Practical Optional Types for Clojure

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's type system 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 report on a quantitative analysis of the use of the features described in this paper 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 and Flow for JavaScript, Hack for PHP, 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. Clojure is 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

[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".

```
; a function annotation for `pname` multimethod.
; Input: non-nil (null) File or String, via union
; Ouput: nilable String
(ann pname [(U File String) -> (U nil String)])
(defmulti pname class) ; multimethod on arg's class
(defmethod pname String [s] ; String implementation
  (pname (new File s))) ; JVM constructors non-nil
(defmethod pname File [f] ; File implementation
  (.getName f)) ; JVM method target `f` verified
                ; non-nil, but return is nilable
(pname "STAINS/JELLY") ; :- (U nil Str)
;=> "JELLY"
```

Figure 1. A simple Typed Clojure program

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.

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 system and handles interoperation appropriately. Notably, Typed Clojure avoids conflating Java's `null` with all other types, leading to automatic type-enforced absence of null-pointer exceptions.

A simple example of these features in combination is given in figure 1. Here, the `pname` multimethod dispatches on the `class` of the argument—when it is a `String`, the first method implementation is called, for a Java `File`, the second. The method for `String`

uses Java interoperation to call a `File` constructor, returning a non-nil `File` instance—the method for `File` invokes the `.getName` method with a non-nil target, returning a nilable type. Typed Clojure uses type information to resolve JVM method or constructor overloads, avoiding expensive runtime reflective calls.

Finally, when not working with existing Java classes and objects, the most common Clojure data structure is the dictionary, a high-performance immutable table which Clojure programmers repurpose for all manner of data types. Simply treating them as uniformly-typed key-value mappings would be entirely insufficient for existing programs and programming styles. Instead, Typed Clojure provides a flexible **heterogenous map** type, in which mandatory, optional, and absent keys can be specified.

While these features may seem disparate, they are unified in important ways. First, they all leverage the type system mechanisms that Typed Clojure inherits 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, and more significantly, they are the 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 the three crucial type system features that Typed Clojure relies on: 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 discuss the full implementation of Typed Clojure, dubbed `core.typed`, which extends the formal model in many ways, and empirical analysis of significant code bases written in `core.typed` 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. 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 Typed Clojure

Here is a simple program in Typed Clojure. We define `greet` as a one-argument function taking and returning a string.

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

Strings are accepted, but providing `nil` (Clojure's name for Java's `null`) is a static type error—`nil` is not a string in Typed Clojure.

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

Unions We can make the annotation more permissive with *ad-hoc unions* to allow `nil`.

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

All Clojure values are true except `nil` and `false`, so the comma is only added when the argument is non-nil.

Typed Clojure guarantees that well-typed code cannot dereference the null-pointer. This is especially important for Clojure programs—`nil` is treated like any other distinct datum in Clojure, and its status as one of only two false values means it is a common choice to indicate "nothing" or "failure".

Flow analysis Typed Clojure uses occurrence typing (Tobin-Hochstadt and Felleisen 2010) to model type-based control flow. In `greetings`, a `when` expression ensures `repeat` is never passed a `nil` argument.

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

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

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

2.3 Java interoperability

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

The following Typed Clojure program constructs a new `File` instance and calls the `getParent` method on the result, returning a string `"a"`, inferred as nullable.

Example 1

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

Typed Clojure helps the Clojure compiler avoid expensive reflective calls, however if a specific constructor, method, or field cannot be found based on the static types of its arguments, a type error is thrown.

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

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

Example 2

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

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
                  ;   java.io.File/getParent on type
                  ;   (U nil File).
```

Since Java-level types do not provide information about nullability, by default Typed Clojure conservatively assumes that Java method and constructor arguments, such as `"a/b"` provided to the `File` constructor in Example 1, must be *non-nullable*, but this can be configured for particular calls as needed. The target of method invocations is always non-nullable.

Any Java method returning a reference can also return `null` — Typed Clojure rejects programs that assume otherwise.

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

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

Example 3

```
(fn [s :- String] :- File
  (new File s))
```

2.4 Multimethods

Multimethods are a kind of extensible function, and they are widely used to define Clojure operations. It combines a *dispatch function* with one or more *methods*.

Value-based dispatch This simple multimethod says hello in different languages, as specified by a keyword argument.

Example 4

```
(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
```

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—the arguments are then passed to the method to finally return a value for the entire expression.

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

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

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

Class-based dispatch For class values, multimethods can choose methods based on subclassing relationships. Recall the multimethod from figure 1, reproduced here.

```
(ann pname [(U File String) -> (U nil String)])
(defmulti pname class)
(defmethod pname String [s] (pname (new File s)))
(defmethod pname File [f] (.getName f))
```

The dispatch function `class` dictates whether the `String` or `File` method is chosen. The multimethod dispatch rules use `isa?`, a hybrid predicate which is a subclassing check for classes and an equality check for other values.

```
(isa? (identity :en) :en) ;=> true
(isa? (identity :en) :fr) ;=> false
(isa? (class "STAINS/JELLY") String) ;=> true
(isa? (class "STAINS/JELLY") Object) ;=> true
(isa? (class (new File "JELLY")) String) ;=> false
```

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.

In our example, `(pname "STAINS/JELLY")` chooses the method for `String` because `(isa? (class "STAINS/JELLY") String)` is true and `(isa? (class "STAINS/JELLY") File)` is false. The `String` method body `(pname (new File "STAINS/JELLY"))` chooses the `File` method for opposite reasons, resulting in

```
(.getName (new File "STAINS/JELLY")) ; :- (U nil Str)
;=> "JELLY"
```

2.5 Heterogeneous hash-maps

Beyond primitives and Java objects, the most common Clojure data structure is the immutable hash-map, typically with keyword keys. This structure is the primary way to represent compound data in Clojure programs.

Hash-maps are accessed with the `get` function:

Example 5

```
(def breakfast
  {:en "waffles" :fr "croissants"})

(get breakfast :en)     ; :- Str
;=> "waffles"
```

Additionally, keywords are functions that look themselves up in a map. Missing keys produce `nil`.

```
(:fr breakfast)         ; :- Str
;=> "croissants"
(:bocce breakfast) ; :- nil
;=> nil
```

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

In Typed Clojure, *HMap types* describe the most common usages of keyword-keyed maps.

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

The inferred type for `breakfast` holds two kinds of information—the known entries–`:en` and `:fr`—and their types, which are `:mandatory`, and that no other key is present, since `:complete?` is `true`.

HMap types default to partial specification—`:complete?` defaults to `false`. The HMap shorthand `'{:en Str :fr Str}` omits information about absent keys, only providing information on `:mandatory` keys.

Example 6

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

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 and empirical result from this code base).

Example 7

```
(defalias RawKeyPair
  "Unencrypted keypair -- extra keys disallowed"
  (HMap :mandatory {:public-key RawKey,
                    :private-key RawKey},
        :complete? true))

(defalias EncKeyPair
  "Encrypted keypair -- extra keys disallowed"
  (HMap :mandatory {:public-key RawKey,
                    :enc-private-key EncKey},
        :complete? true))

(ann enc-keypair [RawKeyPair -> EncKeyPair])
(defn enc-keypair "Encrypt an unencrypted keypair"
  [{pk :private-key :as kp}] ; original map is kp
  (assoc
    ; remove unencrypted private key
    (dissoc kp :private-key)
    ; add encrypted private key
    :enc-private-key (encrypt pk)))
```

If we forget to remove the unencrypted private key, a type error is given, because `EncKeyPair` is fully specified.

```
(ann enc-keypair-bad [RawKeyPair -> EncKeyPair])
(defn enc-keypair-bad
  [{pk :private-key :as kp}]
  (assoc kp :enc-private-key (encrypt pk)))
  ; Type Error:
  ;   Expected EncKeyPair, given
  ;   (HMap :mandatory {:enc-private-key EncKey
  ;                     :private-key RawKey
  ;                     :public-key RawKey}
  ;         :complete? true)
```

The extra `:private-key` entry does not match `EncKeyPair`, so a type error is raised.

2.6 HMaps and multimethods, joined at the hip

Since HMaps are the primary way of specifying the structure of data in Clojure, and multimethods are the primary tool for dispatching on data, they are inevitably linked. There are infinite ways of both structuring and dispatching on data, so we cannot hope to merely add a set of special case rules for handling these features. Instead, as type system designers, we must search for a compositional approach.

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 simple rules in arbitrary ways.

Futhermore, this approach extends to multimethod dispatch—the primitive branching mechanism works like the humble `if` conditional. Only a small number of rules are needed to encode the `isa?`-based dispatch, themselves made of simple pieces. In practice, this means that 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   ; define type abbreviation
  "A meal order, tracking dessert quantities."
  (U '{:Meal ':lunch ; keyword singleton type
       :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 "Total desserts per order."
          :Meal)  ; dispatch on :Meal entry
; destructuring reads backwards, d is :desserts entry
(defmethod desserts :lunch [{d :desserts}] d)
(defmethod desserts :dinner [{d :desserts}] d)
(defmethod desserts :combo [{m1 :meal1 m2 :meal2}]
  (+ (desserts m1) (desserts m2)))

(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)` must be true. Combining `'{:Meal :combo}` 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 An equally valid dispatch mechanism for `desserts` would be on the `class` of the `:desserts` key. We have already seen dispatch on `class` and on keywords in isolation—occurrence

typing automatically understands control flow that combines its simple building blocks.

In the first method, the dispatch value is the class `Long`, a subtype of `Int`, and the second method has dispatch value `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 [{d :desserts :as o}]
  ; o :- (U '{:Meal ':lunch :desserts Int}
  ;          '{:Meal ':dinner :desserts Int}
  ;          '{:Meal ':combo :desserts Int
  ;            :meal1 Order :meal2 Order})
  d)
(defmethod desserts' nil [{m1 :meal1 m2 :meal2 :as o}]
  ; o :- '{:Meal ':combo :meal1 Order :meal2 Order}
  (+ (desserts' m1) (desserts' m2)))
```

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. In this method, we deduce the entry `:desserts` *must* be a present and mapped to a `Long`, even in a `:combo` meal which does not state `: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, like `{}`.

