

TYPED CLOSURE IN THEORY AND PRACTICE

Ambrose Bonnaire-Sergeant

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the School of Informatics, Computing, and Engineering,

Indiana University

June 2019

ProQuest Number: 13879596

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 13879596

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Sam Tobin-Hochstadt, Ph.D.

Chung-chieh Shan, Ph.D.

Ryan R. Newton, Ph.D.

Lawrence S. Moss, Ph.D.

24th April, 2019

Copyright 2019
Ambrose Bonnaire-Sergeant
All rights reserved

Acknowledgements

I first touched down in Bloomington ready to start my graduate career—that is, until my puzzled cab driver Juannita informed me with a gasp we were in Bloomington, *Illinois*. She kindly agreed to drive me to IU, a 12-hour roundtrip—a humbling start to graduate school. The next day, reconnecting with Dan Friedman and the inspirational Will Byrd came full circle. Thanks Jason Hemann, Cam & Rebecca Swords, Jaime Guerrero, and Tori Vollmer for helping me find my feet.

Since then, Andrew Kent and I have done most things together, like deciphering set-theoretic types, floating down a Eugene river, weekly Zelda gaming, and dissertating. Andrew and his delightful family took me under their wing—thanks Carrie, Charlotte, Harrison, Sydney, and Dwight.

Thanks to my donators, talk attendees, and users, including Chas Emerick, Brandon Bloom, David Nolen, Ghadi Shayban, Nicola Mometto, Devin Walters, Reid McKenzie, Nathan Sorenson, Eric Normand, Craig Andera, Jim Duey, Kyle Kingsbury, Colin Fleming, Paula Gearon, Nikko Patten, and Claire Alvis who were very encouraging. CircleCI enabled discussion of Typed Clojure “in practice,” and their post-mortem summarized my frustrations with Typed Clojure.

Sam taught me how (and when) to turn my implementations into research. He has been supportive, trusting, and always game to clarify my bizarre ideas. Mike Vollmer helped brainstorm the annotator; I’m grateful for Mike’s presense and support. I fondly remember our conversations, nagivating Mumbai together, and bad-movie nights. The annotator took shape at weekly meetings—thanks Matteo Cimini, Rajan Walia, Spenser Bauman, Jeremy Siek, Mike Vitousek, Deyaaeldeen Almahallawi, Caner Deric, Sarah Spall, and David Christiansen for your attention and suggestions.

Symbolic closures were the result of many conversations. Andre Kuhlenschmidt’s interest in them during my final PL Wonks talk helped me through the last weeks before my defense. Thanks to Wonks regulars Ryan Scott, Kyle Carter, Vikraman Choudhury, Matthew Heimerdinger, Paulette Koronkevich, Aaron Hsu, Praveen Narayanan, and Chaitanya Koparkar for your engagement.

Thanks to my committee Ryan, Larry, and Ken for your feedback. I found Ryan’s compilers course and Larry’s logic course invigorating, and I’m grateful to Ken for always asking the right questions.

Thanks to my friends and family, especially Mum, Dad, and Aaron. Having a partner with experience in Computer Science research has been wonderful, and my wife Marcela Poffald is my greatest advocate. Her feedback has improved and clarified my work, and her continuous requests to be referred to as “Mrs. Typed Clojure” with a matching cap never fail to make me blush.

Ambrose Bonnaire-Sergeant

TYPED CLOJURE IN THEORY AND PRACTICE

Typed Clojure is an optional type system for the Clojure programming language that aims to type check idiomatic Clojure code. This dissertation presents the design of Typed Clojure, formalizes Typed Clojure’s underlying theory, studies its effectiveness in real-world code bases, and proposes several extensions to help address its shortcomings.

I develop a formal model of Typed Clojure that includes key features like hash-maps, multimethods, Java interoperability, and occurrence typing, and prove the model type sound. Then, I demonstrate that Typed Clojure’s design is useful and corresponds to actual usage patterns with an empirical study of real-world Typed Clojure usage in over 19,000 lines of code. This experience also revealed several usability shortcomings in Typed Clojure.

First, the top-level annotation burden needed to port untyped code is prohibitively high. We present an automatic annotator for Typed Clojure to ease this burden, using runtime observations to synthesize heterogeneous, recursive type annotations. We evaluate our experience using the annotator by porting several open-source projects.

Second, pre-expanding macros before type checking makes type checking brittle. We describe and implement a new analyzer for Clojure code that can provide the foundation of an alternative approach where the user provides custom type rules for macros.

Third, too many local functions require annotations. We present a hybrid approach of symbolic execution and type checking that helps check some common higher-order Clojure idioms.

Sam Tobin-Hochstadt, Ph.D.

Chung-chieh Shan, Ph.D.

Ryan R. Newton, Ph.D.

Lawrence S. Moss, Ph.D.

Contents

Chapter 1. Introduction	1
1.1. My Thesis	1
1.2. Structure of this Dissertation	1
1.3. Previously Published Work	2
Part I. Practical Optional Types for Clojure	3
Chapter 2. Abstract	3
Chapter 3. Background: Clojure with static typing	4
3.1. Contributions	6
Chapter 4. Overview of Typed Clojure	7
4.1. Clojure	7
4.2. Typed Clojure	7
4.3. Java interoperability	8
4.4. Multimethods	9
4.5. Heterogeneous hash-maps	10
4.6. HMaps and multimethods, joined at the hip	11
Chapter 5. A Formal Model of λ_{TC}	14
5.1. Core type system	14
5.2. Java Interoperability	19
5.3. Multimethod preliminaries: isa?	21
5.4. Multimethods	21
5.5. Precise Types for Heterogeneous maps	24
5.6. Proof system	27
Chapter 6. Metatheory	28
Chapter 7. Experience	30
7.1. Implementation	30
7.2. Evaluation	30
7.3. Further challenges	32
Chapter 8. Conclusion	34
Part II. Automatic Annotations for Typed Clojure	36
Chapter 9. Abstract	36
Chapter 10. Introduction	37
Chapter 11. Overview	40

Chapter 12. Formalism	45
12.1. Collection phase	45
12.2. Inference phase	48
Chapter 13. Evaluation	54
13.1. Experiment 1: Manual inspection	54
13.2. Experiment 2: Changes needed to type check	57
13.3. Experiment 3: Specs pass unit tests	62
Chapter 14. Extensions	64
14.1. Space-efficient tracking	64
14.2. Lazy tracking	65
14.3. Automatic contracts with <code>clojure.spec</code>	66
Chapter 15. Conclusion	68
Part III. Typed Clojure Implementations	69
Chapter 16. Background	69
Chapter 17. Expand before checking	70
17.1. Upfront Analysis with <code>tools.analyzer</code>	70
17.2. Extensibility	70
Chapter 18. Interleaved expansion and checking	77
18.1. Interleaved Analysis with <code>core.typed.analyzer</code>	78
18.2. Extensibility in Interleaved checking	82
Chapter 19. Managing Analysis Side effects	84
19.1. Clojure’s Evaluation Model	84
19.2. Is order-of-expansion defined in Clojure?	85
19.3. Preserving evaluation order during type checking	87
Part IV. Symbolic Closures	92
Chapter 20. Background	92
20.1. Enhancing Local Type Inference	92
Chapter 21. Delayed checking for Unannotated Local Functions	96
21.1. Overview	97
21.2. Formal model	102
Chapter 22. Symbolic Closure Metatheory Conjectures	113
Part V. Related and Future Work	117
Chapter 23. Related Work to Typed Clojure	117
Chapter 24. Related work to Automatic Annotations	119
Chapter 25. Related Work to Extensible Type Systems	122
Chapter 26. Related Work to Symbolic Closures	124
Chapter 27. Future work	133
27.1. Future work for Automatic Annotations	133

27.2. Future work for Extensible Types	133
27.3. Future work for Symbolic Closures	133
Bibliography	134
Appendix A. Full rules for λ_{TC}	141
Appendix B. Soundness for λ_{TC}	154
Curriculum Vitae	

CHAPTER 1

Introduction

1.1. My Thesis

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

1.2. Structure of this Dissertation

This document progresses in several parts that support my thesis statement.

Part I motivates and presents the design of Typed Clojure. It addresses both parts of my thesis statement.

- *Typed Clojure is sound* I formalize Typed Clojure, including its characteristic features like hash-maps, multimethods, and Java interoperability, and prove the model type sound.
- *Typed Clojure is practical* I present an empirical study of real-world Typed Clojure usage in over 19,000 lines of code, showing its features correspond to actual usage patterns.

The results and industry feedback of this work inspired three distinct research directions to help improve the experience of using Typed Clojure.

- Part II presents a solution to lower the annotation burden in real-world Typed Clojure programs. I formalize and implement a tool to automatically annotate types for top-level user and library definitions, and empirically study the manual changes needed for the generated annotations to pass type checking.
- Part III describes the design and implementation of a new code analyzer for Clojure, in service of enabling user-provided type rules for Clojure macros to help make type checking complex macro usages more robust.
- Part IV motivates and describes *symbolic closure types*, a technique that enhances type checking with symbolic execution, that helps check some common Clojure idioms via a compatible extension of Typed Clojure’s original design.

Finally, Part V presents the related work and future directions for each part.

1.3. Previously Published Work

Part I has been published:

- Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical Optional Types for Clojure. In *Proceedings of the 25th European Symposium on Programming*, 2016. (ESOP '16)

Part II is in submission:

- Ambrose Bonnaire-Sergeant, and Sam Tobin-Hochstadt. Squash the work: A Workflow for Typing Untyped Programs that use Ad-Hoc Data Structures. *In Submission*

Part I

Practical Optional Types for Clojure

CHAPTER 2

Abstract

Typed Clojure is an optional type system for Clojure, a dynamic language in the Lisp family that targets the JVM. Typed Clojure enables Clojure programmers to gain greater confidence in the correctness of their code via static type checking while remaining in the Clojure world, and has acquired significant adoption in the Clojure community. Typed Clojure repurposes Typed Racket's *occurrence typing*, an approach to statically reasoning about predicate tests, and also includes several new type system features to handle existing Clojure idioms.

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

We provide a formal model of the Typed Clojure type system incorporating these and other features, with a proof of soundness. Additionally, Typed Clojure is now in use by numerous corporations and developers working with Clojure, and we present a quantitative analysis on the use of type system features in two substantial code bases.

CHAPTER 3

Background: Clojure with static typing

The popularity of dynamically-typed languages in software development, combined with a recognition that types often improve programmer productivity, software reliability, and performance, has led to the recent development of a wide variety of optional and gradual type systems aimed at checking existing programs written in existing languages. These include TypeScript [77] and Flow [31] for JavaScript, Hack [35] for PHP, and mypy [52] for Python among the optional systems, and Typed Racket [76], Reticulated Python [78], and Gradualtalk [1] among gradually-typed systems.¹

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

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

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

¹We 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”.

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, `null` is a distinct type in Typed Clojure, designed to automatically rule out null-pointer exceptions.

An example of these features is given in Figure 3.1. Here, the `pname` multimethod dispatches on the `class` of the argument—for `Strings`, the first method implementation is called, for `Files`, the second. The `String` method calls a `File` constructor, returning a non-nil `File` instance—the `getName` method on `File` requires a non-nil target, returning a nilable type.

