Typed lua: An optional type system for lua. In: Proc. Dyla (2014)
15. Microsoft: Typescript language specification. Tech. Rep. Version 1.4 (2014)
16. Millstein, T., Chambers, C.: Modular statically typed multimethods. In: Information and Computation. pp. 279–303. Springer-Verlag (2002)
17. Odersky, M., Cremet, V., Dragos, I., Dubochet, G., Emir, B., McDirmid, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Spoon, L., Zenger, M., et al.: An overview of the scala programming language (second edition). Tech. rep., EPFL Lausanne, Switzerland (2006)
18. mypy project, T.: mypy, `http://mypy-lang.org/`
19. Strickland, T.S., Tobin-Hochstadt, S., Felleisen, M.: Practical variable-arity polymorphism. In: Proc. ESOP (2009)
20. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: Proc. POPL (2008)
21. Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: Proc. ICFP. ICFP '10 (2010)
22. Vitousek, M.M., Kent, A.M., Siek, J.G., Baker, J.: Design and evaluation of gradual typing for python. In: Proc. DLS (2014)
23. Wand, M.: Type inference for record concatenation and multiple inheritance (1989)