Equivalently, we learn `o` is at least of type

```
(U '{:desserts nil}
   ; :absent-keys, a set of known absent entries
   (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 mapped to `nil` or absent).

From multimethod to multiple 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 automatically allows arbitrary combinations and nestings of it and previous simple dispatch rules.

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}. 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

Our presentation will start with a review of occurrence typing (Tobin-Hochstadt and Felleisen 2010), the foundation of λ_{TC}.

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

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

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. 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 over functions, which map bindings to values.

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 $(\mathbf{Val}\ \mathbf{K})$ and **K** is subtle. The former is inhabited by the class literal **K** and the result of $(\mathit{class}\ :\mathsf{a})$—the latter by keywords like :a. Function types $x{:}\sigma \xrightarrow[o]{\psi \mid \psi} \tau$ contain *latent* (terminology from (Lucassen and Gifford 1988)) 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 To reason about nested expressions, each expression is associated with a symbolic representation called an *object*. For example, variable m has object m; $(\mathbf{class}\ (:\mathsf{lunch}\ \mathsf{m}))$ 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 of looking up a keyword k respectively.

Propositions with a logical system In standard type systems, *type environments* track variables.

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

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.

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

In T0-Local, $\Gamma \vdash \tau_{\pi(x)}$ appeals to the proof system to solve τ.

We further extend logical statements to *propositional logic*. Figure 2 describes the syntax for propositions ψ, consisting of positive

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

Figure 4. Constant typing

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 ($\mathtt{tt}$) and impossible ($\mathtt{ff}$) 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. We know the parameter o is of type **Order**, written $\mathbf{Order}_{\mathsf{o}}$ as a proposition. In checking the :combo method, we also assume (: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 (**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.

The full judgment We formalize our type system following Tobin-Hochstadt and Felleisen (2010). 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 ψ'_+, ψ'_- 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.

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

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

First, in T-App the operator $e = n?$ is checked with T-Const, which uses δ_τ (Figure 4, 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 as previously discussed.

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

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

S-REFL $\quad \vdash \tau <: \tau$

S-TOP $\quad \vdash \tau <: \top$

S-UNIONSUPER $\quad \dfrac{\exists i.\ \vdash \tau <: \sigma_i}{\vdash \tau <: (\bigcup \overrightarrow{\sigma}^i)}$

S-UNIONSUB $\quad \dfrac{\overrightarrow{\vdash \tau_i <: \sigma}^i}{\vdash (\bigcup \overrightarrow{\tau}^i) <: \sigma}$

S-FUNMONO $\quad \vdash x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau <: \mathbf{Fn}$

S-OBJECT $\quad \vdash C <: \mathbf{Object}$

S-SCLASS $\quad \vdash (\mathbf{Val}\ C) <: \mathbf{Class}$

S-SBOOL $\quad \vdash (\mathbf{Val}\ b) <: \mathbf{B}$

S-SKW $\quad \vdash (\mathbf{Val}\ k) <: \mathbf{K}$

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

Figure 5. Core subtyping rules

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 demonstrate 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, then T-Subsume.

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

The T-Subsume premises $\Gamma, \psi_+ \vdash \psi'_+$ and $\Gamma, \psi_- \vdash \psi'_-$ allow us to pick compatible propositions for both branches.

Finally T-Abs concludes, using the T-If outputs, with

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

Subtyping Figure 5 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 $(\bigcup\ \mathbf{nil}\ \mathbf{Object})$. 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 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.14) however a notable difference is nil is a false value, which affects the semantics of if:

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

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

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 $(.\ e\ (mth\ \overrightarrow{e}))$.

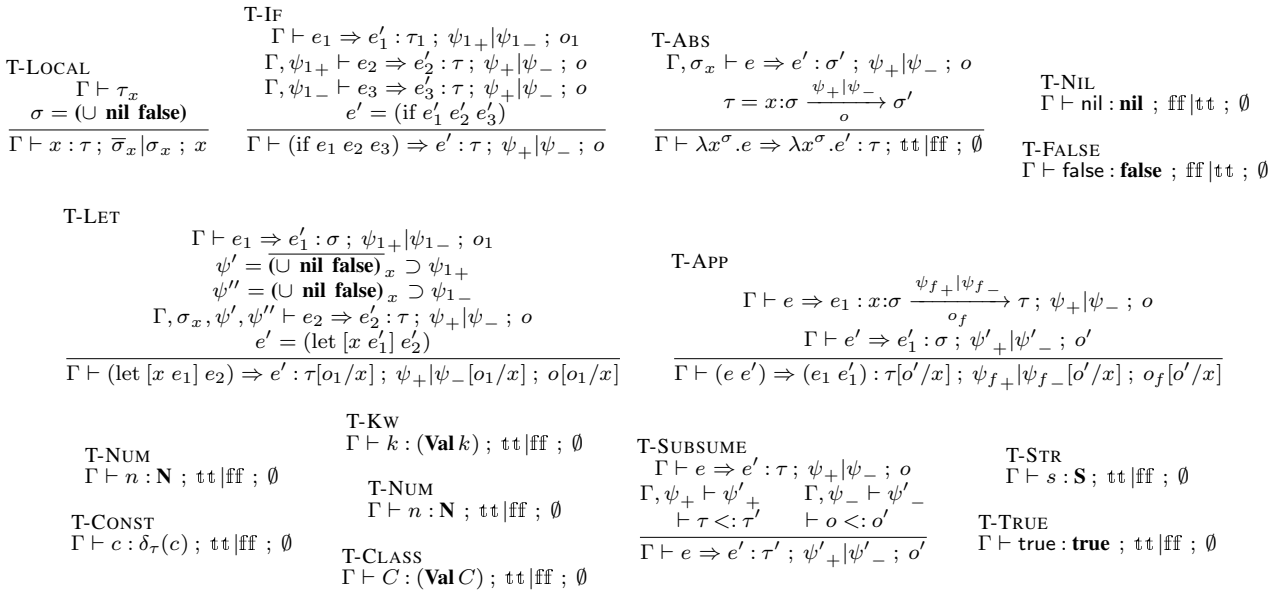

Figure 3. Core typing rules

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 the conversion from unresolved expression 4 to resolved expression 5.

Constructors First we check and convert (new **F** “a/b”) to $(\text{new}_{[\mathbf{S}]}\ \mathbf{F}\ \text{“a/b”})$. 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 allowing non-nilable arguments, but this can be overridden, so we model the more general case. The return Java type **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 $(.\ (\text{new}_{[\mathbf{S}]}\ \mathbf{F}\ \text{“a/b”})\ (\text{getParent}^{\mathbf{F}}_{[[],\mathbf{S}]}))$. 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, $\tau_1 = (\bigcup$ **nil F**$)$, which succeeds by the previous rule. The nilable return type $\tau = (\bigcup$ **nil 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 7) is the IsAProps metafunction, used to calculate the propositions for isa? tests.

As an example, (isa? $(\mathit{class}\ x)$ **K**) has the true and false propositions $\mathsf{IsAProps}(\textbf{class}(x), (\textbf{Val}\,\mathbf{K})) = \mathbf{K}_x|\overline{\mathbf{K}}_x$, 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 7). IsA explicitly handles classes in the second case.

3.4 Multimethods

Figure 7 presents *immutable* multimethods without default methods to ease presentation. Below is the translation of Example 4 to λ_{TC}.

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

We now show to check $(\text{defmulti } x{:}\mathbf{K} \to \mathbf{S}\ \lambda x^{\mathbf{K}}.x)$. The expression $(\text{defmulti } \sigma\ e)$ creates a multimethod with *interface type* σ, and dispatch function e of type σ', producing a value of type $(\textbf{Multi}\ \sigma\ \sigma')$. The T-DefMulti typing rule simply checks the dispatch function, and verifies both the interface and dispatch type's domain agree. Our 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 $(\textbf{Multi}\, x{:}\mathbf{K} \to \mathbf{S}\ x{:}\mathbf{K} \xrightarrow[x]{\mathtt{tt}|\mathtt{tt}} \mathbf{K})$.

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

T-NEW

$$\frac{\overrightarrow{\mathsf{JT}_{\mathsf{nil}}(C_i) = \tau_i} \qquad \overrightarrow{[\overrightarrow{C_i}] \in \mathcal{CT}[C][\mathsf{c}]} \quad \overrightarrow{\Gamma \vdash e_i \Rightarrow e_i' : \tau_i} \qquad \mathsf{JT}(C) = \tau}{\Gamma \vdash (\text{new}\ C\overrightarrow{e_i}) \Rightarrow (\text{new}_{[\overrightarrow{C_i}]}\ C\ \overrightarrow{e_i'}) : \tau\ ;\ \mathbb{tt}|\mathbb{ff}\ ;\ \emptyset}$$

T-METHOD

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow e' : \sigma \qquad \vdash \sigma <: \textbf{Object} \\ \mathsf{TJ}(\sigma) = C_1 \qquad \overrightarrow{mth \mapsto [[\overrightarrow{C_i}], C_2] \in \mathcal{CT}[C_1][\mathsf{m}]} \\ \overrightarrow{\mathsf{JT}_{\mathsf{nil}}(C_i) = \tau_i} \qquad \overrightarrow{\Gamma \vdash e_i \Rightarrow e_i' : \tau_i} \qquad \mathsf{JT}_{\mathsf{nil}}(C_2) = \tau\end{array}}{\Gamma \vdash (.\, e\ (mth\overrightarrow{e_i})) \Rightarrow (.\, e'\ (mth^{C_1}_{[[\overrightarrow{C_i}],C_2]}\ \overrightarrow{e_i'})) : \tau\ ;\ \mathbb{tt}|\mathbb{tt}\ ;\ \emptyset}$$

T-FIELD

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow e' : \sigma \qquad \vdash \sigma <: \textbf{Object} \\ \mathsf{TJ}(\sigma) = C_1 \qquad fld \mapsto C_2 \in \mathcal{CT}[C_1][\mathsf{f}] \qquad \mathsf{JT}_{\mathsf{nil}}(C_2) = \tau\end{array}}{\Gamma \vdash (.\, e\ fld) \Rightarrow (.\, e'\ fld^{C_1}_{C_2}) : \tau\ ;\ \mathbb{tt}|\mathbb{tt}\ ;\ \emptyset}$$

$$\begin{array}{lll} \mathsf{JT}_{\mathsf{nil}}(\textbf{Void}) = \textbf{nil} & & \mathsf{JT}(\textbf{Void}) = \textbf{nil} \\ \mathsf{JT}_{\mathsf{nil}}(C) = (\bigcup\ \textbf{nil}\ C) & & \mathsf{JT}(C) = C \end{array}$$

$$\mathsf{TJ}(\tau) = C \qquad \text{if} \vdash \tau <: \mathsf{JT}_{\mathsf{nil}}(C)$$

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

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

Figure 6. Java Interoperability Syntax, Typing and Operational Semantics

Next, show how to check $(\text{defmethod}\ hi_0\ :\mathsf{en}\ \lambda x^{\mathbf{K}}.\text{“hello”})$. The expression $(\text{defmethod}\ e_m\ e_v\ e_f)$ 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 $(\textbf{Multi}\, x{:}\mathbf{K} \to \mathbf{S}\, x{:}\mathbf{K} \xrightarrow[x]{\mathbb{tt}|\mathbb{tt}} \mathbf{K})$ with $o' = x$ and $\tau_v = (\textbf{Val}\,{:}\mathsf{en})$. Then ${\psi''}_+ = (\textbf{Val}\,{:}\mathsf{en})_x$ by $\mathsf{IsAProps}(x, (\textbf{Val}\,{:}\mathsf{en})) = (\textbf{Val}\,{:}\mathsf{en})_x|\overline{(\textbf{Val}\,{:}\mathsf{en})}_x$. Since $\tau = \mathbf{K}$, we check the method body with $\mathbf{K}_x, (\textbf{Val}\,{:}\mathsf{en})_x \vdash \text{“hello”} : \mathbf{S}\ ;\ \mathbb{tt}|\mathbb{tt}\ ;\ \emptyset$. Finally from the interface type τ_m, we know $\psi_+ = \psi_- = \mathbb{tt}$, and $o = \emptyset$, which also agrees with the method body, above.

Subtyping Multimethods are also functions, which is encoded via S-PMultiFn. This rule says a multimethod can be upcast to its interface type. This means multimethod call sites can be handled by T-App via T-Subsume. Other rules are given in in Figure 7.

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

T-DEFMULTI

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

T-DEFMETHOD

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

T-ISA

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

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

S-PMULTIFN

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

S-PMULTI

$$\frac{\vdash \sigma <: \sigma' \qquad \vdash \tau <: \tau'}{\vdash (\textbf{Multi}\, \sigma\, \tau) <: (\textbf{Multi}\, \sigma'\, \tau')}$$

S-MULTIMONO

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

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

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

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

B-ISA

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

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

Figure 7. Multimethod Syntax, Typing and Operational Semantics

Semantics Multimethod definition semantics are also given in Figure 7. B-DefMulti creates a multimethod with the given dispatch

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

T-GetHMap

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

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

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

T-AssocHMap

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

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

$$\frac{\rho \vdash e \Downarrow m \qquad \rho \vdash e_k \Downarrow k \qquad \rho \vdash e_v \Downarrow v_v \qquad v = m[k \mapsto v_v]}{\rho \vdash (\text{assoc } e\ e_k\ e_v) \Downarrow v}$$

B-Get

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

B-GetMissing

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

Figure 8. HMap Syntax, Typing and Operational Semantics

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.

B-BetaMulti

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

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 7) 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 8 presents heterogeneous map types. The partially specified map of `lunch` in Example 6 is written

$$(\mathbf{HMap}^{\mathcal{P}}\{(\mathbf{Val}\ \text{:en})\ \mathbf{S}, (\mathbf{Val}\ \text{:fr})\ \mathbf{S}\}\ \{\})$$

We abbreviate this type as **Lu** in this section. 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 type of the fully specified map `breakfast` in Example 5 elides the absent entries, written

$$(\mathbf{HMap}^{\mathcal{C}}\{(\mathbf{Val}\ \text{:en})\ \mathbf{S}, (\mathbf{Val}\ \text{:fr})\ \mathbf{S}\})$$

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.

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

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

$$\begin{array}{lcl} \mathbf{key}_k(x)[y/x] & = & \mathbf{key}_k(y) \\ \mathbf{key}_k(x)[\emptyset/x] & = & \emptyset \end{array}$$

An absent lookup is checked by T-GetHMapAbsent.

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

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

A lookup of a key that is neither present or absent is checked by T-GetHMapPartialDefault.

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

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

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

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

The result type is **S**, and the return object is $\mathbf{key}_{\text{:en}}(\mathsf{u})$. However, lookups on :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.

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

The result type is $(\mathbf{HMap}^{\mathcal{C}}\{(\mathbf{Val}\ \text{:en})\ \mathbf{S}, (\mathbf{Val}\ \text{:fr})\ \mathbf{S}, (\mathbf{Val}\ \text{:au})\ \mathbf{S}\})$, which is complete, like its input. 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 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 semantics for get and assoc are straightforward. If the entry is missing, B-GetMissing produces nil.

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

$$
\begin{array}{lcll}
\mathsf{restrict}(\tau, \sigma) & = & \bot & \text{if } \nexists v. \vdash v : \tau\ ;\ \psi\ ;\ o \text{ and} \vdash v : \sigma\ ;\ \psi'\ ;\ o' \\
\mathsf{restrict}(\tau, \sigma) & = & \tau & \text{if} \vdash \tau <: \sigma \\
\mathsf{restrict}(\tau, \sigma) & = & \sigma & \text{otherwise}
\end{array}
\qquad
\begin{array}{lcll}
\mathsf{remove}(\tau, \sigma) & = & \bot & \text{if} \vdash \tau <: \sigma \\
\mathsf{remove}(\tau, \sigma) & = & \tau & \text{otherwise}
\end{array}
$$

Figure 9. Type update (the metavariable ν ranges over τ and $\overline{\tau}$ (without variables), $\vdash \mathbf{nil} \not<: \overline{\tau}$ when $\vdash \mathbf{nil} <: \tau$)

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

Figure 10. Proof system update rule (full rules Figure A.20)

3.6 Proof system

The proof system for occurrence typing uses standard propositional logic, except for where nested information is combined. This is handled by the L-Update rule (Figure 10)—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 (τ, ν, π) 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 9), 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 hit the first HMap case and update to $(\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}[(\mathbf{Val}\,\mathsf{:Meal}) \mapsto \sigma']\ \{\})$ where $\sigma' = \mathsf{update}\ ((\mathbf{Val}\,\mathsf{:lunch}), (\mathbf{Val}\,\mathsf{:combo}), \epsilon)$ and $\mathcal{M} = \{(\mathbf{Val}\,\mathsf{:Meal}) \mapsto (\mathbf{Val}\,\mathsf{:lunch}), (\mathbf{Val}\,\mathsf{:desserts}) \mapsto \mathbf{N}\}$. σ' updates to $\bot$ via the penultimate update case, because restrict $((\mathbf{Val}\,\mathsf{:lunch}), (\mathbf{Val}\,\mathsf{:combo})) = \bot$ by the first restrict case. The same happens to :dinner meals, leaving just the :combo HMap.

In Example 9, $\Gamma \vdash \mathsf{update}(\mathbf{Order}, \mathbf{Long}, [\mathbf{class}, \mathbf{key}_{\mathsf{:desserts}}])_{\mathsf{o}}$ 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 Tobin-Hochstadt and Felleisen (2010). 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 Tobin-Hochstadt and Felleisen (2010).

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. (Assumptions for other operations are 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.

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* $\Gamma \vdash e' \Rightarrow e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$*,* $\rho \models \Gamma$*,* 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*
- $\rho \vdash e \Downarrow \mathsf{err}$.

Proof. By induction on the derivation of $\rho \vdash e \Downarrow \alpha$. (Full proof given as lemma A.8). □

We can now state our soundness theorems.

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'

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

5. Experience

Typed Clojure is implemented as a Clojure library—`core.typed`. In contrast to Racket, Clojure does not provide extension points to

```
(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 11. Type annotation and example call of `swap!`

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, and not as an integral part of the Clojure compilation process.

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. Also, programs cannot depend on Typed Clojure's typs, meaning that type-based optimisation is impossible.

5.1 Further Extensions

In addition to the key features we present in this paper, `core.typed` supports other extensions to handle additional Clojure features.

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 `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 11), 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).

Typed-untyped interoperation Currently, interactions between typed and untyped Clojure code are unchecked which can violate the expectations of Typed Clojure. Gradual typing (Tobin-Hochstadt and Felleisen 2006; Siek and Taha 2006) ensures sound interoperability between typed and untyped code by enforcing invariants of the type system via run-time contracts. We hope to add support for gradual typing in the future.

5.3 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 has been used at CircleCI in production 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 determined that CircleCI has a Clojure code base of 382 namespaces comprising around 55,000 lines, excluding tests, including 87 namespaces and around 19,000 lines of typed code. Some of the type-annotated definitions were so annotated by the first author and contributed back to CircleCI.

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, it's clear that the features we consider are vital—they are used on average more than once per typed function. Furthermore, as we have seen in the CircleCI case study, the combination of heterogenous maps and multimethods is pervasive. The data therefore validates our choice of a type system that supports expressive multimethod definition and acknolwedges the relationship between these seemingly-distinct features.

The other lesson from our case studies and from 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 hope 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 runtime cost, have been adopted in industry, including Hack for PHP (Facebook 2014), and Flow (Facebook 2015) and TypeScript (Microsoft 2014), two extensions of JavaScript. These systems support forms of occurrence typing, but not in the generality presented here, nor do they 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 multimethods. 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

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.

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`.

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.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

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

S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 964–974, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X. . URL `http://doi.acm.org/10.1145/1176617.1176755`.

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.

A. Soundness for Typed Clojure

Assumption A.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.*

Assumption A.2 ($\mathsf{JVM}_{\mathsf{getstatic}}$)**.** *If* $v_1 = C_1\ \{fld : v_f, \overrightarrow{fld_l : v_l}\}$*, then either*

- $\mathsf{JVM}_{\mathsf{getstatic}}[C_1, v_1, fld, C_2] = v_f$*, and either*
  - $v_f = C_2\ \{\overrightarrow{fld_m : v_m}\}$ *or*
  - $v_f = \mathsf{nil}$*, or*
- $\mathsf{JVM}_{\mathsf{getstatic}}[C_1, v_1, fld, C_2] = \mathsf{err}$.

Assumption A.3 ($\mathsf{JVM}_{\mathsf{invokestatic}}$)**.** *If* $v_1 = C_1\ \{\overrightarrow{fld_l : v_l}\}$, $\forall i.\ v_i = C_i\ \{\overrightarrow{fld_j : v_j}\}$ *or* $v_i = \mathsf{nil}$ *then either*

- $\mathsf{JVM}_{\mathsf{invokestatic}}[C_1, v_m, mth, [\overrightarrow{C_i}], [\overrightarrow{v_i}], C_2] = v$ *and either*
  - $v = C_2\ \{\overrightarrow{fld_m : v_m}\}$ *or* $v = \mathsf{nil}$*, or*
- $\mathsf{JVM}_{\mathsf{invokestatic}}[C_1, v_m, mth, [\overrightarrow{C_i}], [\overrightarrow{v_i}], C_2] = \mathsf{err}$*, or*
- $\mathsf{JVM}_{\mathsf{invokestatic}}[C_1, v_m, mth, [\overrightarrow{C_i}], [\overrightarrow{v_i}], C_2]$ *is undefined.*

Lemma A.1. *If* ρ *and* ρ' *agree on* $\mathsf{fv}(\psi)$ *and* $\rho \models \psi$ *then* $\rho' \models \psi$.

Proof. Since the relevant parts of ρ and ρ' agree, the proof follows trivially. □

Lemma A.2. *If*

- $\psi_1 = \psi_2[o/x]$,
- $\rho_2 \models \psi_2$,
- $\forall v \in \mathsf{fv}(\psi_2) - x.\ \rho_1(v) = \rho_2(v)$,
- *and* $\rho_2(x) = \rho_1(o)$

then $\rho_1 \models \psi_1$.

Proof. By induction on the derivation of the model judgement. □

Lemma A.3. *If* $\rho \models \Gamma$ *and* $\Gamma \vdash \psi$ *then* $\rho \models \psi$.

Proof. By structural induction on $\Gamma \vdash \psi$. □

Lemma A.4. *If* $\Gamma \vdash \tau_{\pi(x)}$, $\rho \models \Gamma$ *and* $\rho(\pi(x)) = v$ *then* $\vdash v : \tau\ ;\ {\psi'}_+|{\psi'}_-\ ;\ o'$ *for some* ${\psi'}_+$, ${\psi'}_-$ *and* o'.