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

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

```

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

```

FIGURE 3.1. A simple Typed Clojure program (delimiters: **Java interoperation** (green), **type annotation** (blue), **function invocation** (black), **collection literal** (red), **other** (gray))

3.1. Contributions

Our main contributions are as follows:

- (1) We motivate and describe Typed Clojure, an optional type system for Clojure that understands existing Clojure idioms.
- (2) We present a sound formal model for three crucial type system features: multi-methods, Java interoperability, and heterogeneous 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 part begins with an example-driven presentation of the main type system features in Section 4. We then incrementally present a core calculus for Typed Clojure covering all of these features together in Section 5 and prove type soundness (Section 6). We then present an empirical analysis of significant code bases written in `core.typed`—the full implementation of Typed Clojure—in Section 7. Finally, we discuss related work and conclude.

CHAPTER 4

Overview of Typed Clojure

We now begin a tour of the central features of Typed Clojure, beginning with Clojure itself. Our presentation uses the full Typed Clojure system to illustrate key type system ideas,¹ before studying the core features in detail in Section 5.

4.1. Clojure

Clojure [44] is a Lisp that runs on the Java Virtual Machine with support for concurrent programming and immutable data structures in a mostly-functional style. Clojure provides easy interoperability with existing Java libraries, with Java values being like any other Clojure value. However, this smooth interoperability comes at the cost of pervasive `nil`, which leads to the possibility of null pointer exceptions—a drawback we address in Typed Clojure.

4.2. Typed Clojure

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

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

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

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

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

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

¹Full examples: <https://github.com/typedclojure/esop16>

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

Flow analysis. Occurrence typing [75] models type-based control flow. In `greetings`, a branch ensures `repeat` is never passed `nil`.

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

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

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

4.3. Java interoperability

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

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

Example 1

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

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

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

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

Example 2

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

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

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

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

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

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

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

Example 3

4.4. Multimethods

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

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

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

Example 4

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

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

For example, `(hi :en)` evaluates to `"hello"`—it executes the `:en` method because

- `(= (identity :en) :en)` is true and
- `(= (identity :en) :fr)` is false.

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

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

Class-based dispatch. For class values, multimethods can choose methods based on subclassing relationships. Recall the multimethod from Figure 3.1. The dispatch function `class` dictates whether the `String` or `File` method is chosen. The multimethod dispatch rules use `isa?`, a hybrid predicate which is both a subclassing check for classes and an equality check for other values.

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

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

4.5. Heterogeneous hash-maps

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

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

Example 5

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

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

This says `:en` and `:fr` are known entries mapped to strings, and the map is fully specified—that is, no other entries exist—by `:complete?` being `true`. `HMap` types default to partial specification. `'{:en Str :fr Str}` abbreviates: `(HMap :mandatory :en Str, :fr Str)`.

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

Example 6

HMaps in practice. The next example is extracted from a production system at CircleCI, a company with a large production Typed Clojure system (Section 7.2 presents a case study and empirical result from this code base).

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

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

As `EncKeyPair` is fully specified, we remove extra keys like `:priv` via `dissoc`, which returns a new map that is the first argument without the entry named by the second argument. Notice removing `dissoc` causes a type error.

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

4.6. HMaps and multimethods, joined at the hip

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

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

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

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

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

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

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

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

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

Example 8

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

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

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

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

Example 9

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

In the `nil` method, `(isa? (class (:desserts o)) nil)` must be true—which implies

```
(class (:desserts o))
```

is `nil`. Since lookups on missing keys return `nil`, either

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

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

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

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

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

CHAPTER 5

A Formal Model of λ_{TC}

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

5.1. Core type system

We start with a review of occurrence typing [75], the foundation of λ_{TC} .

Expressions. Syntax is given in Figure 5.1. Expressions e include variables x , values v , applications, abstractions, conditionals, and let expressions. All binding forms introduce fresh variables—a subtle but important point since our type environments are not simply dictionaries. Values include booleans b , `nil`, class literals C , keywords k , integers n , constants c , and strings s . Lexical closures $[\rho, \lambda x^\tau. e]_c$ close value environments ρ —which map bindings to values—over functions.

Types. Types σ or τ include the top type \top , *untagged* unions ($\bigcup \vec{\tau}$), singletons (**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 \perp , (**Val**`nil`) to **nil**, (**Val**`true`) to **true**, and (**Val**`false`) to **false**. The difference between the types (**Val** C) and C is subtle. The former is inhabited by class literals like **K** and the result of (`class :a`)—the latter by *instances* of classes, like a keyword literal `:a`, an instance of the type **K**. Function types $x:\sigma \xrightarrow{o|\psi} \tau$ contain *latent* (terminology from [55]) propositions ψ , object o , and return type τ , which may refer to the function argument x . They are instantiated with the actual object of the argument in applications.

Objects. Each expression is associated with a symbolic representation called an *object*. For example, variable m has object m ; (`class (:lunch m)`) has object **class**(**key**.`lunch`(m)); and `42` has the *empty* object \emptyset since it is unimportant in our system. Figure 5.1 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

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

FIGURE 5.1. Syntax of Terms, Types, Propositions and Objects

elements—**class** and **key_k**—representing the results of calling *class* and looking up a keyword *k*, respectively.

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

$$\begin{array}{c}
\text{LC-LET} \\
\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x \mapsto \sigma \vdash e_2 : \tau}{\Gamma \vdash (\text{let } [x e_1] e_2) : \tau}
\end{array}
\qquad
\begin{array}{c}
\text{LC-LOCAL} \\
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}
\end{array}$$

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 .

$$\begin{array}{c}
\text{T0-LET} \\
\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, \sigma_x \vdash e_2 : \tau}{\Gamma \vdash (\text{let } [x e_1] e_2) : \tau}
\end{array}
\qquad
\begin{array}{c}
\text{T0-LOCAL} \\
\frac{\Gamma \vdash \tau_x}{\Gamma \vdash x : \tau}
\end{array}$$

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

We further extend logical statements to *propositional logic*. Figure 5.1 describes the syntax for propositions ψ , consisting of positive and negative *type propositions* about non-empty objects— $\tau_{\pi(x)}$ and $\bar{\tau}_{\pi(x)}$ 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 (tt) and impossible (ff) propositions. The full proof system judgement $\Gamma \vdash \psi$ says *proposition environment* Γ proves proposition ψ .

$$\begin{array}{c}
\text{T-LOCAL} \quad \frac{\Gamma \vdash \tau_x}{\sigma = (\bigcup \mathbf{nil} \ \mathbf{false})} \quad \text{T-ABS} \quad \frac{\Gamma, \sigma_x \vdash e : \sigma' ; \psi_+ | \psi_- ; o}{\Gamma \vdash \lambda x^\sigma . e : x : \sigma \xrightarrow[\sigma]{\psi_+ | \psi_-} \sigma' ; \text{tt} | \text{ff} ; \emptyset} \quad \text{T-IF} \quad \frac{\Gamma \vdash e_1 : \tau_1 ; \psi_{1+} | \psi_{1-} ; o_1 \quad \Gamma, \psi_{1+} \vdash e_2 : \tau ; \psi_+ | \psi_- ; o \quad \Gamma, \psi_{1-} \vdash e_3 : \tau ; \psi_+ | \psi_- ; o}{\Gamma \vdash (\text{if } e_1 \ e_2 \ e_3) : \tau ; \psi_+ | \psi_- ; o} \\
\\
\text{T-KW} \quad \frac{\Gamma \vdash k : (\mathbf{Val} \ k) ; \text{tt} | \text{ff} ; \emptyset}{\Gamma \vdash n : \mathbf{N} ; \text{tt} | \text{ff} ; \emptyset} \quad \text{T-NIL} \quad \frac{\Gamma \vdash \mathbf{nil} : \mathbf{nil} ; \text{ff} | \text{tt} ; \emptyset}{\Gamma \vdash c : \delta_\tau(c) ; \text{tt} | \text{ff} ; \emptyset} \quad \text{T-STR} \quad \frac{\Gamma \vdash s : \mathbf{S} ; \text{tt} | \text{ff} ; \emptyset}{\Gamma \vdash \mathbf{true} : \mathbf{true} ; \text{tt} | \text{ff} ; \emptyset} \\
\text{T-CLASS} \quad \frac{\Gamma \vdash C : (\mathbf{Val} \ C) ; \text{tt} | \text{ff} ; \emptyset}{\Gamma \vdash c : \delta_\tau(c) ; \text{tt} | \text{ff} ; \emptyset} \quad \text{T-TRUE} \quad \frac{\Gamma \vdash \mathbf{true} : \mathbf{true} ; \text{tt} | \text{ff} ; \emptyset}{\Gamma \vdash c : \delta_\tau(c) ; \text{tt} | \text{ff} ; \emptyset} \\
\\
\text{T-LET} \quad \frac{\Gamma \vdash e_1 : \sigma ; \psi_{1+} | \psi_{1-} ; o_1 \quad \psi' = (\bigcup \mathbf{nil} \ \mathbf{false})_x \supset \psi_{1+} \quad \psi'' = (\bigcup \mathbf{nil} \ \mathbf{false})_x \supset \psi_{1-} \quad \Gamma, \sigma_x, \psi', \psi'' \vdash e_2 : \tau ; \psi_+ | \psi_- ; o}{\Gamma \vdash (\mathbf{let} \ [x \ e_1] \ e_2) : \tau[o_1/x] ; \psi_+ | \psi_- [o_1/x] ; o[o_1/x]} \\
\\
\text{T-APP} \quad \frac{\Gamma \vdash e : x : \sigma \xrightarrow[\sigma_f]{\psi_{f+} | \psi_{f-}} \tau ; \psi_+ | \psi_- ; o \quad \Gamma \vdash e' : \sigma ; \psi'_+ | \psi'_- ; o'}{\Gamma \vdash (e \ e') : \tau[o'/x] ; \psi_{f+} | \psi_{f-} [o'/x] ; o_f[o'/x]} \quad \text{T-SUBSUME} \quad \frac{\Gamma \vdash e : \tau ; \psi_+ | \psi_- ; o \quad \Gamma, \psi_+ \vdash \psi'_+ \quad \Gamma, \psi_- \vdash \psi'_- \quad \vdash \tau <: \tau' \quad \vdash o <: o'}{\Gamma \vdash e : \tau' ; \psi'_+ | \psi'_- ; o'}
\end{array}$$

FIGURE 5.2. Core typing rules

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

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

To attain the type of o , we must solve for τ in $\Gamma \vdash \tau_o$, under proposition environment $\Gamma = \mathbf{Order}_o, (\mathbf{Val} : \text{combo})_{\text{key}:\text{Meal}(o)}$ 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 5.6.

$$\begin{array}{c}
\text{S-UNIONSUPER} \\
\frac{\exists i. \vdash \tau <: \sigma_i}{\vdash \tau <: (\bigcup \vec{\sigma}^i)} \\
\text{S-UNIONSUB} \\
\frac{\overrightarrow{\vdash \tau_i <: \sigma}}{\vdash (\bigcup \vec{\tau}^i) <: \sigma} \\
\text{S-FUNMONO} \\
\vdash x:\sigma \xrightarrow{o}^{\psi_+|\psi_-} \tau <: \mathbf{Fn} \\
\text{S-OBJECT} \\
\vdash C <: \mathbf{Object} \\
\text{S-SCCLASS} \\
\vdash (\mathbf{Val} C) <: \mathbf{Class} \\
\text{S-SBOOL} \\
\vdash (\mathbf{Val} b) <: \mathbf{B} \\
\text{S-FUN} \\
\frac{\vdash \sigma' <: \sigma \quad \vdash \tau <: \tau' \quad \psi_+ \vdash \psi'_+ \quad \psi_- \vdash \psi'_- \quad \vdash o <: o'}{\vdash x:\sigma \xrightarrow{o}^{\psi_+|\psi_-} \tau <: x:\sigma' \xrightarrow{o'}^{\psi'_+|\psi'_-} \tau'} \\
\text{S-REFL} \quad \text{S-TOP} \\
\vdash \tau <: \tau \quad \vdash \tau <: \top \\
\text{S-SKW} \\
\vdash (\mathbf{Val} k) <: \mathbf{K}
\end{array}$$

FIGURE 5.3. Core subtyping rules

$$\begin{array}{c}
\text{B-IFTRUE} \\
\frac{\rho \vdash e_1 \Downarrow v_1 \quad v_1 \neq \mathbf{false} \quad v_1 \neq \mathbf{nil} \quad \rho \vdash e_2 \Downarrow v}{\rho \vdash (\mathbf{if} e_1 e_2 e_3) \Downarrow v} \\
\text{B-IFFALSE} \\
\frac{\rho \vdash e_1 \Downarrow \mathbf{false} \quad \text{or} \quad \rho \vdash e_1 \Downarrow \mathbf{nil} \quad \rho \vdash e_3 \Downarrow v}{\rho \vdash (\mathbf{if} e_1 e_2 e_3) \Downarrow v}
\end{array}$$

FIGURE 5.4. Select core semantics

Typing judgment. We formalize our system following Tobin-Hochstadt and Felleisen [75]. 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 . For ease of presentation, we omit e' when it is easily inferred.

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' .

Typing rules. The core typing rules are given as Figure 5.2. We introduce the interesting rules with the complement number predicate as a running example.

$$(1) \quad \lambda d^{\top}. (\mathbf{if} (n? d) \mathbf{false} \mathbf{true})$$

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

First, in T-App the operator $e = n?$ is checked with T-Const, which uses δ_τ (Figure 5.6, dynamic semantics in the supplemental material) to type constants. $n?$ is a predicate over numbers, and *class* returns its argument’s class.

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

$$(2) \quad \Gamma \vdash d : \top ; \overline{(\bigcup \mathbf{nil} \mathbf{false})}_d | (\bigcup \mathbf{nil} \mathbf{false})_d ; d$$

which encodes the type, proposition, and object information about variables. The proposition $\overline{(\cup \mathbf{nil} \mathbf{false})}_d$ says “it is not the case that variable d is of type $(\cup \mathbf{nil} \mathbf{false})$ ”; $(\cup \mathbf{nil} \mathbf{false})_d$ says “ d is of type $(\cup \mathbf{nil} \mathbf{false})$ ”.

Finally, the T-App rule substitutes the operand’s object o' for the parameter x in the latent type, propositions, and object. The proposition \mathbf{N}_d says “ d is of type \mathbf{N} ”; $\overline{\mathbf{N}}_d$ says “it is not the case that d is of type \mathbf{N} ”. The object d is the symbolic representation of what the expression d evaluates to.

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

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

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

$$\begin{aligned} \Gamma, \mathbf{N}_d \vdash \mathbf{false} : \mathbf{B} ; \overline{\mathbf{N}}_d | \mathbf{N}_d ; \emptyset \\ \Gamma, \overline{\mathbf{N}}_d \vdash \mathbf{true} : \mathbf{B} ; \overline{\mathbf{N}}_d | \mathbf{N}_d ; \emptyset \end{aligned}$$

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

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

Subtyping. Figure 5.3 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 \mathbf{B} , class literals are instances of \mathbf{Class} and keywords are instances of \mathbf{K} . Instances of classes C are subtypes of \mathbf{Object} . Function types are subtypes of \mathbf{Fn} . All types except for \mathbf{nil} are subtypes of \mathbf{Object} , so \top is similar to $(\cup \mathbf{nil} \mathbf{Object})$. 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 [75]. 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 (included in supplemental material) however a notable difference is \mathbf{nil} is a false value, which affects the semantics of if (Figure 5.4).

$e ::= \dots \mid (. e fld) \mid (. e (mth \vec{e})) \mid (new C \vec{e})$ $\mid (. e fld_C^C) \mid (. e (mth_{[[\vec{C}], C]}^C \vec{e})) \mid (new_{[\vec{C}]} C \vec{e})$	Expressions Non-reflective Expressions
$v ::= \dots \mid C \overrightarrow{\{fld : v\}}$	Values
$ce ::= \{\mathbf{m} \mapsto \{mth \mapsto [[\vec{C}], C]\}, \mathbf{f} \mapsto \{fld \mapsto \vec{C}\}, \mathbf{c} \mapsto \{[\vec{C}]\}\}$	Class descriptors
$\mathcal{CT} ::= \{C \mapsto ce\}$	Class Table

$$\frac{\text{T-NEW} \quad \overrightarrow{[\vec{C}_i] \in \mathcal{CT}[C][c]} \quad \overrightarrow{JT_{\text{nil}}(C_i) = \tau_i} \quad \overrightarrow{\Gamma \vdash e_i \Rightarrow e'_i : \tau_i} \quad JT(C) = \tau}{\Gamma \vdash (new C \vec{e}_i) : \tau ; \text{tt}|\text{ff} ; \emptyset}$$

$$\frac{\text{T-METHOD} \quad \Gamma \vdash e \Rightarrow e' : \sigma \quad \overrightarrow{TJ(\sigma) = C_1} \quad \overrightarrow{mth \mapsto [[\vec{C}_i], C_2] \in \mathcal{CT}[C_1][\mathbf{m}]} \quad \overrightarrow{JT_{\text{nil}}(C_i) = \tau_i} \quad \overrightarrow{\Gamma \vdash e_i \Rightarrow e'_i : \tau_i} \quad \overrightarrow{JT_{\text{nil}}(C_2) = \tau} \quad \vdash \sigma <: \mathbf{Object}}{\Gamma \vdash (. e (mth \vec{e}_i)) : \tau ; \text{tt}|\text{tt} ; \emptyset}$$

$$\frac{\text{T-FIELD} \quad \Gamma \vdash e \Rightarrow e' : \sigma \quad \vdash \sigma <: \mathbf{Object} \quad \overrightarrow{TJ(\sigma) = C_1} \quad \overrightarrow{fld \mapsto C_2 \in \mathcal{CT}[C_1][\mathbf{f}]} \quad \overrightarrow{JT_{\text{nil}}(C_2) = \tau}}{\Gamma \vdash (. e fld) : \tau ; \text{tt}|\text{tt} ; \emptyset}$$

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

$$\frac{\text{B-FIELD} \quad \rho \vdash e \Downarrow v \quad \overrightarrow{JVM_{\text{getstatic}}[C_1, v_1, fld, C_2] = v} \quad \text{B-NEW} \quad \overrightarrow{\rho \vdash e_i \Downarrow v_i} \quad \overrightarrow{JVM_{\text{new}}[C_1, [\vec{C}_i], [\vec{v}_i]] = v}}{\rho \vdash (. e fld_{C_2}^{C_1}) \Downarrow v \quad \rho \vdash (new_{[\vec{C}_i]} C \vec{e}_i) \Downarrow v}$$

$$\frac{\text{B-METHOD} \quad \rho \vdash e_m \Downarrow v_m \quad \overrightarrow{\rho \vdash e_a \Downarrow v_a} \quad \overrightarrow{JVM_{\text{invokestatic}}[C_1, v_m, mth, [\vec{C}_a], [\vec{v}_a], C_2] = v}}{\rho \vdash (. e_m (mth_{[[\vec{C}_a], C_2]}^{C_1} \vec{e}_a)) \Downarrow v}$$

FIGURE 5.5. Java Interoperability Syntax, Typing and Operational Semantics

$$\begin{aligned} \delta_\tau(\text{class}) &= x : \top \xrightarrow[\text{class}(x)]{\text{tt}|\text{tt}} (\bigcup \mathbf{nil} \mathbf{Class}) \\ \delta_\tau(n?) &= x : \top \xrightarrow[\emptyset]{\mathbf{N}_x|\mathbf{N}_x} \mathbf{B} \end{aligned}$$

FIGURE 5.6. Constant typing

5.2. Java Interoperability

We present Java interoperability in a restricted setting without class inheritance, overloading or Java Generics. We extend the syntax in Figure 5.5 with Java field lookups and calls to methods and constructors. To prevent ambiguity between zero-argument methods and fields, we use Clojure’s primitive “dot” syntax: field accesses are written $(. e fld)$ and method calls $(. e (mth \vec{e}))$.

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

$$(4) \quad (. (new \mathbf{F} \text{"a/b"}) (getParent))$$

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

$$(5) \quad (. (new_{[\mathbf{S}_1]} \mathbf{F} \text{"a/b"}) (getParent_{[\square, \mathbf{S}_1]}^{\mathbf{F}}))$$

From the left, the one-argument constructor for \mathbf{F} takes a \mathbf{S} , and the `getParent` method of \mathbf{F} takes zero arguments and returns a \mathbf{S} .

We now walk through this conversion.

Constructors. First we check and convert `(new \mathbf{F} "a/b")` to `(new[\mathbf{S}_1] \mathbf{F} "a/b")`. The T-New typing rule checks and rewrites constructors. To check `(new \mathbf{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 $\tau_1 = (\bigcup \mathbf{nil} \mathbf{S})$ 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 \mathbf{F} is converted to a non-nil Typed Clojure type $\tau = \mathbf{F}$ for the return type, and the propositions say constructors can never be false—constructors can never produce the internal boolean value that Clojure uses for `false`, or `nil`. Finally, the constructor rewrites to `(new[\mathbf{S}_1] \mathbf{F} "a/b")`.

Methods. Next we convert `(. (new[\mathbf{S}_1] \mathbf{F} "a/b") (getParent))` to the non-reflective expression `(. (new[\mathbf{S}_1] \mathbf{F} "a/b") (getParent[\square, \mathbf{S}_1] \mathbf{F}))`. The T-Method rule for unresolved methods checks:

$$(. (new_{[\mathbf{S}_1]} \mathbf{F} \text{"a/b"}) (getParent)).$$

We verify the target type $\sigma = \mathbf{F}$ is non-nil by T-New. The overload is chosen from the class table based on $C_1 = \mathbf{F}$ —there is at most one. The nilable return type $\tau = (\bigcup \mathbf{nil} \mathbf{S})$ is given, and the entire expression rewrites to expression 5.

The T-Field rule (Figure 5.5) is like T-Method, but without arguments.

The evaluation rules B-Field, B-New and B-Method (Figure 5.5) simply evaluate their arguments and call the relevant JVM operation, which we do not model—Section 6 states our exact assumptions. There are no evaluation rules for reflective Java interoperability, since there are no typing rules that rewrite to reflective calls.