Proof. Corollary of lemma A.3. □

Lemma A.5 (Paths are independent)**.** *If* $\rho(o) = \rho_1(o')$ *then* $\rho(\pi(o)) = \rho_1(\pi(o'))$

Proof. By induction on π. □

Lemma A.6 (*class*)**.** *If* $\rho \vdash (\text{class } \rho(\pi(x))) \Downarrow C$ *then* $\rho \models C_{\pi(x)}$.

Proof. Induction on the definition of *class*. □

Definition A.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* $\rho(o') = \rho_1(o')$.

Definition A.2. ρ *is consistent iff*
$\forall v \in rng(\rho)$, v *is consistent with* ρ.

Definition A.3. $\mathsf{TrueVal}(v)$ *iff* $v \neq \mathsf{false}$ *and* $v \neq \mathsf{nil}$.

Definition A.4. $\mathsf{FalseVal}(v)$ *iff* $v = \mathsf{false}$ *or* $v = \mathsf{nil}$.

Lemma A.7 (isa? has correct propositions)**.** *If*

- $\Gamma \vdash v_1 \Rightarrow v_1 : \tau_1\ ;\ {\psi_1}_+|{\psi_1}_-\ ;\ o_1$,
- $\Gamma \vdash v_2 \Rightarrow v_2 : \tau_2\ ;\ {\psi_2}_+|{\psi_2}_-\ ;\ o_2$,
- $\mathsf{IsA}(v_1, v_2) = v$,
- $\rho \models \Gamma$,
- $\mathsf{IsAProps}(o_1, \tau_2) = {\psi'}_+|{\psi'}_-$,
- ${\psi'}_+ \vdash \psi_+$*, and*

- $\psi'_- \vdash \psi_-$,

then either

- *if* $\mathsf{TrueVal}(v)$ *then* $\rho \models \psi_+$, *or*
- *if* $\mathsf{FalseVal}(v)$ *then* $\rho \models \psi_-$.

Proof. By cases on the definition of IsA and subcases on IsA.

Subcase ($\mathsf{IsA}(v_1, v_1) = \mathsf{true}$, if $v_1 \neq C$).
$v_1 = v_2$, $v_1 \neq C$, $v_2 \neq C$, $\mathsf{TrueVal}(v)$
Since $\mathsf{TrueVal}(v)$ we prove $\rho \models \psi_+$ by cases on the definition of $\mathsf{IsAProps}$:

Subcase ($\mathsf{IsAProps}(\mathbf{class}(\pi(x)), (\mathbf{Val}\, C)) = C_{\pi(x)}|\overline{C}_{\pi(x)}$).
$o_1 = \mathbf{class}(\pi(x))$, $\tau_2 = (\mathbf{Val}\, C)$, $C_{\pi(x)} \vdash \psi_+$
Unreachable by inversion on the typing relation, since $\tau_2 = (\mathbf{Val}\, C)$, yet $v_2 \neq C$.

Subcase ($\mathsf{IsAProps}(o, (\mathbf{Val}\, l)) = ((\mathbf{Val}\, l)_x|\overline{(\mathbf{Val}\, l)}_x)[o/x]$ if $l \neq C$).
$\tau_2 = (\mathbf{Val}\, l)$, $l \neq C$, $(\mathbf{Val}\, l)_x[o_1/x] \vdash \psi_+$
Since $\tau_2 = (\mathbf{Val}\, l)$ where $l \neq C$, by inversion on the typing judgement v_2 is either true, false, nil or k by T-True, T-False, T-Nil or T-Kw.
Since $v_1 = v_2$ then $\tau_1 = \tau_2$, and since $\tau_2 = (\mathbf{Val}\, l)$ then $\tau_1 = (\mathbf{Val}\, l)$, so $\vdash v_1 : (\mathbf{Val}\, l)$
If $o_1 = \emptyset$ then $\psi_+ = \mathbb{tt}$ and we derive $\rho \models \mathbb{tt}$ with M-Top.
Otherwise $o_1 = \pi(x)$ and $(\mathbf{Val}\, l)_{\pi(x)} \vdash \psi_+$, and since $\vdash v_1 : (\mathbf{Val}\, l)$ then $\vdash \rho(\pi(x)) : (\mathbf{Val}\, l)$, which we can use M-Type to derive $\rho \models (\mathbf{Val}\, l)_{\pi(x)}$.

Subcase ($\mathsf{IsAProps}(o, \tau) = \mathbb{tt}|\mathbb{tt}$).
$\psi_+ = \mathbb{tt}$
$\rho \models \mathbb{tt}$ holds by M-Top.

Subcase ($\mathsf{IsA}(C_1, C_2) = \mathsf{true}$, if $\vdash C_1 <: C_2$).
$v_1 = C_1$, $v_2 = C_2$, $\vdash C_1 <: C_2$, $\mathsf{TrueVal}(v)$
Since $\mathsf{TrueVal}(v)$ we prove $\rho \models \psi_+$ by cases on the definition of $\mathsf{IsAProps}$:

Subcase ($\mathsf{IsAProps}(\mathbf{class}(\pi(x)), (\mathbf{Val}\, C)) = C_{\pi(x)}|\overline{C}_{\pi(x)}$).
$o_1 = \mathbf{class}(\pi(x))$, $\tau_2 = (\mathbf{Val}\, C_2)$, $C_{2\pi(x)} \vdash \psi_+$
By inversion on the typing relation, since **class** is the last path element of o_1 then $\rho \vdash (\mathit{class}\ \rho(\pi(x))) \Downarrow v_1$.
Since $\rho \vdash (\mathit{class}\ \rho(\pi(x))) \Downarrow C_1$, as $v_1 = C_1$, we can derive from lemma A.6 $\rho \models C_{1\pi(x)}$.
By the induction hypothesis we can derive $\Gamma \vdash C_{1\pi(x)}$, and with the fact $\vdash C_1 <: C_2$ we can use L-Sub to conclude $\Gamma \vdash C_{2\pi(x)}$, and finally by lemma A.3 we derive $\rho \models C_{2\pi(x)}$.

Subcase ($\mathsf{IsAProps}(o, (\mathbf{Val}\, l)) = ((\mathbf{Val}\, l)_x|\overline{(\mathbf{Val}\, l)}_x)[o/x]$ if $l \neq C$).
$\tau_2 = (\mathbf{Val}\, l)$, $l \neq C$, $(\mathbf{Val}\, l)_x[o_1/x] \vdash \psi_+$
Unreachable case since $\tau_2 = (\mathbf{Val}\, l)$ where $l \neq C$, but $v_2 = C_2$.

Subcase ($\mathsf{IsAProps}(o, \tau) = \mathbb{tt}|\mathbb{tt}$).
$\psi_+ = \mathbb{tt}$
$\rho \models \mathbb{tt}$ holds by M-Top.

Subcase ($\mathsf{IsA}(v_1, v_2) = \mathsf{false}$, otherwise).
$v_1 \neq v_2$, $\mathsf{FalseVal}(v)$
Since $\mathsf{FalseVal}(v)$ we prove $\rho \models \psi_-$ by cases on the definition of $\mathsf{IsAProps}$:

Subcase ($\mathsf{IsAProps}(\mathbf{class}(\pi(x)), (\mathbf{Val}\, C)) = C_{\pi(x)}|\overline{C}_{\pi(x)}$).
$o_1 = \mathbf{class}(\pi(x))$, $\tau_2 = (\mathbf{Val}\, C)$, $\overline{C}_{\pi(x)} \vdash \psi_-$
By inversion on the typing relation, since **class** is the last path element of o_1 then $\rho \vdash (\mathit{class}\ \rho(\pi(x))) \Downarrow v_1$.
By the definition of *class* either $v_1 = C$ or $v_1 = \mathsf{nil}$.
If $v_1 = \mathsf{nil}$, then we know from the definition of IsA that $\rho(\pi(x)) = \mathsf{nil}$.
Since $\vdash \rho(\pi(x)) : \mathbf{nil}$, and there is no v_1 such that both $\vdash \rho(\pi(x)) : C$ and $\vdash \rho(\pi(x)) : \mathbf{nil}$, we use M-NotType to derive $\rho \models \overline{C}_{\pi(x)}$.
Similarly if $v_1 = C_1$, by the definition of $\mathsf{IsAProps}$ we know $\vdash C_1 \not<: C$ and $\rho(\pi(x)) = C_1$.
Since $\vdash \rho(\pi(x)) : C_1$, and there is no v_1 such that both $\vdash v_1 : C$ and $\vdash v_1 : C_1$, we use M-NotType to derive $\rho \models \overline{C}_{\pi(x)}$.

Subcase ($\mathsf{IsAProps}(o, (\mathbf{Val}\, l)) = ((\mathbf{Val}\, l)_x | \overline{(\mathbf{Val}\, l)}_x)[o/x]$ if $l \neq C$).
$\tau_2 = (\mathbf{Val}\, l)$, $l \neq C$, $\overline{(\mathbf{Val}\, l)}_x[o_1/x] \vdash \psi_-$
Since $\tau_2 = (\mathbf{Val}\, l)$ where $l \neq C$, by inversion on the typing judgement v_2 is either true, false, nil or k by T-True, T-False, T-Nil or T-Kw.
If $o_1 = \emptyset$ then $\psi_- = \mathtt{tt}$ and we derive $\rho \models \mathtt{tt}$ with M-Top.
Otherwise $o_1 = \pi(x)$ and $\overline{(\mathbf{Val}\, l)}_{\pi(x)} \vdash \psi_-$. Noting that $v_1 \neq v_2$, we know $\vdash \rho(\pi(x)) : \sigma$ where $\sigma \neq (\mathbf{Val}\, l)$, and there is no v_1 such that both $\vdash v_1 : (\mathbf{Val}\, l)$ and $\vdash v_1 : \sigma$ so we can use M-NotType to derive $\rho \models \overline{(\mathbf{Val}\, l)}_{\pi(x)}$.

Subcase ($\mathsf{IsAProps}(o, \tau) = \mathtt{tt}|\mathtt{tt}$).
$\psi_- = \mathtt{tt}$
$\rho \models \mathtt{tt}$ holds by M-Top.

□

Lemma A.8. *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*
- $\rho \vdash e \Downarrow \mathsf{err}$.

Proof. By induction and cases on the derivation of $\rho \vdash e \Downarrow \alpha$, and subcases on the penultimate rule of the derivation of $\Gamma \vdash e' \Rightarrow e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$ followed by T-Subsume as the final rule.

Case (B-Val).

Subcase (T-True). $v = \mathsf{true}$, $e' = \mathsf{true}$, $e = \mathsf{true}$, $\vdash \mathbf{true} <: \tau$, $\mathtt{tt} \vdash \psi_+$, $\mathtt{ff} \vdash \psi_-$, $\vdash \emptyset <: o$
Proving part 1 is trivial: o is a superobject of $\emptyset$, which can only be $\emptyset$.
To prove part 2, we note that $v = \mathsf{true}$ and $\mathtt{tt} \vdash \psi_+$, so $\rho \models \psi_+$ by M-Top.
Part 3 holds as e can only be reduced to itself via B-Val.
Part 4 holds vacuously.

Subcase (T-HMap). $v = \{\overrightarrow{v_k \mapsto v_v}\}$, $e' = \{\overrightarrow{v_k \mapsto v_v}\}$, $e = \{\overrightarrow{v_k \mapsto v_v}\}$, $\vdash (\mathbf{HMap}^{\mathcal{C}}\, \mathcal{M}) <: \tau$, $\mathtt{tt} \vdash \psi_+$, $\mathtt{ff} \vdash \psi_-$, $\vdash \emptyset <: o$, $\overrightarrow{\vdash v_k : (\mathbf{Val}\, k)}$, $\overrightarrow{\vdash v_v : \tau_v}$, $\mathcal{M} = \{\overrightarrow{k \mapsto \tau_v}\}$
Similar to T-True.
Part 4 holds by the induction hypothese on $\overrightarrow{v_k}$ and $\overrightarrow{v_v}$.

Subcase (T-Kw). $v = k$, $e' = k$, $e = k$, $\vdash (\mathbf{Val}\, k) <: \tau$, $\mathtt{tt} \vdash \psi_+$, $\mathtt{ff} \vdash \psi_-$, $\vdash \emptyset <: o$
Similar to T-True.

Subcase (T-Str). Similar to T-Kw.

Subcase (T-False). $v = \mathsf{false}$, $e' = \mathsf{false}$, $e = \mathsf{false}$, $\vdash \mathbf{false} <: \tau$, $\mathtt{ff} \vdash \psi_+$, $\mathtt{tt} \vdash \psi_-$, $\vdash \emptyset <: o$
Proving part 1 is trivial: o is a superobject of $\emptyset$, which must be $\emptyset$.
To prove part 2, we note that $v = \mathsf{false}$ and $\mathtt{tt} \vdash \psi_-$, so $\rho \models \psi_-$ by M-Top.
Part 3 holds as e can only be reduced to itself via B-Val.
Part 4 holds vacuously.

Subcase (T-Class). $v = C$, $e' = C$, $e = C$, $\vdash (\mathbf{Val}\, C) <: \tau$, $\mathtt{tt} \vdash \psi_+$, $\mathtt{ff} \vdash \psi_-$, $\vdash \emptyset <: o$
Similar to T-True.

Subcase (T-Instance). $v = C\ \{\overrightarrow{fld_i : v_i}\}$, $e' = C\ \{\overrightarrow{fld : v}\}$, $e = C\ \{\overrightarrow{fld : v}\}$, $\vdash C <: \tau$, $\mathtt{tt} \vdash \psi_+$, $\mathtt{ff} \vdash \psi_-$, $\vdash \emptyset <: o$
Similar to T-True.
Part 4 holds by the induction hypotheses on $\overrightarrow{v_i}$.

Subcase (T-Nil). $v = \mathsf{nil}$, $e' = \mathsf{nil}$, $e = \mathsf{nil}$, $\vdash \mathbf{nil} <: \tau$, $\mathtt{ff} \vdash \psi_+$, $\mathtt{tt} \vdash \psi_-$, $\vdash \emptyset <: o$
Similar to T-False.

Subcase (T-Multi). $v = [v_1, \{\overrightarrow{v_k \mapsto v_v}\}]_{\mathsf{m}}$ $e' = [v_1, \{\overrightarrow{v_k \mapsto v_v}\}]_{\mathsf{m}}$, $\vdash v_1 \Rightarrow v_1 : \tau_1$, $\overrightarrow{\vdash v_k \Rightarrow v_k : \top}$, $\overrightarrow{\vdash v_v \Rightarrow v_v : \sigma}$, $e = [v_1, \{\overrightarrow{v_k \mapsto v_v}\}]_{\mathsf{m}}$, $\vdash (\mathbf{Multi}\, \sigma\ \tau_1) <: \tau$, $\mathtt{tt} \vdash \psi_+$, $\mathtt{ff} \vdash \psi_-$, $\vdash \emptyset <: o$
Similar to T-True.

Subcase (T-Const). $e = c, \vdash \delta_\tau(c) <: \tau, \mathtt{tt} \vdash \psi_+, \mathtt{ff} \vdash \psi_-, \vdash \emptyset <: o$
Parts 1, 2 and 3 hold for the same reasons as T-True.

Case (B-Local). $\rho(x) = v\ ,\ \rho \vdash x \Downarrow v$

Subcase (T-Local). $e' = x, e = x, \overline{(\cup\ \textbf{nil false})}_x \vdash \psi_+, (\cup\ \textbf{nil false})_x \vdash \psi_-, \vdash x <: o, \Gamma \vdash \tau_x$
Part 1 follows from $\rho(o) = v$, since either $o = x$ and $\rho(x) = v$ is a premise of B-Local, or $o = \emptyset$ which also satisfies the goal.
Part 2 considers two cases: if $\mathsf{TrueVal}(v)$, then $\rho \models \overline{(\cup\ \textbf{nil false})}_x$ holds by M-NotType; if $\mathsf{FalseVal}(v)$, then $\rho \models (\cup\ \textbf{nil false})_x$ holds by M-Type.
We prove part 3 by observing $\Gamma \vdash \tau_x$, $\rho \models \Gamma$, and $\rho(x) = v$ (by B-Local) which gives us the desired result.
Part 4 holds vacuously.

Case (B-Do). $\rho \vdash e_1 \Downarrow v_1, \rho \vdash e_2 \Downarrow v$

Subcase (T-Do). $e' = (\mathrm{do}\ e'_1\ e'_2), \Gamma \vdash e'_1 \Rightarrow e_1 : \tau_1\ ;\ \psi_{1+}|\psi_{1-}\ ;\ o_1, \Gamma, \psi_{1+} \vee \psi_{1-} \vdash e' \Rightarrow e : \tau\ ;\ \psi_+|\psi_-\ ;\ o, e = (\mathrm{do}\ e_1\ e_2)$
For all parts we note since e_1 can be either a true or false value then $\rho \models \Gamma, \psi_{1+} \vee \psi_{1-}$ by M-Or, which together with $\Gamma, \psi_{1+} \vee \psi_{1-} \vdash e_2 : \tau\ ;\ \psi_+|\psi_-\ ;\ o$, and $\rho \vdash e_2 \Downarrow v$ allows us to apply the induction hypothesis on e_2.
To prove part 1 we use the induction hypothesis on e_2 to show either $o = \emptyset$ or $\rho(o) = v$, since e always evaluates to the result of e_2.
For part 2 we use the induction hypothesis on e_2 to show if $\mathsf{TrueVal}(v)$ then $\rho \models \psi_+$ or if $\mathsf{FalseVal}(v)$ then $\rho \models \psi_-$.
Parts 3 and 4 follow from the induction hypothesis on e_2.

Case (BE-Do1). $\rho \vdash e_1 \Downarrow \mathsf{err}, \rho \vdash e \Downarrow \mathsf{err}$
Trivially reduces to an error.

Case (BE-Do2). $\rho \vdash e_1 \Downarrow v_1, \rho \vdash e_2 \Downarrow \mathsf{err}, \rho \vdash e \Downarrow \mathsf{err}$
As above.

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

Subcase (T-New). $e' = (\mathrm{new}\ C\overrightarrow{e_i}), [\overrightarrow{C_i}] \in \mathcal{CT}[C][\mathsf{c}], \overrightarrow{\mathsf{JT}_{\mathsf{nil}}(C_i)\ =\ \tau_i}, \overrightarrow{\Gamma \vdash e'_i \Rightarrow e_i : \tau_i}, e = (\mathrm{new}_{[\overrightarrow{C_i}]}\ C\ \overrightarrow{e_i}), \vdash \mathsf{JT}(C) <: \tau, \mathtt{tt} \vdash \psi_+, \mathtt{ff} \vdash \psi_-, \vdash \emptyset <: o$
Part 1 follows $o = \emptyset$.
Part 2 requires some explanation. The two false values in Typed Clojure cannot be constructed with new, so the only case is $v \neq \mathsf{false}$ (or nil) where $\psi_+ = \mathtt{tt}$ so $\rho \models \psi_+$. **Void** also lacks a constructor.
Part 3 holds as B-New reduces to a *non-nilable* instance of C via $\mathsf{JVM}_{\mathsf{new}}$ (by assumption A.1), and τ is a supertype of $\mathsf{JT}\ (C)$.

Subcase (T-NewStatic). $e' = (\mathrm{new}_{[\overrightarrow{C_i}]}\ C\ \overrightarrow{e_i})$
Non-reflective constructors cannot be written directly by the user, so we can assume the class information attached to the syntax corresponds to an actual constructor by inversion from T-New.
The rest of this case progresses like T-New.

Case (BE-New1). $\overrightarrow{\rho \vdash e_{i-1} \Downarrow v_{i-1}}, \rho \vdash e_i \Downarrow \mathsf{err}, \rho \vdash e \Downarrow \mathsf{err}$
Trivially reduces to an error.

Case (BE-New2). $\overrightarrow{\rho \vdash e_i \Downarrow v_i}, \mathsf{JVM}_{\mathsf{new}}[C_1, [\overrightarrow{C_i}], [\overrightarrow{v_i}]] = \mathsf{err}, \rho \vdash e \Downarrow \mathsf{err}$
As above.

Case (B-Field). $\rho \vdash e_1 \Downarrow C_1\ \{fld : v\}$

Subcase (T-Field). $e' = (.\ e'_1\ fld), \Gamma \vdash e' \Rightarrow e : \sigma, \vdash \sigma <: \textbf{Object}, \mathsf{TJ}\ (\sigma) = C_1, fld \mapsto C_2 \in \mathcal{CT}[C_1][\mathsf{f}], e = (.\ e_1\ fld^{C_1}_{C_2}) \vdash \mathsf{JT}_{\mathsf{nil}}(C_2) <: \tau, \mathtt{tt} \vdash \psi_+, \mathtt{tt} \vdash \psi_-, \vdash \emptyset <: o$
Part 1 is trivial as o is always $\emptyset$.
Part 2 holds trivially; v can be either a true or false value and both ψ_+ and ψ_- are $\mathtt{tt}$.
Part 3 relies on the semantics of $\mathsf{JVM}_{\mathsf{getstatic}}$ (assumption A.2) in B-Field, which returns a *nilable* instance of C_2, and τ is a supertype of $\mathsf{JT}_{\mathsf{nil}}(C_2)$. Notice $\vdash \sigma <: \textbf{Object}$ is required to guard from dereferencing nil, as C_1 erases occurrences of **nil** in σ via $\mathsf{TJ}\ (\sigma) = C_1$.

Subcase (T-FieldStatic). $e' = (.\ e_1\ fld^{C_1}_{C_2})$
Non-reflective field lookups cannot be written directly by the user, so we can assume the class information attached to the syntax corresponds to an actual field by inversion from T-Field.
The rest of this case progresses like T-Field.

Case (BE-Field). $\rho \vdash e_1 \Downarrow \mathsf{err}, \rho \vdash e \Downarrow \mathsf{err}$
Trivially reduces to an error.

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

Subcase (T-Method). $\Gamma \vdash e' \Rightarrow e : \sigma, \vdash \sigma <: \mathbf{Object}, \mathsf{TJ}(\sigma) = C_1, mth \mapsto [[\overrightarrow{C_i}], C_2] \in \mathcal{CT}[C_1][\mathsf{m}], \overrightarrow{\mathsf{JT}_{\mathsf{nil}}(C_i) = \tau_i}, \overrightarrow{\Gamma \vdash e'_i \Rightarrow e_i : \tau_i}$, $e = (.\ e_m\ (mth^{C_1}_{[[\overrightarrow{C_i}], C_2]}\ \overrightarrow{e_a}))$, $\vdash \mathsf{JT}_{\mathsf{nil}}(C_2) <: \tau, \mathbb{tt} \vdash \psi_+, \mathbb{tt} \vdash \psi_-, \vdash \emptyset <: o$
Part 1 is trivial as o is always $\emptyset$.
Part 2 holds trivially, v can be either a true or false value and both ψ_+ and ψ_- are $\mathbb{tt}$.
Part 3 relies on the semantics of $\mathsf{JVM}_{\mathsf{invokestatic}}$ (assumption A.3) in B-Method, which returns a *nilable* instance of C_2, and τ is a supertype of $\mathsf{JT}_{\mathsf{nil}}(C_2) = .$ Notice $\vdash \sigma <: \mathbf{Object}$ is required to guard from dereferencing nil, as C_1 erases occurrences of **nil** in σ via $\mathsf{TJ}(\sigma) = C_1$.

Subcase (T-MethodStatic). $e' = (.\ e_1\ (mth^{C_1}_{[[\overrightarrow{C_i}], C_2]}\ \overrightarrow{e_i}))$
Non-reflective method invocations cannot be written directly by the user, so we can assume the class information attached to the syntax corresponds to an actual method by inversion from T-Method.
The rest of this case progresses like T-Method.

Case (BE-Method1). $\rho \vdash e_m \Downarrow \mathsf{err}, \rho \vdash e \Downarrow \mathsf{err}$
Trivially reduces to an error.

Case (BE-Method2). $\rho \vdash e_m \Downarrow v_m, \overrightarrow{\rho \vdash e_{n-1} \Downarrow v_{n-1}}, \rho \vdash e_n \Downarrow \mathsf{err}, \rho \vdash e \Downarrow \mathsf{err}$
As above.

Case (BE-Method3). $\rho \vdash e_m \Downarrow v_m, \overrightarrow{\rho \vdash e_a \Downarrow v_a}, \mathsf{JVM}_{\mathsf{invokestatic}}[C_1, v_m, mth, [\overrightarrow{C_a}], [\overrightarrow{v_a}], C_2] = \mathsf{err}, \rho \vdash e \Downarrow \mathsf{err}$
As above.

Case (B-DefMulti). $v = [v_d, \{\}]_{\mathsf{m}}, \rho \vdash e_d \Downarrow v_d$

Subcase (T-DefMulti). $e' = (\text{defmulti}\ \sigma\ e'_d), \sigma = x{:}\tau_1 \xrightarrow[o_1]{\psi_{1+}|\psi_{1-}} \tau_2, \tau_d = x{:}\tau_1 \xrightarrow[o_2]{\psi_{2+}|\psi_{2-}} \tau_3, \Gamma \vdash e' \Rightarrow e : \sigma', e = (\text{defmulti}\ \sigma\ e_d)$, $\vdash (\mathbf{Multi}\, \sigma\ \tau_d) <: \tau, \mathbb{tt} \vdash \psi_+, \mathbb{ff} \vdash \psi_-, \vdash \emptyset <: o$
Part 1 and 2 hold for the same reasons as T-True. For part 3 we show $\vdash [v_d, \{\}]_{\mathsf{m}} : (\mathbf{Multi}\, \sigma\ \tau_d)$ by T-Multi, since $\vdash v_d : \tau_d$ by the inductive hypothesis on e_d and $\{\}$ vacuously satisfies the other premises of T-Multi, so we are done.

Case (BE-DefMulti). $\rho \vdash e_d \Downarrow \mathsf{err}, \rho \vdash e \Downarrow \mathsf{err}$
Trivially reduces to an error.

Case (B-DefMethod).

1. $v = [v_d, t']_{\mathsf{m}}$,
2. $\rho \vdash e_m \Downarrow [v_d, t]_{\mathsf{m}}$,
3. $\rho \vdash e_v \Downarrow v_v$,
4. $\rho \vdash e_f \Downarrow v_f$,
5. $t' = t[v_v \mapsto v_f]$

Subcase (T-DefMethod).

6. $e' = (\text{defmethod}\ e'_m\ e'_v\ e'_f)$,
7. $\tau_m = x{:}\tau_1 \xrightarrow[o_m]{\psi_{m+}|\psi_{m-}} \sigma$,
8. $\tau_d = x{:}\tau_1 \xrightarrow[o_d]{\psi_{d+}|\psi_{d-}} \sigma'$,
9. $\Gamma \vdash e'_m \Rightarrow e_m : (\mathbf{Multi}\, \tau_m\ \tau_d)$
10. $\mathsf{IsAProps}(o_d, \tau_v) = \psi_{i+}|\psi_{i-}$,
11. $\Gamma \vdash e_v \Rightarrow e_v : \tau_v$
12. $\Gamma, \tau_{1x}, \psi_{i+} \vdash e'_f \Rightarrow e_f : \sigma\ ;\ \psi_{m+}|\psi_{m-}\ ;\ o_m$
13. $e = (\text{defmethod}\ e_m\ e_v\ e_f)$,
14. $e_f = \lambda x^{\tau_1}.e_b$,
15. $\vdash (\mathbf{Multi}\, \tau_m\ \tau_d) <: \tau$,
16. $\mathbb{tt} \vdash \psi_+$,
17. $\mathbb{ff} \vdash \psi_-$,
18. $\vdash \emptyset <: o$

Part 1 and 2 hold for the same reasons as T-True, noting that the propositions and object agree with T-Multi.
For part 3 we show $\vdash [v_d, t[v_v \mapsto v_f]]_{\mathsf{m}} : (\textbf{Multi}\ \tau_m\ \tau_d)$ by noting $\vdash v_d : \tau_d$, $\vdash v_v : \top$ and $\vdash v_f : \tau_m$, and since t is in the correct form by the inductive hypothesis on e_m we can satisfy all premises of T-Multi, so we are done.

Case (BE-DefMethod1). $\rho \vdash e_m \Downarrow \mathsf{err}$, $\rho \vdash e \Downarrow \mathsf{err}$
Trivially reduces to an error.

Case (BE-DefMethod2). $\rho \vdash e_m \Downarrow [v_d, t]_{\mathsf{m}}$, $\rho \vdash e_v \Downarrow \mathsf{err}$, $\rho \vdash e \Downarrow \mathsf{err}$
Trivially reduces to an error.

Case (BE-DefMethod3). $\rho \vdash e_m \Downarrow [v_d, t]_{\mathsf{m}}$, $\rho \vdash e_v \Downarrow v_v$, $\rho \vdash e_f \Downarrow \mathsf{err}$, $\rho \vdash e \Downarrow \mathsf{err}$
Trivially reduces to an error.

Case (B-BetaClosure).

- $\rho \vdash e \Downarrow v$,
- $\rho \vdash e_1 \Downarrow [\rho_c, \lambda x^{\sigma}.e_b]_{\mathsf{c}}$,
- $\rho \vdash e_2 \Downarrow v_2$,
- $\rho_c[x \mapsto v_2] \vdash e_b \Downarrow v$

Subcase (T-App).

- $e' = (e_1'\ e_2')$,
- $\Gamma \vdash e_1' \Rightarrow e_1 : x{:}\sigma \xrightarrow[o_f]{\psi_{f_+}|\psi_{f_-}} \tau_f\ ;\ \psi_{1_+}|\psi_{1_-}\ ;\ o_1$,
- $\Gamma \vdash e_2' \Rightarrow e_2 : \sigma\ ;\ \psi_{2_+}|\psi_{2_-}\ ;\ o_2$,
- $e = (e_1\ e_2)$,
- $\vdash \tau_f[o_2/x] <: \tau$,
- $\psi_{f_+}[o_2/x] \vdash \psi_+$,
- $\psi_{f_-}[o_2/x] \vdash \psi_-$,
- $\vdash o_f[o_2/x] <: o$

By inversion on e_1 from T-Clos there is some environment Γ_c such that

- $\rho_c \models \Gamma_c$ and
- $\Gamma_c \vdash \lambda x^{\sigma}.e_b : x{:}\sigma \xrightarrow[o_f]{\psi_{f_+}|\psi_{f_-}} \tau_f\ ;\ \psi_{1_+}|\psi_{1_-}\ ;\ o_1$,

and also by inversion on e_1 from T-Abs

- $\Gamma_c, \sigma_x \vdash e_b' \Rightarrow e_b : \tau_f\ ;\ \psi_{f_+}|\psi_{f_-}\ ;\ o_f$.

From

- $\rho_c \models \Gamma_c$,
- $\Gamma \vdash e_2' \Rightarrow e_2 : \sigma\ ;\ \psi_{2_+}|\psi_{2_-}\ ;\ o_2$ and
- $\rho \vdash e_2 \Downarrow v_2$,

we know (by substitution) $\rho_c[x \mapsto v_2] \models \Gamma_c, \sigma_x$.
We want to prove $\Gamma_c \vdash e_b'[v_2/x] \Rightarrow e_b[v_2/x] : \tau_f[o_2/x]\ ;\ \psi_{f_+}|\psi_{f_-}[o_2/x]\ ;\ o_f[o_2/x]$, which can be justified by noting

- $\Gamma_c, \sigma_x \vdash e_b' \Rightarrow e_b : \tau_f$,
- $\Gamma \vdash e_2' \Rightarrow e_2 : \sigma\ ;\ \psi_{2_+}|\psi_{2_-}\ ;\ o_2$ and
- $\rho \vdash e_2 \Downarrow v_2$.

From the previous fact and $\rho_c \models \Gamma_c$, we know $\rho_c \vdash e_b[v_2/x] \Downarrow v$.
Noting that $\vdash \tau_f[o_2/x] <: \tau$, $\psi_{f_+}[o_2/x] \vdash \psi_+$, $\psi_{f_-}[o_2/x] \vdash \psi_-$ and $\vdash o_f[o_2/x] <: o$, we can use

- $\Gamma_c \vdash e_b'[v_2/x] \Rightarrow e_b[v_2/x] : \tau_f[o_2/x]\ ;\ \psi_{f_+}|\psi_{f_-}[o_2/x]\ ;\ o_f[o_2/x]$,
- $\rho_c \models \Gamma_c$,
- ρ_c is consistent (via induction hypothesis on e_1'), and
- $\rho_c \vdash e_b[v_2/x] \Downarrow v$.

to apply the induction hypothesis on $e_b'[v_2/x]$ and satisfy all conditions.