5.3. Multimethod preliminaries: isa?

We now consider the `isa?` operation, a core part of the multimethod dispatch mechanism. Recalling the examples in Section 4.4, `isa?` is a subclassing test for classes, but otherwise is an equality test. The T-IsA rule uses `IsAProps` (Figure 5.7), a metafunction which produces the propositions for `isa?` expressions.

To demonstrate the first `IsAProps` case, the expression $(\text{isa? } (\text{class } x) \mathbf{K})$ is true if x is a keyword, otherwise false. When checked with T-IsA, the object of the left subexpression $o = \mathbf{class}(x)$ (which starts with the `class` path element) and the type of the right subexpression $\tau = (\mathbf{Val} \mathbf{K})$ (a singleton class type) together trigger the first `IsAProps` case $\text{IsAProps}(\mathbf{class}(x), (\mathbf{Val} \mathbf{K})) = \mathbf{K}_x | \overline{\mathbf{K}}_x$, giving propositions that correspond to our informal description $\psi_+ | \psi_- = \mathbf{K}_x | \overline{\mathbf{K}}_x$.

The second `IsAProps` case captures the simple equality mode for non-class singleton types. For example, the expression $(\text{isa? } x \text{:en})$ produces true when x evaluates to `:en`, otherwise it produces false. Using T-IsA, it has the propositions $\psi_+ | \psi_- = \text{IsAProps}(x, (\mathbf{Val} \text{:en})) = (\mathbf{Val} \text{:en})_x | \overline{(\mathbf{Val} \text{:en})}_x$ since $o = x$ and $\tau = (\mathbf{Val} \text{:en})$. The side condition on the second `IsAProps` case ensures we are in equality mode—if x can possibly be a class in $(\text{isa? } x \mathbf{Object})$, `IsAProps` uses its conservative default case, since if x is a class literal, subclassing mode could be triggered. Capture-avoiding substitution of objects $[o/x]$ used in this case erases propositions that would otherwise have \emptyset substituted in for their objects—it is defined in the appendix.

The operational behavior of `isa?` is given by B-IsA (Figure 5.7). `IsA` explicitly handles classes in the second case.

5.4. Multimethods

Figure 5.7 presents *immutable* multimethods without default methods to ease presentation. Figure 5.8 translates the mutable Example 4 to λ_{TC} .

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

Next, we show how to check $(\text{defmethod } hi_0 \text{:en } \lambda x^{\mathbf{K}}.\text{“hello”})$. 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	$::= \dots \mid (\text{defmulti } \tau e) \mid (\text{defmethod } e e e) \mid (\text{isa? } e e)$	Expressions
v	$::= \dots \mid [v, t]_m$	Values
t	$::= \{v \mapsto \vec{v}\}$	Dispatch tables
σ, τ	$::= \dots \mid (\mathbf{Multi} \tau \tau)$	Types

T-DEFMULTI

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

T-DEFMETHOD

$$\tau_m = x:\tau \xrightarrow{o}^{\psi_+|\psi_-} \sigma \quad \tau_d = x:\tau \xrightarrow{o'}^{\psi'_+|\psi'_-} \sigma' \quad \Gamma \vdash e_m \Rightarrow e'_m : (\mathbf{Multi} \tau_m \tau_d)$$

$$\frac{\Gamma \vdash e_v \Rightarrow e'_v : \tau_v \quad \text{IsAProps}(o', \tau_v) = \psi''_+|\psi''_- \quad \Gamma, \tau_x, \psi''_+ \vdash e_b : \sigma ; \psi_+|\psi_- ; o \quad e' = (\text{defmethod } e'_m e'_v \lambda x^\tau. e_b)}{\Gamma \vdash (\text{defmethod } e_m e_v \lambda x^\tau. e_b) : (\mathbf{Multi} \tau_m \tau_d) ; \mathbb{tt}|\text{fff} ; \emptyset}$$

T-ISA

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

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

S-PMULTIFN

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

S-PMULTI

$$\frac{\vdash \sigma <: \sigma' \quad \vdash \tau <: \tau'}{\vdash (\mathbf{Multi} \sigma \tau) <: (\mathbf{Multi} \sigma' \tau')}$$

S-MULTIMONO

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

B-DEFMULTI

$$\frac{\rho \vdash e \Downarrow v_d \quad v = [v_d, \{\}]_m}{\rho \vdash (\text{defmulti } \tau e) \Downarrow v}$$

B-DEFMETHOD

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

$$\begin{aligned} \text{GM}(t, v_e) &= v_f \quad \text{if } \vec{v}_{fs} = \{v_f\} \text{ where } \vec{v}_{fs} = \{v_f | v_k \mapsto v_f \in t \text{ and } \text{IsA}(v_e, v_k) = \text{true}\} \\ \text{GM}(t, v_e) &= \text{err} \quad \text{otherwise} \end{aligned}$$

B-ISA

$$\frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e_2 \Downarrow v_2 \quad \text{IsA}(v_1, v_2) = v}{\rho \vdash (\text{isa? } e_1 e_2) \Downarrow v} \quad \begin{array}{l} \text{IsA}(v, v) = \text{true} \quad v \neq C \\ \text{IsA}(C, C') = \text{true} \quad \vdash C <: C' \\ \text{IsA}(v, v') = \text{false} \quad \text{otherwise} \end{array}$$

B-BETAMULTI

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

FIGURE 5.7. Multimethod Syntax, Typing and Operational Semantics

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

FIGURE 5.8. Multimethod example

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 ψ''_+ , which is assumed when checking the method body. Our example checks with e_m being of type $(\mathbf{Multi} x: \mathbf{K} \rightarrow \mathbf{S} x: \mathbf{K} \xrightarrow[x]{\text{tt}|\text{tt}} \mathbf{K})$ with $o' = x$ (from below the arrow on the right argument of the previous type) and $\tau_v = (\mathbf{Val}:\text{en})$. Then $\psi''_+ = (\mathbf{Val}:\text{en})_x$ from $\text{lsAProps}(x, (\mathbf{Val}:\text{en})) = (\mathbf{Val}:\text{en})_x | \overline{(\mathbf{Val}:\text{en})}_x$ (see Section 5.3). Since $\tau = \mathbf{K}$, we check the method body with

$$\mathbf{K}_x, (\mathbf{Val}:\text{en})_x \vdash \text{“hello”} : \mathbf{S} ; \text{tt}|\text{tt} ; \emptyset.$$

Finally from the interface type τ_m , we know $\psi_+ = \psi_- = \text{tt}$, and $o = \emptyset$, which also agrees with the method body, above. Notice the overall type of a `defmethod` is the same as its first subexpression e_m .

It is worth noting the lack of special typing rules for overlapping methods—each method is checked independently based on local type information.

Subtyping. Multimethods are functions, via S-PMultiFn, which says a multimethod can be upcast to its interface type. Multimethod call sites are then handled by T-App via T-Subsume. Other rules are given in Figure 5.7.

Semantics. Multimethod definition semantics are also given in Figure 5.7. B-DefMulti creates a multimethod with the given dispatch function and an empty dispatch table. B-DefMethod produces a new multimethod with an extended dispatch table.

The overall dispatch mechanism is summarised by B-BetaMulti. First the dispatch function v_d is applied to the argument v' to obtain the dispatch value v_e . Based on v_e , the GM metafunction (Figure 5.7) extracts a method v_f from the method table t and applies it to the original argument for the final result.

$e ::= \dots$	$ $ (get $e e$) $ $ (assoc $e e e$)	Expressions
$v ::= \dots$	$ $ $\{\}$	Values
$\tau ::= \dots$	$ $ ($\mathbf{HMap}^\mathcal{E} \mathcal{M} \mathcal{A}$)	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

T-ASSOCHMAP

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

T-GETHMAP

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

T-GETHMAPABSENT

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

T-GETHMAPPARTIALDEFAULT

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

S-HMAPMONO

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

S-HMAPP

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

B-ASSOC	B-GET	B-GETMISSING
$\rho \vdash e \Downarrow m \quad \rho \vdash e_k \Downarrow k$	$\rho \vdash e \Downarrow m \quad \rho \vdash e' \Downarrow k$	$\rho \vdash e \Downarrow m$
$\rho \vdash e_v \Downarrow v_v$	$k \in \text{dom}(m)$	$\rho \vdash e' \Downarrow k \quad k \notin \text{dom}(m)$
$\rho \vdash (\text{assoc } e e_k e_v) \Downarrow m[k \mapsto v_v]$	$\rho \vdash (\text{get } e e') \Downarrow m[k]$	$\rho \vdash (\text{get } e e') \Downarrow \mathbf{nil}$

FIGURE 5.9. HMap Syntax, Typing and Operational Semantics

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

FIGURE 5.10. Restrict and remove

5.5. Precise Types for Heterogeneous maps

Figure 5.9 presents heterogeneous map types. The type $(\mathbf{HMap}^\mathcal{E} \mathcal{M} \mathcal{A})$ contains \mathcal{M} , a map of *present* entries (mapping keywords to types), \mathcal{A} , a set of keyword keys that are known to

be *absent* and tag \mathcal{E} which is either \mathcal{C} (“complete”) if the map is fully specified by \mathcal{M} , and \mathcal{P} (“partial”) if there are *unknown* entries. The partially specified map of `lunch` in Example 6 is written $(\mathbf{HMap}^{\mathcal{P}}\{(\mathbf{Val}:\text{en}) \mathbf{S}, (\mathbf{Val}:\text{fr}) \mathbf{S}\} \{\})$ (abbreviated **Lu**). The type of the fully specified map `breakfast` in Example 5 elides the absent entries, written $(\mathbf{HMap}^{\mathcal{C}}\{(\mathbf{Val}:\text{en}) \mathbf{S}, (\mathbf{Val}:\text{fr}) \mathbf{S}\})$ (abbreviated **Bf**). To ease presentation, if an HMap has completeness tag \mathcal{C} then \mathcal{A} is elided and implicitly contains all keywords not in the domain of \mathcal{M} —dissociating keys is not modelled, so the set of absent entries otherwise never grows. 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`.

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

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

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

An absent lookup is checked by `T-GetHMapAbsent`.

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

The result type is **nil**—since **Bf** is fully specified—with return object $\mathbf{key}_{:\text{bocce}}(b)$.

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

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

The result type is \top —since **Lu** has an unknown `:bocce` entry—with return object $\mathbf{key}_{:\text{bocce}}(u)$. Notice propositions are erased once they enter a HMap type.

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

FIGURE 5.11. Type update (the metavariable ν ranges over τ and $\bar{\tau}$ (without variables), $\vdash \mathbf{nil} \not\prec: \bar{\tau}$ when $\vdash \mathbf{nil} <: \tau$, see Figure 5.10 for restrict and remove.)

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

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

The result type is \mathbf{S} , and the return object is $\mathbf{key}_{\text{en}}(u)$. However, lookups of :bocce on $(\bigcup \mathbf{Bf Lu})$ maps are unsupported. This restriction still allows us to check many of the examples in Section 4—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. Adding a rule to handle Example 9 is otherwise straightforward.

Extending a map with T-AssocHMap preserves its completeness.

$$\lambda b \mathbf{Bf}.(\text{assoc } b : \text{au} \text{ "beans"})$$

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

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

The semantics for `get` and `assoc` are straightforward.