Case (B-Delta). $\rho \vdash e_1 \Downarrow c$, $\rho \vdash e_2 \Downarrow v_2$, $\delta(c, v_2) = v$

Subcase (T-App).

- $e' = (e_1'\ e_2')$,
- $\Gamma \vdash e_1' \Rightarrow e_1 : x{:}\sigma \xrightarrow[o_f]{\psi_{f_+}|\psi_{f_-}} \tau_f\ ;\ \psi_{1_+}|\psi_{1_-}\ ;\ o_1$,
- $\Gamma \vdash e_2' \Rightarrow e_2 : \sigma\ ;\ \psi_{2_+}|\psi_{2_-}\ ;\ o_2$,

- $e = (e_1\ e_2)$,
- $\vdash \tau_f[o_2/x] <: \tau$,
- $\psi_{f+}[o_2/x] \vdash \psi_+$,
- $\psi_{f-}[o_2/x] \vdash \psi_-$,
- $\vdash o_f[o_2/x] <: o$

Prove by cases on c.

Subcase (c = *class*). $\vdash x\colon \top \xrightarrow[\mathbf{class}(x)]{\mathtt{tt}\mid\mathtt{tt}} (\bigcup\ \mathbf{nil}\ \mathbf{Class}) <: x{:}\sigma \xrightarrow[o_f]{\psi_{f+}\mid\psi_{f-}} \tau_f$

Prove by cases on v_2.

Subcase ($v_2 = C\ \{\overrightarrow{fld_i : v_i}\}$). $v = C$
To prove part 1, note $\vdash o_f[o_2/x] <: o$, and $\vdash \mathbf{class}(x) <: o_f$. Then either $o = \emptyset$ and we are done, or $o = \mathbf{class}(o_2)$ and by the induction hypothesis on e_2 we know $\rho(o_2) = v_2$ and by the definition of path translation we know $\rho(\mathbf{class}(o_2)) = (\mathit{class}\ \rho(o_2))$, which evaluates to v.
Part 2 is trivial since both propositions can only be $\mathtt{tt}$.
Part 3 holds because $v = C, \vdash (\bigcup\ \mathbf{nil}\ \mathbf{Class}) <: \tau_f[o_2/x]$ and $\vdash \tau_f[o_2/x] <: \tau$, so $\vdash v : \tau$ since $\vdash C : (\bigcup\ \mathbf{nil}\ \mathbf{Class})$.

Subcase ($v_2 = C$). $v = \mathbf{Class}$
As above.

Subcase ($v_2 = \mathsf{true}$). $v = \mathbf{B}$
As above.

Subcase ($v_2 = \mathsf{false}$). $v = \mathbf{B}$
As above.

Subcase ($v_2 = [\rho, \lambda x^{\tau}.e]_{\mathsf{c}}$). $v = \mathbf{Fn}$
As above.

Subcase ($v_2 = [v_d, t]_{\mathsf{m}}$). $v = \mathbf{Map}$
As above.

Subcase ($v_2 = \{\overrightarrow{v_1 \mapsto v_2}\}$). $v = \mathbf{K}$
As above.

Subcase ($v_2 = \mathsf{nil}$). $v = \mathsf{nil}$
Parts 1 and 2 as above. Part 3 holds because $v = \mathsf{nil}$ and $\vdash \mathsf{nil} : (\bigcup\ \mathbf{nil}\ \mathbf{Class})$.

Case (B-BetaMulti).

- $\rho \vdash e_1 \Downarrow [v_d, t]_{\mathsf{m}}$,
- $\rho \vdash e_2 \Downarrow v_2$,
- $\rho \vdash (v_d\ v_2) \Downarrow v_e$,
- $\mathsf{GM}\ (t, v_e) = v_g$,
- $\rho \vdash (v_g\ v_2) \Downarrow v$,
- $t = \{\overrightarrow{v_k \mapsto v_v}\}$

Subcase (T-App).
- $e' = (e'_1\ e'_2)$,
- $\Gamma \vdash e'_1 \Rightarrow e_1 : x{:}\sigma \xrightarrow[o_f]{\psi_{f+}\mid\psi_{f-}} \tau_f\ ;\ \psi_{1+}\mid\psi_{1-}\ ;\ o_1$,
- $\Gamma \vdash e'_2 \Rightarrow e_2 : \sigma\ ;\ \psi_{2+}\mid\psi_{2-}\ ;\ o_2$,
- $e = (e_1\ e_2)$,
- $\vdash \tau_f[o_2/x] <: \tau$,
- $\psi_{f+}[o_2/x] \vdash \psi_+$,
- $\psi_{f-}[o_2/x] \vdash \psi_-$,
- $\vdash o_f[o_2/x] <: o$,

By inversion on e_1 via T-Multi we know
- $\Gamma \vdash e'_1 \Rightarrow e_1 : (\mathbf{Multi}\ \sigma_t\ \sigma_d)\ ;\ \psi_{1+}\mid\psi_{1-}\ ;\ o_1$,

- $\sigma_t = x{:}\sigma \xrightarrow[o_f]{\psi_{f+}|\psi_{f-}} \tau_f$,
- $\sigma_d = x{:}\sigma \xrightarrow[o_d]{\psi_{d+}|\psi_{d-}} \tau_d$,
- $\vdash v_d : \sigma_d$
- $\vdash v_k : \top$, and
- $\vdash v_v : \sigma_t$.

By the inductive hypothesis on $\rho \vdash e_2 \Downarrow v_2$ we know $\Gamma \vdash v_2 \Rightarrow v_2 : \sigma \; ; \; \psi_{2+}|\psi_{2-} \; ; \; o_2$.
We then consider applying the evaluated argument to the dispatch function: $\rho \vdash (v_d \; v_2) \Downarrow v_e$.
Since we can satisfy T-App with

- $\vdash v_d : x{:}\sigma \xrightarrow[o_d]{\psi_{d+}|\psi_{d-}} \tau_d$, and
- $\Gamma \vdash v_2 \Rightarrow v_2 : \sigma \; ; \; \psi_{2+}|\psi_{2-} \; ; \; o_2$.

we can then apply the inductive hypothesis to derive $\Gamma \vdash v_e \Rightarrow v_e : \tau_d[o_2/x] \; ; \; \psi_{d+}|\psi_{d-}[o_2/x] \; ; \; o_d[o_2/x]$.
Now we consider how we choose which method to dispatch to.
As $\mathsf{GM}\,(t, v_e) = v_g$, by inversion on GM we know there exists exactly one v_k such that $v_k \mapsto v_g \in t$ and $\mathsf{IsA}(v_e, v_k) = \mathsf{true}$.
By inversion we know T-DefMethod must have extended t with the well-typed dispatch value v_k, thus $\vdash v_k : \tau_k$, and the well-typed method v_g, so $\vdash v_g : \sigma_t$.
We can also prove that given

- $\Gamma \vdash v_e \Rightarrow v_e : \tau_d[o_2/x] \; ; \; \psi_{d+}|\psi_{d-}[o_2/x] \; ; \; o_d[o_2/x]$.
- $\Gamma \vdash v_k : \tau_k$,
- $\mathsf{IsA}(v_e, v_k) = \mathsf{true}$,
- $\rho \models \Gamma$,
- $\mathsf{IsAProps}(o_d[o_2/x], \tau_k) = \psi'_+|\psi'_-$,
- $\psi'_+ \vdash \psi'_+$, and
- $\psi'_- \vdash \psi'_-$.

we can apply Lemma A.7 to derive then $\rho \models \psi'_+$.
Now we consider applying the evaluated argument to the chosen method: $\rho \vdash (v_g \; v_2) \Downarrow v$.
By inversion via B-DefMethod we can assume $v_g = \lambda x^\sigma . e_b$, ie. that we have chosen a method to dispatch to that is a closure.
Because $\rho \vdash (v_g \; v_2) \Downarrow v$ and $\Gamma \vdash v_2 : \sigma$, by inversion via B-BetaClosure we know $v = e_b[v_2/x]$.
With the following premises:

- $\Gamma, \psi'_+ \vdash e'_b[v_2/x] \Rightarrow e_b[v_2/x] : \tau_f[o_2/x] \; ; \; \psi_{f+}|\psi_{f-}[o_2/x] \; ; \; o_f[o_2/x]$,
  - From $\Gamma, \sigma_x \vdash e_b \Rightarrow e_b : \tau_f \; ; \; \psi_{f+}|\psi_{f-} \; ; \; o_f$ via the inductive hypothesis on $\rho \vdash (\lambda x^\sigma . e_b \; v_2) \Downarrow v$,
  - then we can derive $\Gamma \vdash e'_b[v_2/x] \Rightarrow e_b[v_2/x] : \tau_f[o_2/x] \; ; \; \psi_{f+}|\psi_{f-}[o_2/x] \; ; \; o_f[o_2/x]$ via substitution and the fact that x is fresh therefore $x \notin \mathsf{fv}(\Gamma)$ so we do not need to substitution for x in Γ.
  - $\rho \models \Gamma, \psi'_+$ because $\rho \models \Gamma$ and $\rho \models \psi'_+$ via M-And.
- $\rho \models \Gamma, \psi'_+$,
  - From $\rho \models \Gamma$ and
  - $\rho \models \psi'_+$ via M-And.
- ρ is consistent, and
- $\rho \vdash e_b[v_2/x] \Downarrow v$.

we can apply the inductive hypothesis to satisfy our overall goal for this subcase.

Case (BE-Beta1).
Reduces to an error.

Case (BE-Beta2).
Reduces to an error.

Case (BE-BetaClosure).
Reduces to an error.

Case (BE-BetaMulti1).
Reduces to an error.

Case (BE-BetaMulti2).
Reduces to an error.

Case (BE-Delta).
Reduces to an error.

Case (B-IsA). $\rho \vdash e_1 \Downarrow v_1$, $\rho \vdash e_2 \Downarrow v_2$, $\mathsf{IsA}(v_1, v_2) = v$

Subcase (T-IsA). e' = (isa? e'_1 e'_2), $\Gamma \vdash e'_1 \Rightarrow e_1 : \tau$; $\psi_{1+}|\psi_{1-}$; o_1, $\Gamma \vdash e'_2 \Rightarrow e_2 : \tau_2$; $\psi_{2+}|\psi_{2-}$; o_2, e = (isa? e_1 e_2), $\vdash \mathbf{B} <: \tau$, $\mathsf{IsAProps}(o_1, \tau_2) = \psi'_+|\psi'_-, \psi'_+ \vdash \psi_+, \psi'_- \vdash \psi_-, \vdash \emptyset <: o$
Part 1 holds trivially with $o = \emptyset$.
For part 2, by the induction hypothesis on e_1 and e_2 we know $\Gamma \vdash v_1 \Rightarrow v_1 : \tau_1$; $\psi_{1+}|\psi_{1-}$; o_1 and $\Gamma \vdash v_2 \Rightarrow v_2 : \tau_2$; $\psi_{2+}|\psi_{2-}$; o_2, so we can then apply Lemma A.7 to reach our goal.
Part 3 holds because by the definition of IsA v can only be true or false, and since $\Gamma \vdash \mathsf{true} : \tau$ and $\Gamma \vdash \mathsf{false} : \tau$ we are done.

Case (BE-IsA1). $\rho \vdash e_1 \Downarrow \mathsf{err}$
Trivially reduces to an error.

Case (BE-IsA2). $\rho \vdash e_1 \Downarrow v_1, \rho \vdash e_2 \Downarrow \mathsf{err}$
Trivially reduces to an error.

Case (B-Get). $\rho \vdash e_m \Downarrow v_m, v_m = \{\overrightarrow{(v_a\ v_b)}\}, \rho \vdash e_k \Downarrow k, k \in dom(\{\overrightarrow{(v_a\ v_b)}\}), \{\overrightarrow{(v_a\ v_b)}\}[k] = v$

Subcase (T-GetHMap). e' = (get e'_m e'_k), $\Gamma \vdash e'_m \Rightarrow e_m : (\bigcup \overrightarrow{(\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})})$; $\psi_{m+}|\psi_{m-}$; o_m, $\Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val}\, k)$, $\overrightarrow{\mathcal{M}[k] = \tau_i}$, e = (get e_m e_k), $\vdash (\bigcup \overrightarrow{\tau_i}) <: \tau$, $\psi_+ = \mathbb{tt}$, $\psi_- = \mathbb{tt}$, $\vdash \mathbf{key}_k(x)[o_m/x] <: o$
To prove part 1 we consider two cases on the form of o_m:
- if $o_m = \emptyset$ then $o = \emptyset$ by substitution, which gives the desired result;
- if $o_m = \pi_m(x_m)$ then $\vdash \mathbf{key}_k(o_m) <: o$ by substitution. We note by the definition of path translation $\rho(\mathbf{key}_k(o_m)) = (\mathrm{get}\ \rho(o_m)\ k)$ and by the induction hypothesis on e_m $\rho(o_m) = \{\overrightarrow{(v_a\ v_b)}\}$, which together imply $\rho(o) = (\mathrm{get}\ \{\overrightarrow{(v_a\ v_b)}\}\ k)$. Since this is the same form as B-Get, we can apply the premise $\{\overrightarrow{(v_a\ v_b)}\}[k] = v$ to derive $\rho(o) = v$.

Part 2 holds trivially as $\psi_+ = \mathbb{tt}$ and $\psi_- = \mathbb{tt}$.
To prove part 3 we note that (by the induction hypothesis on e_m) $\vdash v_m : (\bigcup \overrightarrow{(\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})})$, where $\overrightarrow{\mathcal{M}[k] = \tau_i}$, and both $k \in dom(\{\overrightarrow{(v_a\ v_b)}\})$ and $\{\overrightarrow{(v_a\ v_b)}\}[k] = v$ imply $\vdash v : (\bigcup \overrightarrow{\tau_i})$.

Subcase (T-GetHMapAbsent). e' = (get e'_m e'_k), $\Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val}\, k)$, $\Gamma \vdash e'_m \Rightarrow e_m : (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})$; $\psi_{m+}|\psi_{m-}$; o_m, $k \in \mathcal{A}$, e = (get e_m e_k), $\vdash \mathbf{nil} <: \tau$, $\psi_+ = \mathbb{tt}$, $\psi_- = \mathbb{tt}$, $\vdash \mathbf{key}_k(x)[o_m/x] <: o$
Unreachable subcase because $k \in dom(\{\overrightarrow{(v_a\ v_b)}\})$, contradicts $k \in \mathcal{A}$.

Subcase (T-GetHMapPartialDefault). e' = (get e'_m e'_k), $\Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val}\, k)$, $\Gamma \vdash e'_m \Rightarrow e_m : (\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}\ \mathcal{A})$; $\psi_{m+}|\psi_{m-}$; o_m, $k \notin dom(\mathcal{M})$, $k \notin \mathcal{A}$, e = (get e_m e_k), $\tau = \top$, $\psi_+ = \mathbb{tt}$, $\psi_- = \mathbb{tt}$, $\vdash \mathbf{key}_k(x)[o_m/x] <: o$
Parts 1 and 2 are the same as the B-Get subcase. Part 3 is trivial as $\tau = \top$.

Case (B-GetMissing). $v = \mathsf{nil}$, $\rho \vdash e_m \Downarrow \{\overrightarrow{(v_a\ v_b)}\}, \rho \vdash e_k \Downarrow k, k \notin dom(\{\overrightarrow{(v_a\ v_b)}\})$

Subcase (T-GetHMap). e' = (get e'_m e'_k), $\Gamma \vdash e'_m \Rightarrow e_m : (\bigcup \overrightarrow{(\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})})$; $\psi_{m+}|\psi_{m-}$; o_m, $\Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val}\, k)$, $\overrightarrow{\mathcal{M}[k] = \tau_i}$, e = (get e_m e_k), $\vdash (\bigcup \overrightarrow{\tau_i}) <: \tau$, $\psi_+ = \mathbb{tt}$, $\psi_- = \mathbb{tt}$, $\vdash \mathbf{key}_k(x)[o_m/x] <: o$
Unreachable subcase because $k \notin dom(\{\overrightarrow{(v_a\ v_b)}\})$ contradicts $\mathcal{M}[k] = \tau$.

Subcase (T-GetHMapAbsent). e' = (get e'_m e'_k), $\Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val}\, k)$, $\Gamma \vdash e'_m \Rightarrow e_m : (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})$; $\psi_{m+}|\psi_{m-}$; o_m, $k \in \mathcal{A}$, e = (get e_m e_k), $\vdash \mathbf{nil} <: \tau$, $\psi_+ = \mathbb{tt}$, $\psi_- = \mathbb{tt}$, $\vdash \mathbf{key}_k(x)[o_m/x] <: o$
To prove part 1 we consider two cases on the form of o_m:
- if $o_m = \emptyset$ then $o = \emptyset$ by substitution, which gives the desired result;
- if $o_m = \pi_m(x_m)$ then $\vdash \mathbf{key}_k(o_m) <: o$ by substitution. We note by the definition of path translation $\rho(\mathbf{key}_k(o_m)) = (\mathrm{get}\ \rho(o_m)\ k)$ and by the induction hypothesis on e_m $\rho(o_m) = \{\overrightarrow{(v_a\ v_b)}\}$, which together imply $\rho(o) = (\mathrm{get}\ \{\overrightarrow{(v_a\ v_b)}\}\ k)$. Since this is the same form as B-GetMissing, we can apply the premise $v = \mathsf{nil}$ to derive $\rho(o) = v$.

Part 2 holds trivially as $\psi_+ = \mathbb{tt}$ and $\psi_- = \mathbb{tt}$.
To prove part 3 we note that e_m has type $(\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})$ where $k \in \mathcal{A}$, and the premises of B-GetMissing $k \notin dom(\{\overrightarrow{(v_a\ v_b)}\})$ and $v = \mathsf{nil}$ tell us v must be of type τ.

Subcase (T-GetHMapPartialDefault). e' = (get e'_m e'_k), $\Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val}\, k)$, $\Gamma \vdash e'_m \Rightarrow e_m : (\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}\ \mathcal{A})$; $\psi_{m+}|\psi_{m-}$; o_m, $k \notin dom(\mathcal{M})$, $k \notin \mathcal{A}$, e = (get e_m e_k), $\tau = \top$, $\psi_+ = \mathbb{tt}$, $\psi_- = \mathbb{tt}$, $\vdash \mathbf{key}_k(x)[o_m/x] <: o$
Parts 1 and 2 are the same as the B-GetMissing subcase of T-GetHMapAbsent. Part 3 is trivial as $\tau = \top$.

Case (BE-Get1).
Reduces to an error.

Case (BE-Get2).
Reduces to an error.

Case (B-Assoc). $v = \{\overrightarrow{(v_a\ v_b)}\}[k \mapsto v_v], \rho \vdash e_m \Downarrow \{\overrightarrow{(v_a\ v_b)}\}, \rho \vdash e_k \Downarrow k, \rho \vdash e_v \Downarrow v_v$

Subcase (T-AssocHMap). $\Gamma \vdash e'_m \Rightarrow e_m : (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}), \Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val}\ k), \Gamma \vdash e'_v \Rightarrow e_v : \tau, k \notin \mathcal{A}, e' = (\text{assoc}\ e'_m\ e'_k\ e'_v), e = (\text{assoc}\ e_m\ e_k\ e_v), \vdash (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}[k \mapsto \tau]\ \mathcal{A}) <: \tau, \psi_+ = \mathtt{tt}, \psi_- = \mathtt{ff}, o = \emptyset$
Parts 1 and 2 hold for the same reasons as T-True.

Case (BE-Assoc1).
Reduces to an error.

Case (BE-Assoc2).
Reduces to an error.

Case (BE-Assoc3).
Reduces to an error.

Case (B-IfFalse). $\rho \vdash e_1 \Downarrow \mathsf{false}$ or $\rho \vdash e_1 \Downarrow \mathsf{nil}, \rho \vdash e_3 \Downarrow v$

Subcase (T-If). $e' = (\text{if}\ e'_1\ e'_2\ e'_3), \Gamma \vdash e'_1 \Rightarrow e_1 : \tau_1\ ;\ \psi_{1_+}|\psi_{1_-}\ ;\ o_1, \Gamma, \psi_{1_+} \vdash e'_2 \Rightarrow e_2 : \tau\ ;\ \psi_{2_+}|\psi_{2_-}\ ;\ o, \Gamma, \psi_{1_-} \vdash e'_3 \Rightarrow e_3 : \tau\ ;\ \psi_{3_+}|\psi_{3_-}\ ;\ o, e = (\text{if}\ e_1\ e_2\ e_3), \psi_{2_+} \vee \psi_{3_+} \vdash \psi_+, \psi_{2_-} \vee \psi_{3_-} \vdash \psi_-$
For part 1, either $o = \emptyset$, or e evaluates to the result of e_3.
To prove part 2, we consider two cases:
- if $\mathsf{FalseVal}(v)$ then e_3 evaluates to a false value so $\rho \models \psi_{3_-}$, and thus $\rho \models \psi_{2_-} \vee \psi_{3_-}$ by M-Or,
- otherwise $\mathsf{TrueVal}(v)$, so e_3 evaluates to a true value so $\rho \models \psi_{3_+}$, and thus $\rho \models \psi_{2_+} \vee \psi_{3_+}$ by M-Or.

Part 3 is trivial as $\rho \vdash e_3 \Downarrow v$ and $\vdash v : \tau$ by the induction hypothesis on e_3.

Case (B-IfTrue). $\rho \vdash e_1 \Downarrow v_1, v_1 \neq \mathsf{false}, v_1 \neq \mathsf{nil}, \rho \vdash e_2 \Downarrow v$

Subcase (T-If). $e' = (\text{if}\ e'_1\ e'_2\ e'_3), \Gamma \vdash e'_1 \Rightarrow e_1 : \tau_1\ ;\ \psi_{1_+}|\psi_{1_-}\ ;\ o_1, \Gamma, \psi_{1_+} \vdash e'_2 \Rightarrow e_2 : \tau\ ;\ \psi_{2_+}|\psi_{2_-}\ ;\ o, \Gamma, \psi_{1_-} \vdash e'_3 \Rightarrow e_3 : \tau\ ;\ \psi_{3_+}|\psi_{3_-}\ ;\ o, e = (\text{if}\ e_1\ e_2\ e_3), \psi_{2_+} \vee \psi_{3_+} \vdash \psi_+, \psi_{2_-} \vee \psi_{3_-} \vdash \psi_-$
For part 1, either $o = \emptyset$, or e evaluates to the result of e_2.
To prove part 2, we consider two cases:
- if $\mathsf{FalseVal}(v)$ then e_2 evaluates to a false value so $\rho \models \psi_{2_-}$, and thus $\rho \models \psi_{2_-} \vee \psi_{3_-}$ by M-Or,
- otherwise $\mathsf{TrueVal}(v)$, so e_2 evaluates to a true value so $\rho \models \psi_{2_+}$, and thus $\rho \models \psi_{2_+} \vee \psi_{3_+}$ by M-Or.

Part 3 is trivial as $\rho \vdash e_2 \Downarrow v$ and $\vdash v : \tau$ by the induction hypothesis on e_2.

Case (BE-If).
Reduces to an error.

Case (BE-IfFalse).
Reduces to an error.

Case (BE-IfTrue).
Reduces to an error.

Case (B-Let). $e = (\text{let}\ [x\ e_1]\ e_2), \rho \vdash e_1 \Downarrow v_1, \rho[x \mapsto v_1] \vdash e_2 \Downarrow v$

Subcase (T-Let). $e' = (\text{let}\ [x\ e'_1]\ e'_2), \Gamma \vdash e'_1 \Rightarrow e'_1 : \sigma\ ;\ \psi_{1_+}|\psi_{1_-}\ ;\ o_1, \psi' = \overline{(\cup\ \mathbf{nil}\ \mathbf{false})}_x\ \supset\ \psi_{1_+}, \psi'' = (\cup\ \mathbf{nil}\ \mathbf{false})_x\ \supset\ \psi_{1_-}, \Gamma, \sigma_x, \psi', \psi'' \vdash e'_2 \Rightarrow e_2 : \tau\ ;\ \psi_+|\psi_-\ ;\ o$
For all the following cases (with a reminder that x is fresh) we apply the induction hypothesis on e_2. We justify this by noting that occurrences of x inside e_2 have the same type as e_1 and simulate the propositions of e_1 because
- $\Gamma, \sigma_x, \psi', \psi'' \vdash e'_2 \Rightarrow e_2 : \tau\ ;\ \psi_+|\psi_-\ ;\ o$,
- $\rho[x \mapsto v_1] \models \Gamma, \sigma_x, \psi', \psi''$,
- $\rho[x \mapsto v_1]$ is consistent, and
- $\rho[x \mapsto v_1] \vdash e_2 \Downarrow v$.

We prove parts 1, 2 and 3 by directly using the induction hypothesis on e_2.

Case (BE-Let).
Reduces to an error.

Case (B-Abs). $v = [\rho, \lambda x^{\sigma}.e_1]_{\mathsf{c}}$

Subcase (T-Clos). $e' = [\rho, \lambda x^{\sigma}.e_1]_{\mathsf{c}}, \exists \Gamma'.\rho \models \Gamma'$ and $\Gamma' \vdash \lambda x^{\sigma}.e_1 \Rightarrow \lambda x^{\sigma}.e_1 : \tau\ ;\ \psi_{f_+}|\psi_{f_-}\ ;\ o_f, e = [\rho, \lambda x^{\sigma}.e_1]_{\mathsf{c}}, \psi_+ = \mathtt{tt}, \psi_- = \mathtt{ff}, o = \emptyset$
We assume some Γ', such that

- $\rho \models \Gamma'$
- $\Gamma' \vdash \lambda x^{\sigma}.e_1 : \tau \; ; \; \psi_+|\psi_- \; ; \; o$.

Note the last rule in the derivation of $\Gamma' \vdash \lambda x^{\sigma}.e_1 : \tau \; ; \; \psi_+|\psi_- \; ; \; o$ must be T-Abs, so $\psi_+ = \mathtt{tt}$, $\psi_- = \mathtt{ff}$ and $o = \emptyset$. Thus parts 1 and 2 hold for the same reasons as T-True. Part 3 holds as v has the same type as $\lambda x^{\sigma}.e_1$ under Γ'.

Case (B-Abs). $v = [\rho, \lambda x^{\sigma}.e_1]_{\mathsf{c}}$, $\rho \vdash \lambda x^{\tau}.e_1 \Downarrow [\rho, \lambda x^{\sigma}.e_1]_{\mathsf{c}}$

Subcase (T-Abs). $e' = \lambda x^{\sigma}.e_1'$, $\Gamma, \sigma_x \vdash e_1' \Rightarrow e_1 : \tau \; ; \; \psi_{1+}|\psi_{1-} \; ; \; o_1, \vdash x{:}\sigma \xrightarrow[o_1]{\psi_{1+}|\psi_{1-}} \tau_1 <: \tau, \mathtt{tt} \vdash \psi_+, \mathtt{ff} \vdash \psi_-, o = \emptyset$

Parts 1 and 2 hold for the same reasons as T-True. Part 3 holds directly via T-Clos, since v must be a closure.

Case (BE-Error). $\rho \vdash e \Downarrow \mathsf{err}$

Subcase (T-Error). $e' = \mathsf{err}$, $e = \mathsf{err}$, $\tau = \bot$, $\psi_+ = \mathtt{ff}$, $\psi_- = \mathtt{ff}$, $o = \emptyset$
Trivially reduces to an error.

□

Theorem A.1 (Well-typed programs don't go wrong)**.** *If* $\vdash e' \Rightarrow e : \tau \; ; \; \psi_+|\psi_- \; ; \; o$ *then* $\not\vdash e \Downarrow wrong$.

Proof. Corollary of lemma A.8, since by lemma A.8 when $\vdash e' \Rightarrow e : \tau \; ; \; \psi_+|\psi_- \; ; \; o$, either $\vdash e \Downarrow v$ or $\vdash e \Downarrow \mathsf{err}$, therefore $\not\vdash e \Downarrow wrong$. □

Theorem A.2 (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* ψ'_+, ψ'_- *and* o'

Proof. Corollary of lemma A.8. □

d, e	$::= x \mid v \mid (e\ e) \mid \lambda x^{\tau}.e \mid (\text{if } e\ e\ e) \mid (\text{do } e\ e) \mid (\text{let } [x\ e]\ e) \mid \beta \mid R \mid E \mid M \mid G$	Expressions
v	$::= l \mid I \mid \{\} \mid c \mid n \mid s \mid m \mid [\rho, \lambda x^{\tau}.e]_{\mathsf{c}} \mid [v, t]_{\mathsf{m}}$	Values
m	$::= \{\overrightarrow{v \mapsto v}\}$	Map Values
c	$::= \mathit{class} \mid \mathit{n?}$	Constants
G	$::= (\text{get } e\ e) \mid (\text{assoc } e\ e\ e)$	Hash Maps
E	$::= (.\ e\ \mathit{fld}_C^C) \mid (.\ e\ (\mathit{mth}^C_{[[\overrightarrow{C}],C]}\ \overrightarrow{e})) \mid (\text{new}_{[\overrightarrow{C}]}\ C\ \overrightarrow{e})$	Non-Reflective Java Interop
R	$::= (.\ e\ \mathit{fld}) \mid (.\ e\ (\mathit{mth}\ \overrightarrow{e})) \mid (\text{new}\ C\ \overrightarrow{e})$	Reflective Java Interop
M	$::= (\text{defmulti}\ \tau\ e) \mid (\text{defmethod}\ e\ e\ e) \mid (\text{isa?}\ e\ e)$	Immutable First-Class Multimethods
σ, τ	$::= \top \mid C \mid (\mathbf{Val}\, l) \mid (\bigcup\ \overrightarrow{\tau}) \mid x{:}\tau \xrightarrow[o]{\psi\mid\psi} \tau \mid (\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}) \mid (\mathbf{Multi}\ \tau\ \tau)$	Types
$\mathcal{M}$	$::= \{\overrightarrow{k \mapsto \tau}\}$	HMap mandatory entries
$\mathcal{A}$	$::= \{\overrightarrow{k}\}$	HMap absent entries
$\mathcal{E}$	$::= \mathcal{C} \mid \mathcal{P}$	HMap completeness tags
l	$::= k \mid C \mid \mathsf{nil} \mid b$	Value types
b	$::= \mathsf{true} \mid \mathsf{false}$	Boolean values
ρ	$::= \{\overrightarrow{x \mapsto v}\}$	Value environments
ψ	$::= \tau_{\pi(x)} \mid \overline{\tau}_{\pi(x)} \mid \psi \supset \psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathbb{tt} \mid \mathbb{ff}$	Propositions
o	$::= \pi(x) \mid \emptyset$	Objects
π	$::= \overrightarrow{pe}$	Paths
pe	$::= \mathbf{class} \mid \mathbf{key}_k$	Path elements
Γ	$::= \overrightarrow{\psi}$	Proposition environments
t	$::= \{\overrightarrow{v \mapsto v}\}$	Dispatch tables
ce	$::= \{\mathsf{m} \mapsto \{\mathit{mth} \mapsto [[\overrightarrow{C}], C]\}, \mathsf{f} \mapsto \{\overrightarrow{\mathit{fld} \mapsto C}\}, \mathsf{c} \mapsto \{[\overrightarrow{C}]\}\}$	Class descriptors
$\mathcal{CT}$	$::= \{\overrightarrow{C \mapsto ce}\}$	Class Table
C	$::= \mathbf{Object} \mid \mathbf{K} \mid \mathbf{Class} \mid \mathbf{B} \mid \mathbf{Fn} \mid \mathbf{Multi} \mid \mathbf{Map} \mid \mathbf{Void}$	Class literals
I	$::= C\ \{\overrightarrow{\mathit{fld} : v}\}$	Class Values
β	$::= \mathit{wrong} \mid \mathsf{err}$	Wrong or error
α	$::= v \mid \beta$	Defined reductions
pol	$::= \mathsf{pos} \mid \mathsf{neg}$	Substitution Polarity

Figure A.1. Syntax of Terms, Types, Propositions, and Objects

nil	$\equiv$	(**Val** nil)
true	$\equiv$	(**Val** true)
false	$\equiv$	(**Val** false)

Figure A.2. Type abbreviations

$\Gamma \vdash e : \tau$	$\equiv$	$\Gamma \vdash e : \tau\ ;\ \psi_+ \mid \psi_-\ ;\ o$ for some ψ_+, ψ_- and o
$\tau[o/x]$	$\equiv$	$\tau[o/x]^{\mathsf{pos}}$
$\psi[o/x]$	$\equiv$	$\psi[o/x]^{\mathsf{pos}}$
$\psi \mid \psi[o/x]$	$\equiv$	$\psi \mid \psi[o/x]^{\mathsf{pos}}$
$o[o/x]$	$\equiv$	$o[o/x]^{\mathsf{pos}}$