5.6. Proof system

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

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

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

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

$$\Gamma \vdash \text{update}(\mathbf{Order}, (\mathbf{Val}:\text{combo}), [\mathbf{key}:\text{Meal}]_o)$$

Since \mathbf{Order} is a union of HMaps, we structurally recur on the first case of update (Figure 5.11), which preserves π . Each initial recursion hits the first HMap case, since there is some τ such that $\mathcal{M}[k] = \tau$ and \mathcal{E} accepts partial maps \mathcal{P} .

To demonstrate, `:lunch` meals are handled by the first HMap case and update to

$$(\mathbf{HMap}^{\mathcal{P}} \mathcal{M}[(\mathbf{Val}:\text{Meal}) \mapsto \sigma'] \{\})$$

where $\sigma' = \text{update}((\mathbf{Val}:\text{lunch}), (\mathbf{Val}:\text{combo}), \epsilon)$ and

$$\mathcal{M} = \{(\mathbf{Val}:\text{Meal}) \mapsto (\mathbf{Val}:\text{lunch}), (\mathbf{Val}:\text{desserts}) \mapsto \mathbf{N}\}.$$

σ' updates to \perp via the penultimate update case, because $\text{restrict}((\mathbf{Val}:\text{lunch}), (\mathbf{Val}:\text{combo})) = \perp$ by the first restrict case. The same happens to `:dinner` meals, leaving just the `:combo` HMap.

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

We describe the other update cases. The first \mathbf{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 \mathbf{nil} .

CHAPTER 6

Metatheory

We prove type soundness following Tobin-Hochstadt and Felleisen [75]. Our model is extended to include errors `err` and a *wrong* value, and we prove well-typed programs do not go wrong; this is therefore a stronger theorem than proved by Tobin-Hochstadt and Felleisen [75]. Errors behave like Java exceptions—they can be thrown and propagate “upwards” in the evaluation rules (`err` rules are deferred to the appendix).

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.

ASSUMPTION 6.1 (JVM_{new}). *If $\forall i. v_i = C_i \{\overrightarrow{fld_j : v_j}\}$ or $v_i = \text{nil}$ and v_i is consistent with ρ then either*

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

ASSUMPTION 6.2 ($\text{JVM}_{\text{getstatic}}$). *If $v_1 = C_1 \{fld : v_f, \overrightarrow{fld_l : v_l}\}$, then either*

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

ASSUMPTION 6.3 ($\text{JVM}_{\text{invokestatic}}$). *If $v_1 = C_1 \{\overrightarrow{fld_l : v_l}\}$, $\forall i. v_i = C_i \{\overrightarrow{fld_j : v_j}\}$ or $v_i = \text{nil}$ then either*

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

For readability we define logical truth in Clojure.

DEFINITION 6.1 (TrueVal). $\text{TrueVal}(v)$ iff $v \neq \text{false}$ and $v \neq \text{nil}$.

DEFINITION 6.2 (FalseVal). $\text{FalseVal}(v)$ iff $v = \text{false}$ or $v = \text{nil}$.

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

DEFINITION 6.3 (Consistent with). v is consistent with ρ iff

- $\forall [\rho_1, \lambda x^\sigma . e]_c$ in v , if $\vdash [\rho_1, \lambda x^\sigma . e]_c : \tau ; \text{tt} \mid \text{ff} ; \emptyset$, and $\forall o'$ in τ , either $o' = \emptyset$, or $o' = \pi'(x)$, or $\rho(o') = \rho_1(o')$.

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

LEMMA 6.1. If $\Gamma \vdash e' \Rightarrow e : \tau ; \psi_+ \mid \psi_- ; o, \rho \models \Gamma, \rho$ 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 $\text{TrueVal}(v)$ and $\rho \models \psi_+$ or $\text{FalseVal}(v)$ and $\rho \models \psi_-$,
 - (3) $\vdash v : \tau ; \psi'_+ \mid \psi'_- ; o'$ for some ψ'_+, ψ'_- and o' , and
 - (4) v is consistent with ρ , or
- $\rho \vdash e \Downarrow \text{err}$.

THEOREM 6.1 (Type soundness for λ_{TC}). If $\Gamma \vdash e' \Rightarrow e : \tau ; \psi_+ \mid \psi_- ; o$ and $\rho \vdash e \Downarrow v$ then $\vdash v : \tau ; \psi'_+ \mid \psi'_- ; o'$ for some ψ'_+, ψ'_- and o' .

THEOREM 6.2 (Well-typed programs don't go wrong).

If $\vdash e' \Rightarrow e : \tau ; \psi_+ \mid \psi_- ; o$ then $\not\vdash e \Downarrow \text{wrong}$.

CHAPTER 7

Experience

Typed Clojure is implemented as `core.typed` [7], which has seen wide usage.

7.1. Implementation

`core.typed` provides preliminary integration with the Clojure compilation pipeline, primarily to resolve Java interoperability.

The `core.typed` implementation extends this paper in several key areas to handle checking real Clojure code, including an implementation of Typed Racket’s variable-arity polymorphism [74], and support for other Clojure idioms like datatypes and protocols. There is no integration with Java Generics, so only Java 1.4-style erased types are “trusted” by `core.typed`. Casts are needed to recover the discarded information, which—for collections—are then tracked via Clojure’s universal sequence interface [42].

7.2. Evaluation

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

To evaluate this hypothesis, we analyzed two existing `core.typed` code bases, one from the open-source community, and one from a company that uses `core.typed` in production. For our data gathering, we instrumented the `core.typed` type checker to record how often various features were used (summarized in Figure 7.1).

feeds2imap. `feeds2imap`¹ is an open source library written in Typed Clojure. It provides an RSS reader using the *javax.mail* framework.

¹<https://github.com/frenchy64/feeds2imap.clj>

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

FIGURE 7.1. Typed Clojure Features used in Practice

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 existing *javax.mail* functionality.

CircleCI. CircleCI [19] provides continuous integration services built with a mixture of open- and closed-source software. Typed Clojure was used at CircleCI in production systems for two years [21], maintaining 87 namespaces and 19,000 lines of code, an experience we summarise in Section 7.3.

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 overridden to return non-nil, and 1 method overridden to accept nil—suggesting that `core.typed` disallowing `nil` as a method argument by default is justified.

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

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

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

7.3. Further challenges

After a 2 year trial, the second case study decided to disabled type checking [20]. They were supportive of the fundamental ideas presented in this paper, but primarily cited issues with the

checker implementation in practice and would reconsider type checking if they were resolved. This is also supported by Figure 7.1, where 78% of `def` expressions are unchecked.

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

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

Unsupported idioms While the current set of features is vital to checking Clojure code, there is still much work to do. For example, common Clojure functions are often too polymorphic for the current implementation or theory to account for. The post-mortem [20] contains more details.

CHAPTER 8

Conclusion

Optional type systems must be designed with close attention to the language that they are intended to work for. We have therefore designed Typed Clojure, an optionally-typed version of Clojure, with a type system that works with a wide variety of distinctive Clojure idioms and features. Although based on the foundation of Typed Racket’s occurrence typing approach, Typed Clojure both extends the fundamental control-flow based reasoning as well as applying it to handle seemingly unrelated features such as multi-methods. In addition, Typed Clojure supports crucial features such as heterogeneous maps and Java interoperability while integrating these features into the core type system. Not only are each of these features important in isolation to Clojure and Typed Clojure programmers, but they must fit together smoothly to ensure that existing untyped 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, is suitable for typechecking a 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.

Our empirical analysis of existing Typed Clojure programs bears out our design choices. Multi-methods, 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.

Part II

Automatic Annotations for Typed Clojure

CHAPTER 9

Abstract

We present a semi-automated workflow for porting untyped programs to annotation-driven optional type systems. Unlike previous work, we infer useful types for recursive heterogeneous entities that have “ad-hoc” representations as plain data structures like maps, vectors, and sequences.

Our workflow starts by using dynamic analysis to collect samples from program execution via test suites or examples. Then, initial type annotations are inferred by combining observations across different parts of the program. Finally, the programmer uses the type system as a feedback loop to tweak the provided annotations until they type check.

Since inferring perfect annotations is usually undecidable and dynamic analysis is necessarily incomplete, the key to our approach is generating close-enough annotations that are easy to manipulate to their final form by following static type error messages. We explain our philosophy behind achieving this along with a formal model of the automated stages of our workflow, featuring maps as the primary “ad-hoc” data representation.

We report on using our workflow to convert real untyped Clojure programs to type check with Typed Clojure, which both feature extensive support for ad-hoc data representations. First, we visually inspect the initial annotations for conformance to our philosophy. Second, we quantify the kinds of manual changes needed to amend them. Third, we verify the initial annotations are meaningfully underprecise by enforcing them at runtime.

We find that the kinds of changes needed are usually straightforward operations on the initial annotations, leading to a substantial reduction in the effort required to port such programs.

CHAPTER 10

Introduction

Consider the exercise of counting binary tree nodes using JavaScript. With a class-based tree representation, we naturally add a method to each kind of node like so.

```
class Node { nodes() { return 1 + this.left.nodes() + this.right.nodes(); } }
class Leaf { nodes() { return 1; } }
new Node(new Leaf(1), new Leaf(2)).nodes(); //=> 3 (constructors implicit)
```

An alternative “ad-hoc” representation uses plain JavaScript Objects with explicit tags, which is less extensible but simpler. Then, the method becomes a recursive function that explicitly takes a tree as input.

```
function nodes(t) { switch t.op {
    case "node": return 1 + nodes(t.left) + nodes(t.right);
    case "leaf": return 1; } }
nodes({op: "node", left:{op: "leaf", val: 1}, right:{op: "leaf", val: 2}})//=>3
```

Now, consider the problem of inferring type annotations for these programs. The class-based representation is idiomatic to popular dynamic languages like JavaScript and Python, and so many existing solutions support it. For example, TypeWiz [72] uses dynamic analysis to generate the following TypeScript annotations from the above example execution of `nodes`.

```
class Node { public left: Leaf; public right: Leaf; ... }
class Leaf { public val: number; ... }
```

The intuition behind inferring such a type is straightforward. For example, an instance of `Leaf` was observed in `Node`’s `left` field, and so the nominal type `Leaf` is used for its annotation.

The second “ad-hoc” style of programming seems peculiar in JavaScript, Python, and, indeed, object-oriented style in general. Correspondingly, existing state-of-the-art automatic annotation

tools are not designed to support them. There are several ways to trivially handle such cases. Some enumerate the tree representation “verbatim” in a union, like TypeWiz [72].

```
function nodes(t: {left: {op: string, val: number}, op: string,
                 right: {op: string, val: number}}
              | {op: string, val: number}) ...
```

Others “discard” most (or all) structure, like Typette [38] and PyType [33] for Python.

```
def nodes(t: Dict[(Sequence, object)]) -> int: ... # Typette
def nodes(t) -> int: ... # PyType
```

Each annotation is clearly insufficient to meaningfully check both the function definition and valid usages. To show a desirable annotation for the “ad-hoc” program, we port it to Clojure [44], where it enjoys full support from the built-in runtime verification library `clojure.spec` and primary optional type system Typed Clojure [8].

```
(defn nodes [t] (case (:op t)
                  :node (+ 1 (nodes (:left t)) (nodes (:right t)))
                  :leaf 1))
(nodes {:op :node, :left {:op :leaf, :val 1}, :right {:op :leaf, :val 2}}) ;=>3
```

Making this style viable requires a harmony of language features, in particular to support programming with functions and immutable values, but none of which comes at the expense of object-orientation. Clojure is hosted on the Java Virtual Machine and has full interoperability with Java objects and classes—even Clojure’s core design embraces object-orientation by exposing a collection of Java interfaces to create new kinds of data structures. The `{k v ...}` syntax creates a persistent and immutable Hash Array Mapped Trie [5], which can be efficiently manipulated by dozens of built-in functions. The leading colon syntax like `:op` creates an interned *keyword*, which are ideal for map keys for their fast equality checks, and also look themselves up in maps when used as functions (e.g., `(:op t)` is like JavaScript’s `t.op`). *Multimethods* regain the extensibility we lost when abandoning methods, like the following.

```
(defmulti nodes-mm :op)
(defmethod nodes-mm :node [t] (+ 1 (nodes-mm (:left t)) (nodes-mm (:right t))))
(defmethod nodes-mm :leaf [t] 1)
```

On the type system side, Typed Clojure supports a variety of heterogeneous types, in particular for maps, along with occurrence typing [75] to follow local control flow. Many key features come together to represent our “ad-hoc” binary tree as the following type.

```
(defalias Tree
  (U '{:op ':node, :left Tree, :right Tree}
     '{:op ':leaf, :val Int}))
```

The `defalias` form introduces an equi-recursive type alias `Tree`, `U` a union type, `'{:kw Type ...}` for heterogeneous keyword map types, and `' :node` for keyword singleton types. With the following function annotation, Typed Clojure can intelligently type check the definition and usages of `nodes`.

```
(ann nodes [Tree -> Int])
```

This (manually written) Typed Clojure annotation involving `Tree` is significantly different from TypeWiz’s “verbatim” annotation for `nodes`. First, it is recursive, and so supports trees of arbitrary depth (TypeWiz’s annotation supports trees of height < 3). Second, it uses singleton types `' :leaf` and `' :node` to distinguish each case (TypeWiz upcasts `"leaf"` and `"node"` to `string`). Third, the tree type is factored out under a name to enhance readability and reusability. On the other end of the spectrum, the “discarding” annotations of Typette and PyType are too imprecise to use meaningfully (they include trees of arbitrary depth, but also many other values).

The challenge we overcome in this research is to automatically generate annotations like Typed Clojure’s `Tree`, in such a way that the ease of manual amendment is only mildly reduced by unresolvable ambiguities and incomplete data collection.

CHAPTER 11

Overview

We demonstrate our approach by synthesizing a Typed Clojure annotation for `nodes`. The following presentation is somewhat loose to keep from being bogged down by details—interested readers may follow the pointers to subsequent sections where they are made precise.

We use dynamic analysis to observe the execution of functions, so we give an explicit test suite for `nodes`.

```
(def t1 {:op :node, :left {:op :leaf, :val 1}, :right {:op :leaf, :val 2}})
(deftest nodes-test (is (= (nodes t1) 3)))
```

The first step is the instrumentation phase (formalized in Section 12.1), which monitors the inputs and outputs of `nodes` by redefining it to use the `track` function like so (where `<nodes-body>` begins the `case` expression of the original `nodes` definition):

```
(def nodes (fn [t'] (track ((fn [t] <nodes-body>) (track t' ['nodes :dom]))
                          ['nodes :rng])))
```

The `track` function (given later in Figure 12.2) takes a value to track and a *path* that represents its origin, and returns an instrumented value along with recording some runtime samples about the value. A path is represented as a vector of *path elements*, and describes the source of the value in question. For example, `(track 3 ['nodes :rng])` returns 3 and records the sample `Int['nodes :rng]` which says “`Int` was recorded at `nodes`’s range.” Running our test suite `nodes-test` with an instrumented `nodes` results in more samples like this, most which use the path element `{:key :kw}` which represents a map lookup on the `:kw` entry.

```
' :leaf['nodes :dom {:key :op}]      ' :node['nodes :dom {:key :op}]      ?['nodes :dom {:key :val}]
      ?['nodes :dom {:key :left}]      ?['nodes :dom {:key :right}]
' :leaf['nodes :dom {:key :left} {:key :op}]      ' :leaf['nodes :dom {:key :right} {:key :op}]
      Int['nodes :dom {:key :left} {:key :val}]      Int['nodes :dom {:key :right} {:key :val}]
```

Now, our task is to transform these samples into a readable and useful annotation. This is the function of the inference phase (formalized in Section 12.2), which is split into three passes: first it

generates a naive type from samples, then it combines types that occur syntactically near each other (“squash locally”), and then aggressively across different function annotations (“squash globally”).

The initial naive type generated from these samples resembles TypeWiz’s “verbatim” annotation given in Chapter 10, except the ? placeholder represents incomplete information about a path (this process is formalized as `toEnv` in Figure 12.3).

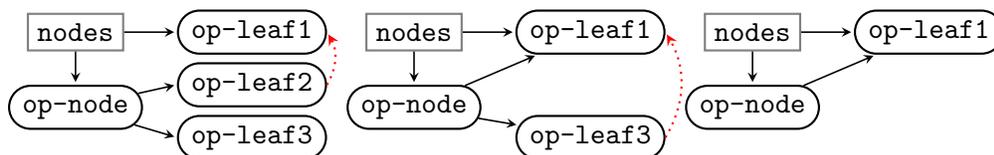
```
(ann nodes [(U '{:op ':leaf, :val ?}' '{:op ':node,
              :left '{:op ':leaf, :val Int}',
              :right '{:op ':leaf, :val Int}}) -> Int])
```

Next, the two “squashing” phases. The intuition behind both are based on seeing types as directed graphs, where vertices are type aliases, and an edge connects two vertices u and v if u is mentioned in v ’s type.

Local squashing (`squashLocal` in Figure 12.4) constructs such a graph by creating type aliases from map types using a post-order traversal of the original types. In this example, the previous annotations become:

```
(defalias op-leaf1 '{:op ':leaf, :val ?})
(defalias op-leaf2 '{:op ':leaf, :val Int})
(defalias op-leaf3 '{:op ':leaf, :val Int})
(defalias op-node '{:op ':node, :left op-leaf2, :right op-leaf3})
(ann nodes [(U op-leaf1 op-node) -> Int])
```

As a graph, this becomes the left-most graph below. The dotted edge from `op-leaf2` to `op-leaf1` signifies that they are to be merged, based on the similar structure of the types they point to.



After several merges (reading the graphs left-to-right), local squashing results in the following:

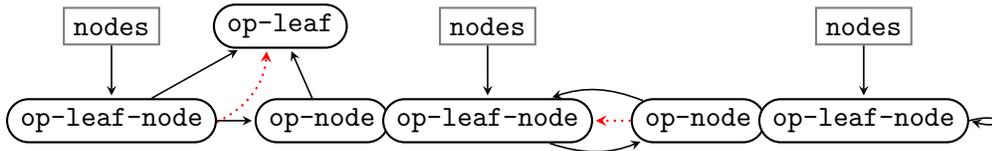
```
(defalias op-leaf '{:op ':leaf, :val Int})
(defalias op-node '{:op ':node, :left op-leaf, :right op-leaf})
(ann nodes [(U op-leaf op-node) -> Int])
```

All three duplications of the `':leaf` type in the naive annotation have been consolidated into their own name, with the ? placeholder for the `:val` entry being absorbed into `Int`.

Now, the global squashing phase (`squashGlobal` in Figure 12.5) proceeds similarly, except the notion of a vertex is expanded to also include *unions* of map types, calculated, again, with a post-order traversal of the types giving:

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

This creates `op-leaf-node`, giving the left-most graph below.



Now, type aliases are merged based on overlapping *sets* of top-level keysets and likely tags. Since `op-leaf` and `op-leaf-node` refer to maps with identical keysets (`:op` and `:val`) and whose likely tags agree (the `:op` entry is probably a tag, and they are both `':leaf`), they are merged and all occurrences of `op-leaf` are renamed to `op-leaf-node`, creating a *mutually recursive type* between the remaining aliases in the middle graph:

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

In the right-most graph, the aliases `op-node` and `op-leaf-node` are merged for similar reasons:

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

All that remains is to choose a recognizable name for the alias. Since all its top-level types seem to use the `:op` entry for tags, we choose the name `Op` and output the final annotation:

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

The rest of the porting workflow involves the programmer repeatedly type checking their code and gradually tweaking the generated annotations until they type check. It turns out that this

annotation immediately type checks the definition of `nodes` and all its valid usages, so we turn to a more complicated function `visit-leaf` to demonstrate a typical scenario.

```
(defn visit-leaf "Updates :leaf nodes in tree t with function f."
  [f t] (case (:op t)
          :node (assoc t :left (visit-leaf f (:left t))
                       :right (visit-leaf f (:right t)))
          :leaf (f t)))
```

This higher-order function uses `assoc` to associate new children as it recurses down a given tree to update leaf nodes with the provided function. The following test simply increments the leaf values of the previously-defined `t1`.

```
(deftest visit-leaf-test
  (is (= (visit-leaf (fn [leaf] (assoc leaf :val (inc (:val leaf)))) t1)
        {:op :node, :left {:op :leaf, :val 2}, :right {:op :leaf, :val 3}})))
```

Running this test under instrumentation yields some interesting runtime samples whose calculation is made efficient by space-efficient tracking (Section 14.1), which ensures a function is not repeatedly tracked unnecessarily. The following two samples demonstrate how to handle multiple arguments (by parameterizing the `:dom` path element) and higher-order functions (by nesting `:dom` or `:rng` path elements).

```
' :leaf['visit-leaf {:dom 1} {:key :op}]      ' :leaf['visit-leaf {:dom 0} {:dom 0} {:key :op}]
```

Here is our automatically generated initial annotation.

```
(defalias Op (U '{:op ':leaf, :val t/Int} '{:op ':node, :left Op, :right Op}))
(ann visit-leaf [[Op -> Any] Op -> Any])
```

Notice the surprising occurrences of `Any`. They originate from `?` placeholders due to the lazy tracking of maps (Section 14.2). Since `visit-leaf` does not traverse the results of `f`, nor does anything traverse `visit-leaf`'s results (hash-codes are used for equality checking) neither tracking is realized. Also notice the first argument of `visit-leaf` is underprecise. These could trigger type errors on usages of `visit-leaf`, so manual intervention is needed (highlighted). We factor out and use a new alias `Leaf` and replace occurrences of `Any` with `Op`.

```
(defalias Leaf '{:op ':leaf, :val Int})
(defalias Op (U Leaf '{:op ':node, :left Op, :right Op}))
(ann visit-leaf [[Leaf -> Op] Op -> Op])
```

We measure the success of our workflow by using it to type check real Clojure programs. Experiment 1 (Section 13.1) manually inspects a selection of inferred types. Experiment 2 (Section 13.2) classifies and quantifies the kinds of changes needed. Experiment 3 (Section 13.3) enforces initial annotations at runtime to ensure they are meaningfully underprecise.

CHAPTER 12

Formalism

We present λ_{track} , an untyped λ -calculus describing the essence of our approach to automatic annotations. We split our model into two phases: the collection phase `collect` that runs an instrumented program and collects observations, and an inference phase `infer` that derives type annotations from these observations that can be used to automatically annotate the program.