Figure A.3. Judgment abbreviations

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

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

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

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

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

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

T-DO
$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : \tau_1\ ;\ \psi_{1+}|\psi_{1-}\ ;\ o_1 \qquad \Gamma, \psi_{1+} \vee \psi_{1-} \vdash e \Rightarrow e' : \tau\ ;\ \psi_+|\psi_-\ ;\ o}{\Gamma \vdash (\mathrm{do}\ e_1\ e) \Rightarrow (\mathrm{do}\ e_1'\ e') : \tau\ ;\ \psi_+|\psi_-\ ;\ o}$$

T-IF
$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow e_1' : \tau_1\ ;\ \psi_{1+}|\psi_{1-}\ ;\ o_1 \\ \Gamma, \psi_{1+} \vdash e_2 \Rightarrow e_2' : \tau\ ;\ \psi_+|\psi_-\ ;\ o \\ \Gamma, \psi_{1-} \vdash e_3 \Rightarrow e_3' : \tau\ ;\ \psi_+|\psi_-\ ;\ o \\ e' = (\mathrm{if}\ e_1'\ e_2'\ e_3')\end{array}}{\Gamma \vdash (\mathrm{if}\ e_1\ e_2\ e_3) \Rightarrow e' : \tau\ ;\ \psi_+|\psi_-\ ;\ o}$$

T-LET
$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow e_1' : \sigma\ ;\ \psi_{1+}|\psi_{1-}\ ;\ o_1 \\ \psi' = \overline{(\cup\ \textbf{nil}\ \textbf{false})}_x \supset \psi_{1+} \\ \psi'' = (\cup\ \textbf{nil}\ \textbf{false})_x \supset \psi_{1-} \\ \Gamma, \sigma_x, \psi', \psi'' \vdash e_2 \Rightarrow e_2' : \tau\ ;\ \psi_+|\psi_-\ ;\ o \\ e' = (\mathrm{let}\ [x\ e_1']\ e_2')\end{array}}{\Gamma \vdash (\mathrm{let}\ [x\ e_1]\ e_2) \Rightarrow e' : \tau[o_1/x]\ ;\ \psi_+|\psi_-[o_1/x]\ ;\ o[o_1/x]}$$

T-APP
$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow e_1 : x{:}\sigma \xrightarrow[o_f]{\psi_{f+}|\psi_{f-}} \tau\ ;\ \psi_+|\psi_-\ ;\ o \\ \Gamma \vdash e' \Rightarrow e_1' : \sigma\ ;\ \psi'_+|\psi'_-\ ;\ o'\end{array}}{\Gamma \vdash (e\ e') \Rightarrow (e_1\ e_1') : \tau[o'/x]\ ;\ \psi_{f+}|\psi_{f-}[o'/x]\ ;\ o_f[o'/x]}$$

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

T-CLOS
$$\frac{\exists \Gamma.\rho \models \Gamma \text{ and } \Gamma \vdash \lambda x^\tau.e \Rightarrow \lambda x^\tau.e' : \sigma\ ;\ \psi_+|\psi_-\ ;\ o}{\vdash [\rho, \lambda x^\tau.e]_{\mathsf{c}} \Rightarrow [\rho, \lambda x^\tau.e']_{\mathsf{c}} : \sigma\ ;\ \psi_+|\psi_-\ ;\ o}$$

T-ERROR
$$\Gamma \vdash \mathsf{err} \Rightarrow \mathsf{err} : \bot\ ;\ \mathtt{ff}|\mathtt{ff}\ ;\ \emptyset$$

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

Figure A.4. Standard Typing Rules

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-NEWSTATIC

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

T-FIELD

$$\frac{\Gamma \vdash e \Rightarrow e' : \sigma \qquad \vdash \sigma <: \textbf{Object} \qquad \mathsf{TJ}(\sigma) \;=\; C_1 \qquad fld \mapsto C_2 \in \mathcal{CT}[C_1][\mathsf{f}] \qquad \mathsf{JT}_{\mathsf{nil}}(C_2) \;=\; \tau}{\Gamma \vdash (.\; e\; fld) \Rightarrow (.\; e'\; fld_{C_2}^{C_1}) : \tau \;;\; \mathtt{tt}|\mathtt{tt} \;;\; \emptyset}$$

T-FIELDSTATIC

$$\frac{\mathsf{JT}(C_1) \;=\; \sigma \qquad \vdash \sigma <: \textbf{Object} \qquad \mathsf{JT}_{\mathsf{nil}}(C_2) \;=\; \tau \qquad \Gamma \vdash e \Rightarrow e' : \sigma}{\Gamma \vdash (.\; e\; fld_{C_2}^{C_1}) \Rightarrow (.\; e'\; fld_{C_2}^{C_1}) : \tau \;;\; \mathtt{tt}|\mathtt{tt} \;;\; \emptyset}$$

T-METHOD

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow e' : \sigma \\ \vdash \sigma <: \textbf{Object} \qquad \mathsf{TJ}(\sigma) \;=\; C_1 \qquad mth \mapsto [[\overrightarrow{C_i}], C_2] \in \mathcal{CT}[C_1][\mathsf{m}] \qquad \overrightarrow{\mathsf{JT}_{\mathsf{nil}}(C_i) \;=\; \tau_i} \qquad \overrightarrow{\Gamma \vdash e_i \Rightarrow e_i' : \tau_i} \qquad \mathsf{JT}_{\mathsf{nil}}(C_2) \;=\; \tau\end{array}}{\Gamma \vdash (.\; e\; (mth\,\overrightarrow{e_i})) \Rightarrow (.\; e'\; (mth_{[[\overrightarrow{C_i}], C_2]}^{C_1}\; \overrightarrow{e_i'})) : \tau \;;\; \mathtt{tt}|\mathtt{tt} \;;\; \emptyset}$$

T-METHODSTATIC

$$\frac{\overrightarrow{\mathsf{JT}(C_i) \;=\; \tau_i} \qquad \mathsf{JT}(C_1) \;=\; \sigma \qquad \vdash \sigma <: \textbf{Object} \qquad \mathsf{JT}_{\mathsf{nil}}(C_2) \;=\; \tau \qquad \Gamma \vdash e \Rightarrow e' : \sigma \qquad \overrightarrow{\Gamma \vdash e_i \Rightarrow e_i' : \tau_i}}{\Gamma \vdash (.\; e\; (mth_{[[\overrightarrow{C_i}], C_2]}^{C_1}\; \overrightarrow{e_i})) \Rightarrow (.\; e'\; (mth_{[[\overrightarrow{C_i}], C_2]}^{C_1}\; \overrightarrow{e_i'})) : \tau \;;\; \mathtt{tt}|\mathtt{tt} \;;\; \emptyset}$$

T-CLASS

$$\Gamma \vdash C : (\textbf{Val}\; C) \;;\; \mathtt{tt}|\mathtt{ff} \;;\; \emptyset$$

T-INSTANCE

$$\Gamma \vdash C\; \{\overrightarrow{fld : v}\} : C \;;\; \mathtt{tt}|\mathtt{ff} \;;\; \emptyset$$

Figure A.5. Java Interop Typing Rules

T-DEFMULTI

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

T-DEFMETHOD

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

T-ISA

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

T-MULTI

$$\frac{\vdash v \Rightarrow v' : \tau \qquad \overrightarrow{\vdash v_k \Rightarrow v_k' : \top} \qquad \overrightarrow{\vdash v_v \Rightarrow v_v' : \sigma}}{\vdash [v, \{\overrightarrow{v_k \mapsto v_v}\}]_{\mathsf{m}} \Rightarrow [v', \{\overrightarrow{v_k' \mapsto v_v'}\}]_{\mathsf{m}} : (\textbf{Multi}\; \sigma\; \tau) \;;\; \mathtt{tt}|\mathtt{ff} \;;\; \emptyset}$$

Figure A.6. Multimethod Typing Rules

$$\text{T-HMap}\ \frac{\overrightarrow{\vdash v_k \Rightarrow v_k' : (\textbf{Val}\, k)} \qquad \overrightarrow{\vdash v_v \Rightarrow v_v' : \tau_v} \qquad \mathcal{M} = \{\overrightarrow{k \mapsto \tau_v}\}}{\vdash \{\overrightarrow{v_k \mapsto v_v}\} \Rightarrow \{\overrightarrow{v_k' \mapsto v_v'}\} : (\textbf{HMap}^{\mathcal{C}}\ \mathcal{M})\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset}$$

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

$$\text{T-GetHMap}\ \frac{\Gamma \vdash e \Rightarrow e' : (\bigcup\ \overrightarrow{(\textbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})}^{i})\ ;\ \psi_{1+}|\psi_{1-}\ ;\ o \qquad \Gamma \vdash e_k \Rightarrow e_k' : (\textbf{Val}\, k) \qquad \overrightarrow{\mathcal{M}[k] = \tau}^{i}}{\Gamma \vdash (\text{get}\ e\ e_k) \Rightarrow (\text{get}\ e'\ e_k') : (\bigcup\ \overrightarrow{\tau}^{i})\ ;\ \mathtt{tt}|\mathtt{tt}\ ;\ \textbf{key}_k(x)[o/x]}$$

$$\text{T-GetHMapAbsent}\ \frac{\Gamma \vdash e \Rightarrow e' : (\textbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A})\ ;\ \psi_{1+}|\psi_{1-}\ ;\ o \qquad \Gamma \vdash e_k \Rightarrow e_k' : (\textbf{Val}\, k) \qquad k \in \mathcal{A}}{\Gamma \vdash (\text{get}\ e\ e_k) \Rightarrow (\text{get}\ e'\ e_k') : \textbf{nil}\ ;\ \mathtt{tt}|\mathtt{tt}\ ;\ \textbf{key}_k(x)[o/x]}$$

$$\text{T-GetHMapPartialDefault}\ \frac{\Gamma \vdash e \Rightarrow e' : (\textbf{HMap}^{\mathcal{P}}\ \mathcal{M}\ \mathcal{A})\ ;\ \psi_{1+}|\psi_{1-}\ ;\ o \qquad \Gamma \vdash e_k \Rightarrow e_k' : (\textbf{Val}\, k) \qquad k \notin dom(\mathcal{M}) \qquad k \notin \mathcal{A}}{\Gamma \vdash (\text{get}\ e\ e_k) \Rightarrow (\text{get}\ e'\ e_k') : \top\ ;\ \mathtt{tt}|\mathtt{tt}\ ;\ \textbf{key}_k(x)[o/x]}$$

$$\text{T-AssocHMap}\ \frac{\Gamma \vdash e \Rightarrow e' : (\textbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}) \qquad \Gamma \vdash e_k \Rightarrow e_k' : (\textbf{Val}\, k) \qquad \Gamma \vdash e_v \Rightarrow e_v' : \tau \qquad k \notin \mathcal{A} \qquad e' = (\text{assoc}\ e'\ e_k'\ e_v')}{\Gamma \vdash (\text{assoc}\ e\ e_k\ e_v) \Rightarrow e' : (\textbf{HMap}^{\mathcal{E}}\ \mathcal{M}[k \mapsto \tau]\ \mathcal{A})\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset}$$

Figure A.7. Map Typing Rules

$$\text{SO-Refl}\ \vdash o <: o \qquad \text{S-Refl}\ \vdash \tau <: \tau \qquad \text{SO-Top}\ \vdash o <: \emptyset \qquad \text{S-Top}\ \vdash \tau <: \top$$

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

$$\text{S-FunMono}\ \frac{}{\vdash x{:}\sigma\ \xrightarrow[o]{\psi_+|\psi_-}\ \tau <: \textbf{Fn}} \qquad \text{S-Object}\ \vdash C <: \textbf{Object} \qquad \text{S-SClass}\ \vdash (\textbf{Val}\, C) <: \textbf{Class}$$

$$\text{S-SBool}\ \vdash (\textbf{Val}\, b) <: \textbf{B} \qquad \text{S-SKw}\ \vdash (\textbf{Val}\, k) <: \textbf{K}$$

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

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

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

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

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

$$\text{S-HMapP}\ \frac{\forall i.\, \mathcal{M}[k_i] = \sigma_i \text{ and } \vdash \sigma_i <: \tau_i}{\vdash (\textbf{HMap}^{\mathcal{C}}\ \mathcal{M}\ \mathcal{A}') <: (\textbf{HMap}^{\mathcal{P}}\ \{\overrightarrow{k \mapsto \tau}\}^{i}\ \mathcal{A})} \qquad \text{S-HMapMono}\ \vdash (\textbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}) <: \textbf{Map}$$

Figure A.8. Subtyping rules

$$\begin{array}{lcl} \mathsf{JT}(\textbf{Void}) & = & \textbf{nil} \\ \mathsf{JT}(C) & = & C \\ \mathsf{JT}_{\mathsf{nil}}(\textbf{Void}) & = & \textbf{nil} \\ \mathsf{JT}_{\mathsf{nil}}(C) & = & (\bigcup\ \textbf{nil}\ C) \end{array}$$

Figure A.9. Java Type Conversion

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

Figure A.10. Constant Typing

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

Figure A.11. Primitives

$$
\begin{array}{lcll}
\mathsf{IsAProps}(\mathbf{class}(\pi(x)), (\mathbf{Val}\, C)) & = & C_{\pi(x)}|\overline{C}_{\pi(x)} & \\
\mathsf{IsAProps}(o, (\mathbf{Val}\, l)) & = & ((\mathbf{Val}\, l)_x|\overline{(\mathbf{Val}\, l)}_x)[o/x] & \\
 & & \text{if } l \neq C & \\
\mathsf{IsAProps}(o, \tau) & = & \mathtt{tt}\,|\,\mathtt{tt} & \text{otherwise} \\
\mathsf{IsA}(v, v) = \mathsf{true} & & v \neq C & \\
\mathsf{IsA}(C, C') = \mathsf{true} & & \vdash C <: C' & \\
\mathsf{IsA}(v, v') = \mathsf{false} & & \text{otherwise} &
\end{array}
$$

Figure A.12. Definition of isa?