We define the top-level driver function `annotate` that connects both pieces. It says, given a program e and top-level variables \bar{x} to infer annotations for, return an annotation environment Δ with possible entries for \bar{x} based on observations from evaluating an instrumented e .

$$\begin{aligned} \text{annotate} &: e, \bar{x} \rightarrow \Delta \\ \text{annotate} &= \text{infer} \circ \text{collect} \end{aligned}$$

To contextualize the presentation of these phases, we begin a running example: inferring the type of a top-level function f , that takes a map and returns its `:a` entry, based on the following usage.

```
define f =  $\lambda m$ .(get m :a)
(f {:a 42}) => 42
```

Plugging this example into our driver function we get a candidate annotation for f :

$$\text{annotate}((f \text{ {:a 42}}), [f]) = \{f : \{\text{:a N}\} \rightarrow \text{N}\}$$

12.1. Collection phase

Now that we have a high-level picture of how these phases interact, we describe the syntax and semantics of λ_{track} , before presenting the details of `collect`. Figure 12.1 presents the syntax of λ_{track} . Values v consist of numbers n , Clojure-style keywords k , closures $[\lambda x.e, \rho]_c$, constants c , and keyword keyed hash maps $\{\overrightarrow{k} \ v\}$.

Expressions e consist of variables x , values, functions, maps, and function applications. The special form (`track` $e \ \pi$) observes e as related to path π . Paths π record the source of a runtime value with respect to a sequence of path elements l , always starting with a variable x , and are read

v	$::= n \mid k \mid [\lambda x.e, \rho]_c \mid \{\overline{k v}\} \mid c$	Values
e	$::= x \mid v \mid (\mathbf{track} \ e \ \pi) \mid \lambda x.e \mid \{\overline{e \bar{e}}\} \mid (e \ \bar{e})$	Expressions
ρ	$::= \{\overline{x \mapsto v}\}$	Runtime environments
l	$::= x \mid \mathbf{dom} \mid \mathbf{rng} \mid \mathbf{key}_m(k)$	Path Elements
π	$::= \bar{l}$	Paths
r	$::= \{\overline{\tau_\pi}\}$	Inference results
τ, σ	$::= \mathbf{N} \mid [\tau \rightarrow \tau] \mid (\mathbf{HMap}_o^m) \mid (\bigcup \tau \tau)$ $\mid a \mid k \mid \mathbf{K} \mid \top \mid (\mathbf{Map} \ \tau \ \tau) \mid ?$	Types
Γ	$::= \{\overline{x : \tau}\}$	Type environments
m, o	$::= \{\overline{k \ \tau}\}$	HMap entries
A	$::= \{\overline{a \mapsto \tau}\}$	Type alias environments
Δ	$::= (A, \Gamma)$	Annotation environments

FIGURE 12.1. Syntax of Terms, Types, Inference results, and Environments for λ_{track}

left-to-right. Other path elements are a function domain **dom**, a function range **rng**, and a map entry $\mathbf{key}_{\bar{k}_1}(k_2)$ which represents the result of looking up k_2 in a map with keyset \bar{k}_1 .

Inference results $\{\overline{\tau_\pi}\}$ are pairs of paths π and types τ that say the path π was observed to be type τ . Types τ are numbers \mathbf{N} , function types $[\tau \rightarrow \tau]$, ad-hoc union types $(\bigcup \tau \tau)$, type aliases a , and unknown type $?$ that represents a temporary lack of knowledge during the inference process. Heterogeneous keyword map types $\{\overline{k \ \tau}\}$ for now represent a series of required keyword entries—we will extend them to have optional entries in later phases.

The big-step operational semantics $\rho \vdash e \Downarrow v ; r$ (Figure 12.2) says under runtime environment ρ expression e evaluates to value v with inference results r . Most rules are standard, with extensions to correctly propagate inference results r . B-Track is the only interesting rule, which instruments its fully-evaluated argument with the **track** metafunction.

The metafunction $\mathbf{track}(v, \pi) = v' ; r$ (Figure 12.2) says if value v occurs at path π , then return a possibly-instrumented v' paired with inference results r that can be immediately derived from the knowledge that v occurs at path π . It has a case for every kind of value. The first three cases records the number input as type \mathbf{N} . The fourth case, for closures, returns a wrapped value resembling higher-order function contracts [28], but we track the domain and range rather than verify them. The remaining rules case, for maps, recursively tracks each map value, and returns a map with possibly wrapped values. Immediately accessible inference results are combined and returned. A specific rule for the empty map is needed because we otherwise only rely on recursive calls to **track** to gather inference results—in the empty case, we have no data to recur on.

Now we have sufficient pieces to describe the initial collection phase of our model. Given an expression e and variables \bar{x} to track, $\mathbf{instrument}(e, \bar{x}) = e'$ returns an instrumented expression e'

$$\begin{array}{c}
\text{B-TRACK} \\
\frac{\rho \vdash e \Downarrow v ; r \quad \text{track}(v, \pi) = v' ; r'}{\rho \vdash (\mathbf{track} \ e \ \pi) \Downarrow v' ; r \cup r'} \\
\\
\text{B-APP} \\
\frac{\rho \vdash e_1 \Downarrow [\lambda x.e, \rho]_c ; r_1 \quad \rho \vdash e_2 \Downarrow v ; r_2 \quad \rho'[x \mapsto v] \vdash e \Downarrow v' ; r_3}{\rho \vdash (e_1 \ e_2) \Downarrow v' ; \bigcup \bar{r}_i} \\
\\
\text{B-CLOS} \\
\rho \vdash \lambda x.e \Downarrow [\lambda x.e, \rho]_c ; \{\} \\
\\
\text{B-VAL} \\
\rho \vdash v \Downarrow v ; \{\} \\
\\
\text{B-VAR} \\
\rho \vdash x \Downarrow \rho(x) ; \{\} \\
\\
\text{B-DELTA} \\
\frac{\rho \vdash e \Downarrow c ; r_1 \quad \rho \vdash e' \Downarrow v ; r' \quad \delta(c, \bar{v}) = v' ; r_2}{\rho \vdash (e \ \bar{e}') \Downarrow v' ; r \cup r'} \\
\\
\text{track}(n, \pi) = n ; \{\mathbf{N}_\pi\} \\
\text{track}(k, \pi) = k ; \{\mathbf{K}_\pi\} \\
\text{track}(c, \pi) = c ; \{\} \\
\text{track}([\lambda x.e, \rho]_c, \pi) = [e', \rho]_c ; \{\} \\
\text{where } y \text{ is fresh,} \\
e' = \lambda y. (\mathbf{track} \ ((\lambda x.e) \ (\mathbf{track} \ y \ \pi :: [\mathbf{dom}]))) \\
\pi :: [\mathbf{rng}] \\
\\
\text{track}(\{\}, \pi) = \{\} ; \{\{\}\}_\pi \\
\text{track}(\{\bar{k}_1 \ \bar{k}_2 \ \bar{k} \ v\}, \pi) = \{\bar{k}_1 \ \bar{k}_2 \ \bar{k} \ v\} ; \bigcup r \\
\text{where } \text{track}(v, \pi :: [\mathbf{key}_{\{\bar{k}_1 \ \bar{k}_2 \ \bar{k} \ ?\}}(k)]) = v' ; r \\
\\
\delta(\text{assoc}, \{\bar{k} \ v\}, k', v') = \{\bar{k} \ v\}[k' \mapsto v'] ; \{\} \\
\delta(\text{get}, \{k \ v, \bar{k}' \ v'\}, k) = v ; \{\} \\
\delta(\text{dissoc}, \{k \ v, \bar{k}' \ v'\}, k) = \{\bar{k}' \ v'\} ; \{\}
\end{array}$$

FIGURE 12.2. Operational semantics, $\text{track}(v, \pi) = v ; r$ and constants

that tracked usages of \bar{x} . It is defined via capture-avoiding substitution:

$$\text{instrument}(e, \bar{x}) = e[\overline{(\mathbf{track} \ x \ [x])} / \bar{x}]$$

Then, the overall collection phase $\text{collect}(e, \bar{x}) = r$ says, given an expression e and variables \bar{x} to track, returns inference results r that are the results of evaluating e with instrumented occurrences of \bar{x} . It is defined as:

$$\text{collect}(e, \bar{x}) = r, \text{ where } \vdash \text{instrument}(e, \bar{x}) \Downarrow v ; r$$

For our running example of collecting for the program $(f \ \{\text{:a} \ 42\})$, we instrument the program by wrapping occurrences of f with \mathbf{track} with path $[f]$.

$$\text{instrument}((f \ \{\text{:a} \ 42\}), [f]) = ((\mathbf{track} \ f \ [f]) \ \{\text{:a} \ 42\})$$

Then we evaluate the instrumented program and derive two inference results (colored in red for readability):

$$\vdash ((\mathbf{track} \ f \ [f]) \ \{:\mathbf{a} \ 42\}) \Downarrow 42 ; \ \{\mathbf{N}_{[f, \mathbf{dom}, \mathbf{key}(:\mathbf{a})]}, \mathbf{N}_{[f, \mathbf{rng}]}\}$$

Here is the full derivation:

$$\begin{aligned} &=> ((\mathbf{track} \ f \ [f]) \ \{:\mathbf{a} \ 42\}) \\ &=> (\mathbf{track} \ (\mathbf{get} \ (\mathbf{track} \ \{:\mathbf{a} \ 42\} \ [f, \ \mathbf{dom}]) \ :\mathbf{a}) \ [f, \ \mathbf{rng}]) \\ &=> (\mathbf{track} \ (\mathbf{get} \ \{:\mathbf{a} \ 42\} ; \ \{\mathbf{N}_{[f, \ \mathbf{dom}, \ \mathbf{key}(:\mathbf{a})]}\} \ :\mathbf{a}) \ [f, \ \mathbf{rng}]) \\ &=> (\mathbf{track} \ 42 ; \ \{\mathbf{N}_{[f, \ \mathbf{dom}, \ \mathbf{key}(:\mathbf{a})]}\} \ [f, \ \mathbf{rng}]) \\ &=> 42 ; \ \{\mathbf{N}_{[f, \ \mathbf{dom}, \ \mathbf{key}(:\mathbf{a})]}, \ \mathbf{N}_{[f, \ \mathbf{rng}]}\} \end{aligned}$$

Notice that intermediate values can have inference results (colored) attached to them with a semi-colon, and the final value has inference results about both f 's domain and range.

12.2. Inference phase

After the collection phase, we have a collection of inference results r which can be passed to the metafunction $\mathbf{infer}(r) = \Delta$ to produce an annotation environment:

$$\begin{aligned} \mathbf{infer} &: r \rightarrow \Delta \\ \mathbf{infer} &= \mathbf{inferRec} \circ \mathbf{toEnv} \end{aligned}$$

The first pass $\mathbf{toEnv}(r) = \Gamma$ generates an initial type environment from inference results r . The second pass

$$\mathbf{squashLocal}(\Gamma) = \Delta'$$

creates individual type aliases for each HMap type in Γ and then merges aliases that both occur inside the same nested type into possibly recursive types. The third pass $\mathbf{squashGlobal}(\Delta) = \Delta'$ merges type aliases in Δ based on their similarity.