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

Figure A.13. Definition of get-method

$$\textsc{B-Local}\ \frac{\rho(x) = v}{\rho \vdash x \Downarrow v} \qquad \textsc{B-Do}\ \frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e \Downarrow v}{\rho \vdash (\text{do } e_1\ e) \Downarrow v} \qquad \textsc{B-Let}\ \frac{\rho \vdash e_a \Downarrow v_a \quad \rho[x \mapsto v_a] \vdash e \Downarrow v}{\rho \vdash (\text{let } [x\ e_a]\ e) \Downarrow v}$$

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

$$\textsc{B-IfFalse}\ \frac{\rho \vdash e_1 \Downarrow \mathsf{false} \text{ or } \rho \vdash e_1 \Downarrow \mathsf{nil} \quad \rho \vdash e_3 \Downarrow v}{\rho \vdash (\text{if } e_1\ e_2\ e_3) \Downarrow v} \qquad \textsc{B-Abs}\ \rho \vdash \lambda x^{\tau}.e \Downarrow [\rho, \lambda x^{\tau}.e]_{\mathsf{c}}$$

$$\textsc{B-BetaClosure}\ \frac{\rho \vdash e_f \Downarrow [\rho_c, \lambda x^{\tau}.e_b]_{\mathsf{c}} \quad \rho \vdash e_a \Downarrow v_a \quad \rho_c[x \mapsto v_a] \vdash e_b \Downarrow v}{\rho \vdash (e_f\ e_a) \Downarrow v} \qquad \textsc{B-Delta}\ \frac{\rho \vdash e \Downarrow c \quad \rho \vdash e' \Downarrow v \quad \delta(c, v) = v'}{\rho \vdash (e\ e') \Downarrow v'}$$

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

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

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

$$\textsc{B-New}\ \frac{\overrightarrow{\rho \vdash e_i \Downarrow v_i} \quad \mathsf{JVM}_{\mathsf{new}}[C_1, [\overrightarrow{C_i}], [\overrightarrow{v_i}]] = v}{\rho \vdash (\text{new}_{[\overrightarrow{C_i}]}\ C\ \overrightarrow{e_i}) \Downarrow v} \qquad \textsc{B-DefMulti}\ \frac{\rho \vdash e \Downarrow v_d \quad v = [v_d, \{\}]_{\mathsf{m}}}{\rho \vdash (\text{defmulti } \tau\ e) \Downarrow v}$$

$$\textsc{B-DefMethod}\ \frac{\rho \vdash e \Downarrow [v_d, t]_{\mathsf{m}} \quad \rho \vdash e' \Downarrow v_v \quad \rho \vdash e_f \Downarrow v_f \quad v = [v_d, t[v_v \mapsto v_f]]_{\mathsf{m}}}{\rho \vdash (\text{defmethod } e\ e'\ e_f) \Downarrow v} \qquad \textsc{B-IsA}\ \frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e_2 \Downarrow v_2 \quad \mathsf{IsA}(v_1, v_2) = v}{\rho \vdash (\text{isa? } e_1\ e_2) \Downarrow v}$$

$$\textsc{B-Assoc}\ \frac{\rho \vdash e \Downarrow m \quad \rho \vdash e_k \Downarrow k \quad \rho \vdash e_v \Downarrow v_v \quad v = m[k \mapsto v_v]}{\rho \vdash (\text{assoc } e\ e_k\ e_v) \Downarrow v} \qquad \textsc{B-Get}\ \frac{\rho \vdash e \Downarrow m \quad \rho \vdash e' \Downarrow k \quad k \in dom(m) \quad m[k] = v}{\rho \vdash (\text{get } e\ e') \Downarrow v}$$

$$\textsc{B-GetMissing}\ \frac{\rho \vdash e \Downarrow m \quad \rho \vdash e' \Downarrow k \quad k \notin dom(m)}{\rho \vdash (\text{get } e\ e') \Downarrow \mathsf{nil}}$$

Figure A.14. Operational Semantics

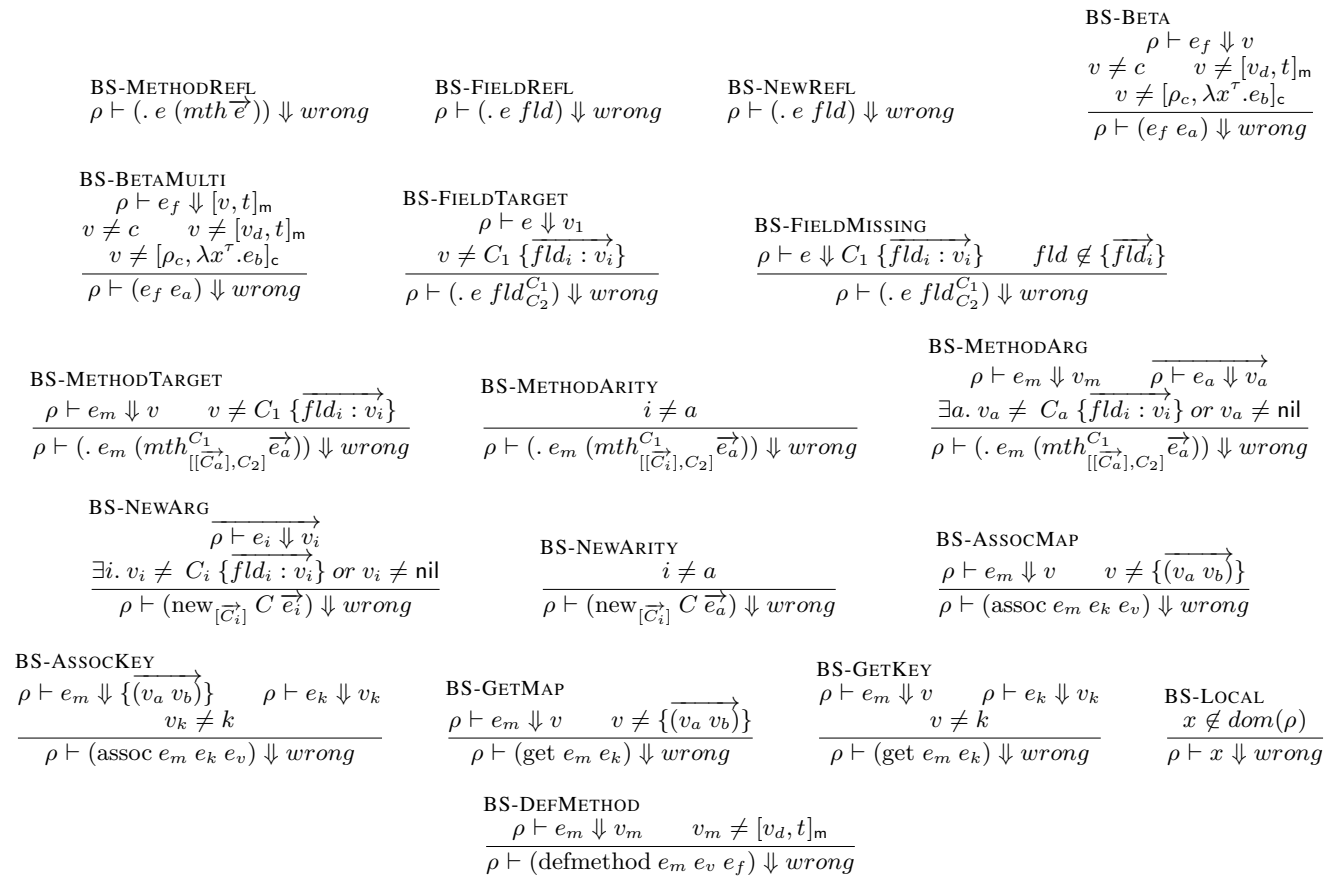

Figure A.15. Stuck programs

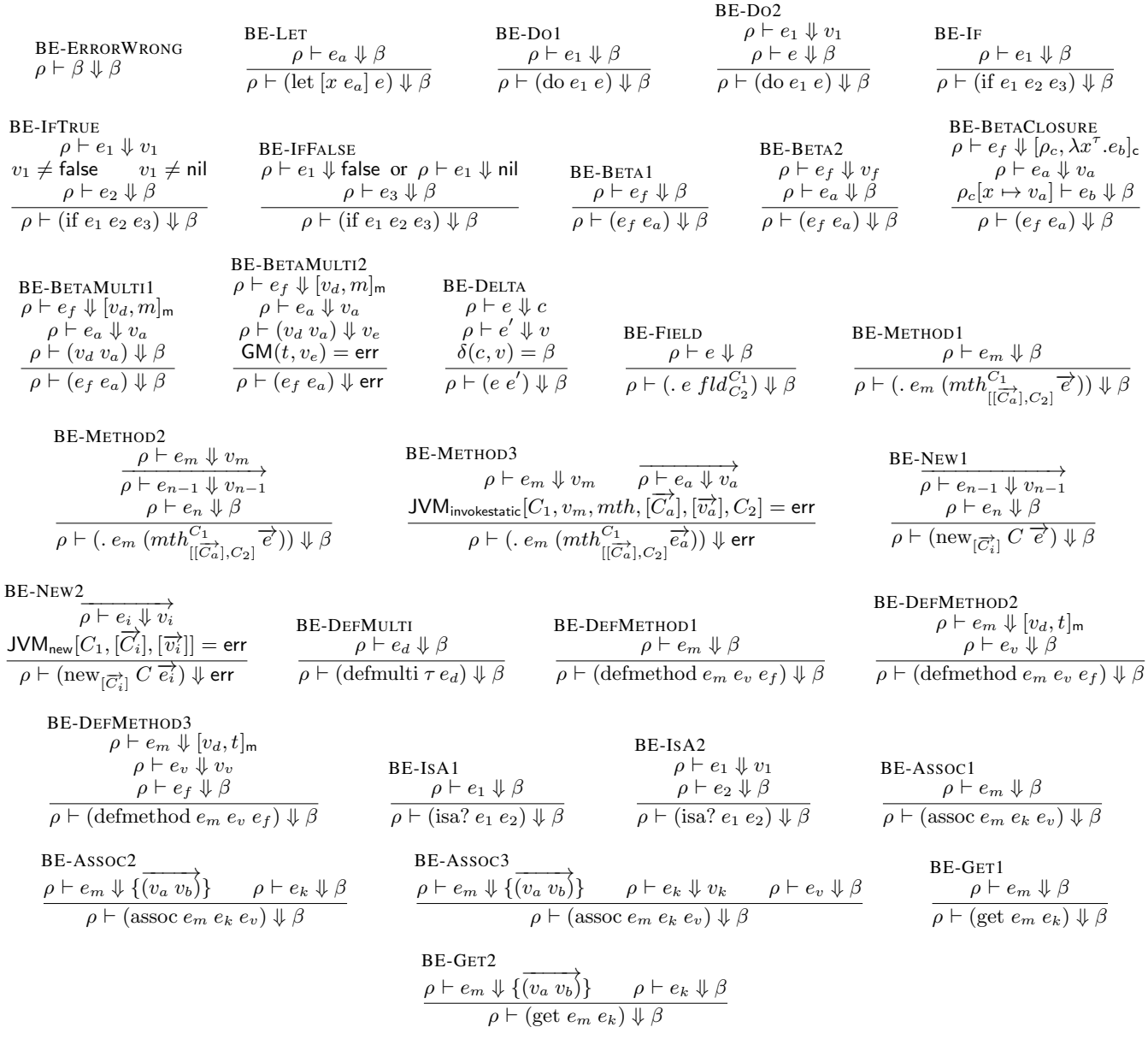

Figure A.16. Error and stuck propagation

$$
\begin{array}{lcll}
\rho(x) & = & v & (x, v) \in \rho \\
\rho(\mathbf{key}_k(o)) & = & (\text{get } \rho(o)\ k) & \\
\rho(\mathbf{class}(o)) & = & (\mathit{class}\ \rho(o)) &
\end{array}
$$

Figure A.17. Path translation

$$
\begin{array}{lcll}
\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}) & \text{if } \mathcal{M}[k] = \tau \\
\mathsf{update}((\mathbf{HMap}^{\mathcal{E}}\ \mathcal{M}\ \mathcal{A}), \nu, \pi :: \mathbf{key}_k) & = & \bot & \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}) & \text{if } \vdash \mathbf{nil} <: \tau,\ k \notin dom(\mathcal{M}) \text{ and } k \notin \mathcal{A} \\
 & & \quad (\mathbf{HMap}^{\mathcal{P}}\ \mathcal{M}\ (\mathcal{A} \cup \{k\}))) & \\
\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 } \vdash \mathbf{nil} \not<: \nu,\ k \notin 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) & \\
 & & & \\
\mathsf{restrict}(\tau, \sigma) & = & \bot & \text{if } \nexists v. \vdash v : \tau\ ;\ \psi\ ;\ o \text{ and} \vdash v : \sigma\ ;\ \psi'\ ;\ o' \\
\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}(\tau, \sigma) & = & \tau & \text{otherwise}
\end{array}
$$

Figure A.18. Type Update

$$
\text{M-Or}\ \frac{\rho \models \psi_1 \text{ or } \rho \models \psi_2}{\rho \models \psi_1 \vee \psi_2}
\qquad
\text{M-Imp}\ \frac{\rho \models \psi_1 \text{ implies } \rho \models \psi_2}{\rho \models \psi_1 \supset \psi_2}
\qquad
\text{M-And}\ \frac{\rho \models \psi_1 \qquad \rho \models \psi_2}{\rho \models \psi_1 \wedge \psi_2}
\qquad
\text{M-Top}\ \rho \models \mathtt{tt}
$$

$$
\text{M-Type}\ \frac{\vdash \rho(\pi(x)) : \tau\ ;\ \psi_+|\psi_-\ ;\ o}{\rho \models \tau_{\pi(x)}}
\qquad
\text{M-NotType}\ \frac{\begin{array}{c}\vdash \rho(\pi(x)) : \sigma\ ;\ \psi_+|\psi_-\ ;\ o \\ \text{there is no } v \text{ such that } \vdash v : \tau\ ;\ \psi_{1+}|\psi_{1-}\ ;\ o_1 \text{ and } \vdash v : \sigma\ ;\ \psi_{2+}|\psi_{2-}\ ;\ o_2\end{array}}{\rho \models \overline{\tau}_{\pi(x)}}
$$

Figure A.19. Satisfaction Relation

$$
\text{L-Atom}\ \frac{\psi \in \Gamma}{\Gamma \vdash \psi}
\qquad
\text{L-True}\ \Gamma \vdash \mathtt{tt}
\qquad
\text{L-False}\ \frac{\Gamma \vdash \mathtt{ff}}{\Gamma \vdash \psi}
\qquad
\text{L-AndI}\ \frac{\Gamma \vdash \psi_1 \qquad \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \wedge \psi_2}
\qquad
\text{L-AndE}\ \frac{\Gamma, \psi_1, \psi_2 \vdash \psi}{\Gamma, \psi_1 \wedge \psi_2 \vdash \psi}
$$

$$
\text{L-ImplI}\ \frac{\Gamma, \psi_1 \vdash \psi_2}{\Gamma, \psi_1 \vdash \psi_1 \supset \psi_2}
\qquad
\text{L-ImplE}\ \frac{\Gamma \vdash \psi_1 \qquad \Gamma \vdash \psi_1 \supset \psi_2}{\Gamma \vdash \psi_2}
\qquad
\text{L-OrI}\ \frac{\Gamma \vdash \psi_1 \text{ or } \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \vee \psi_2}
$$

$$
\text{L-OrE}\ \frac{\Gamma, \psi_1 \vdash \psi \qquad \Gamma, \psi_2 \vdash \psi}{\Gamma, \psi_1 \vee \psi_2 \vdash \psi}
\qquad
\text{L-Sub}\ \frac{\Gamma \vdash \tau_{\pi(x)} \qquad \vdash \tau <: \sigma}{\Gamma \vdash \sigma_{\pi(x)}}
\qquad
\text{L-SubNot}\ \frac{\Gamma \vdash \overline{\sigma}_{\pi(x)} \qquad \vdash \tau <: \sigma}{\Gamma \vdash \overline{\tau}_{\pi(x)}}
$$

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

(The metavariable ν ranges over τ and $\overline{\tau}$ (without variables).)

Figure A.20. Proof System

$$
\begin{array}{lcll}
\psi_+|\psi_-[o/x]^{pol} & = & \psi_+[o/x]^{pol}|\psi_-[o/x]^{pol} & \\
\\
\nu_{\pi(x)}[\pi'(y)/x]^{pol} & = & (\nu[\pi'(y)/x]^{pol})_{\pi(\pi'(y))} & \\
\nu_{\pi(x)}[\emptyset/x]^{\mathsf{pos}} & = & \mathtt{tt} & \\
\nu_{\pi(x)}[\emptyset/x]^{\mathsf{neg}} & = & \mathtt{ff} & \\
\nu_{\pi(x)}[o/z]^{pol} & = & \nu_{\pi(x)} & x \neq z \text{ and } z \notin \mathsf{fv}(\nu) \\
\nu_{\pi(x)}[o/z]^{\mathsf{pos}} & = & \mathtt{tt} & x \neq z \text{ and } z \in \mathsf{fv}(\nu) \\
\nu_{\pi(x)}[o/z]^{\mathsf{neg}} & = & \mathtt{ff} & x \neq z \text{ and } z \in \mathsf{fv}(\nu) \\
\mathtt{tt}[o/x]^{pol} & = & \mathtt{tt} & \\
\mathtt{ff}[o/x]^{pol} & = & \mathtt{ff} & \\
(\psi_1 \supset \psi_2)[o/x]^{\mathsf{pos}} & = & \psi_1[o/x]^{\mathsf{neg}} \supset \psi_2[o/x]^{\mathsf{pos}} & \\
(\psi_1 \supset \psi_2)[o/x]^{\mathsf{neg}} & = & \psi_1[o/x]^{\mathsf{pos}} \supset \psi_2[o/x]^{\mathsf{neg}} & \\
(\psi_1 \vee \psi_2)[o/x]^{pol} & = & \psi_1[o/x]^{pol} \vee \psi_2[o/x]^{pol} & \\
(\psi_1 \wedge \psi_2)[o/x]^{pol} & = & \psi_1[o/x]^{pol} \wedge \psi_2[o/x]^{pol} & \\
\\
\pi(x)[\pi'(y)/x]^{pol} & = & \pi(\pi'(y)) & \\
\pi(x)[\emptyset/x]^{pol} & = & \emptyset & \\
\pi(x)[o/z]^{pol} & = & \pi(x) & x \neq z \\
\emptyset[o/x]^{pol} & = & \emptyset &
\end{array}
$$

Substitution on types is capture-avoiding structural recursion.

Figure A.21. Substitution