12.2.1. Pass 1: Generating initial type environment

The first pass is given in Figure 12.3. The entry point \mathbf{toEnv} folds over inference results to create an initial type environment via \mathbf{update} . This style is inspired by occurrence typing [75], from which we also borrow the concepts of paths into types.

$$\begin{array}{l}
\sqcup : \tau, \tau \rightarrow \tau \\
(\bigcup \bar{\sigma}) \sqcup \tau = (\bigcup \overline{\sigma \sqcup \tau}) \\
\tau \sqcup (\bigcup \bar{\sigma}) = (\bigcup \overline{\sigma \sqcup \tau}) \\
? \sqcup \tau = \tau \\
\tau \sqcup ? = \tau \\
[\tau_1 \rightarrow \sigma_1] \sqcup [\tau_2 \rightarrow \sigma_2] = [\tau_1 \sqcup \tau_2 \rightarrow \sigma_1 \sqcup \sigma_2] \\
(\text{HMap}_{o_1}^{m_1}) \sqcup (\text{HMap}_{o_2}^{m_2}) = (\text{HMap}_{o_1}^{m_1}) \sqcup^{\text{H}} (\text{HMap}_{o_2}^{m_2}), \\
\quad \frac{(k, k_i) \in m_i \Rightarrow k_{i-1} = k_i}{(k, k_i) \in m_i \Rightarrow k_{i-1} = k_i} \\
\tau \sqcup \sigma = (\bigcup \tau \sigma), \text{ otherwise}
\end{array}
\qquad
\begin{array}{l}
(\text{HMap}_{o_1}^{m_1}) \sqcup^{\text{H}} (\text{HMap}_{o_2}^{m_2}) = (\text{HMap}_o^m) \\
\text{where } \text{req} = \bigcup \overline{\text{dom}(m_i)} \\
\text{opt} = \bigcup \overline{\text{dom}(o_i)} \\
\overline{k^r} = \bigcap \overline{\text{dom}(m_i)} \setminus \text{opt} \\
\overline{k^o} = \text{opt} \cup (\text{req} \setminus \overline{k^r}) \\
m = \overline{\{k^r \sqcup m_i[k^r]\}} \\
o = \overline{\{k^o \sqcup m_i[k^o], o_i[k^o]\}}
\end{array}$$

$$\begin{array}{l}
\text{fold} : \forall \alpha, \beta. (\alpha, \beta \rightarrow \alpha), \alpha, \bar{\beta} \rightarrow \alpha \quad \text{update} : \Gamma, \tau_\pi \rightarrow \Gamma \\
\text{fold}(f, \mathbf{a}_0, \bar{\mathbf{b}}^n) = \mathbf{a}_n \quad \text{update}(\Gamma, \tau_{\pi::[\text{key}_{\{k^r \sigma\}}(k)]}) = \text{update}(\Gamma, \{\overline{k^r \sigma k \tau}\}_\pi) \\
\text{where } \bar{\mathbf{a}}_i = f(\mathbf{a}_{i-1}, \mathbf{b}_i)^{1 \leq i \leq n} \quad \text{update}(\Gamma, \tau_{\pi::[\text{dom}]}) = \text{update}(\Gamma, [\tau \rightarrow ?]_\pi) \\
\quad \text{update}(\Gamma, \tau_{\pi::[\text{rng}]}) = \text{update}(\Gamma, [? \rightarrow \tau]_\pi) \\
\text{toEnv} : r \rightarrow \Gamma \quad \text{update}(\Gamma[x \mapsto \sigma], \tau_{[x]}) = \Gamma[x \mapsto \tau \sqcup \sigma] \\
\text{toEnv}(r) = \text{fold}(\text{update}, \{\}, r) \quad \text{update}(\Gamma, \tau_{[x]}) = \Gamma[x \mapsto \tau]
\end{array}$$

FIGURE 12.3. Definition of $\text{toEnv}(r) = \Gamma$

We process paths right-to-left in `update`, building up types from leaves to root, before joining the fully constructed type with the existing type environment via \sqcup . The first case handles the **key** path element. The extra map of type information preserves both keyset information and any entries that might represent tags (populated by the final case of **track**, Figure 12.2). This information helps us avoid prematurely collapsing tagged maps, by the side condition of the `HMap` \sqcup case. The \sqcup^{H} metafunction aggressively combines two `HMaps`—required keys in both maps are joined and stay required, otherwise keys become optional.

The second and third `update` cases update the domain and range of a function type, respectively. The \sqcup case for function types joins covariantly on the domain to yield more useful annotations. For example, if a function accepts **N** and **K**, it will have type $[\mathbf{N} \rightarrow ?] \sqcup [\mathbf{K} \rightarrow ?] = [(\bigcup \mathbf{N} \mathbf{K}) \rightarrow ?]$.

Returning to our running example, we now want to convert our inference results

$$r = \{\mathbf{N}_{[f, \text{dom}, \text{key}(\cdot : \mathbf{a})]}, \mathbf{N}_{[f, \text{rng}]}\}.$$

into a type environment. Via `toEnv`(r), we start to trace `update`($\{\}, \mathbf{N}_{[f, \text{dom}, \text{key}(\cdot : \mathbf{a})]}$)

12.2.2. Pass 2: Squash locally

We now describe the algorithm for generating recursive type aliases. The first step `squashLocal` creates recursive types from directly nested types. It folds over each type in the type environment, first creating aliases with `aliasHMap`, and then attempting to merge these aliases by `squashAll`.

$$\begin{array}{l}
\text{aliasHMap} : \Delta, \tau \rightarrow (\Delta, \tau) \\
\text{aliasHMap}(\Delta, \tau) = \text{postwalk}(\Delta, \tau, f) \\
\text{where } f(\Delta, (\text{HMap}_{m_2}^{m_1})) = \text{reg}(\Delta, (\text{HMap}_{m_2}^{m_1})) \\
\quad f(\Delta, (\bigcup \bar{\tau})) = \text{reg}(\Delta, (\bigcup \text{resolve}(\tau))), \\
\quad \text{if } a \in \bar{\tau} \\
\quad f(\Delta, \tau) = (\Delta, \tau), \text{ otherwise} \\
\\
\text{aliases} : \tau \rightarrow \bar{a} \\
\text{aliases}(a) = [a] \\
\text{aliases}(\tau(\bar{\sigma})) = \bigcup \overline{\text{aliases}(\sigma)} \\
\\
\text{postwalk} : \Delta, \tau, (\Delta, \tau \rightarrow (\Delta, \tau)) \rightarrow (\Delta, \tau) \\
\text{postwalk}(\Delta_0, \tau(\bar{\sigma}^n), w) = w(\Delta_n, \tau(\bar{\sigma}')) \\
\text{where } (\Delta_i, \sigma'_i) = \text{postwalk}(\Delta_{i-1}, \sigma_i, w) \\
\\
\text{mergeAliases} : \Delta, \bar{a} \rightarrow \Delta \\
\text{mergeAliases}(\Delta, []) = \Delta \\
\text{mergeAliases}(\Delta, [a_1 \dots a_n]) = \Delta[\overline{a_i \mapsto a_1}][a_1 \mapsto \sigma] \\
\text{where } \sigma = \bigsqcup f(\text{resolve}(\Delta, a_i))[a_1/a_i] \\
\quad f(a') = (\bigcup), \text{ if } a' \in \bar{a} \\
\quad f((\bigcup \bar{\tau})) = (\bigcup f(\bar{\tau})) \\
\quad f(\tau) = \tau, \text{ otherwise} \\
\\
\text{squashLocal} : \Gamma \rightarrow \Delta \\
\text{squashLocal}(\Gamma) = \mathbf{fold}(h, (\{\}, \{\}), \Gamma) \\
\text{where } h(\Delta, x : \tau) = \Delta_2[x \mapsto \tau_2] \\
\quad \text{where } (\Delta_1, \tau_1) = \text{aliasHMap}(\Delta, \tau) \\
\quad \quad (\Delta_2, \tau_2) = \text{squashAll}(\Delta_1, \tau_1) \\
\\
\text{reg} : \Delta, \tau \rightarrow (\Delta, \tau) \\
\text{reg}(\Delta, \tau) = (\Delta[a \mapsto \tau], a), \text{ where } a \text{ is fresh} \\
\\
\text{resolve} : \Delta, \tau \rightarrow \tau \\
\text{resolve}(\Delta, a) = \text{resolve}(\Delta[a]) \\
\text{resolve}(\Delta, \tau) = \tau, \text{ otherwise} \\
\\
\text{squashAll} : \Delta, \tau \rightarrow \Delta \\
\text{squashAll}(\Delta_0, \tau) = \Delta_n \\
\text{where } \bar{a}^n = \text{aliases}(\tau) \\
\quad \Delta_i = \text{squash}(\Delta_{i-1}, [a_i], []) \\
\\
\text{squash} : \Delta, \bar{a}, \bar{a} \rightarrow \Delta \\
\text{squash}(\Delta, [], d) = \Delta \\
\text{squash}(\Delta, a_1 :: w, d) = \\
\quad \text{squash}(\Delta', w \cup \text{as}, d \cup \{a_1\}) \\
\text{where} \\
\quad \text{as} = \text{aliases}(\Delta[a_1]) \setminus d \\
\quad \text{ap} = d \setminus \{a_1\} \\
\quad f(\Delta, a_2) = \text{if } \neg \text{merge?}(\overline{\text{resolve}(\Delta, a)}), \\
\quad \quad \text{then } \Delta \\
\quad \quad \text{else } \text{mergeAliases}(\Delta, \bar{a}_i) \\
\quad \Delta' = \text{if } a \in d, \text{ then } \Delta, \\
\quad \quad \text{else } \text{fold}(f, \Delta, \text{ap} \cup \text{as}) \\
\\
\text{merge?} : \bar{\tau} \rightarrow \mathbf{Bool} \\
\text{merge?}((\text{HMap}_{o_i}^{m_i})) = \exists k. \overline{(k, k_i) \in m_i} \\
\text{merge?}(\bar{\tau}) = \mathbf{F}, \text{ otherwise}
\end{array}$$

FIGURE 12.4. Definition of $\text{squashLocal}(\Gamma) = \Delta$

A type is aliased by `aliasHMap` either if it is a union containing a `HMap`, or a `HMap` that is not a member of a union. While we will use the structure of `HMaps` to determine when to create a recursive type, keeping surrounding type information close to `HMaps` helps create more compact and readable recursive types. The implementation uses a post-order traversal via `postwalk`, which also threads an annotation environment as it applies the provided function.

Then, `squashAll` follows each alias a_i reachable from the type environment and attempts to merge it with any alias reachable from a_i . The `squash` function maintains a set of already visited aliases to avoid infinite loops.

The logic for merging aliases is contained in `mergeAliases`. Merging a_2 into a_1 involves mapping a_2 to a_1 and a_1 to the join of both definitions. Crucially, before joining, we rename occurrences of a_2 to a_1 . This avoids a linear increase in the width of union types, proportional to the number of

$$\begin{aligned}
& \text{req} : \Delta, a \rightarrow m \\
& \text{req}(\Delta, a) = \text{req}(\Delta, \Delta[a]) \\
& \text{req}(\Delta, (\text{HMap}_o^m)) = m \\
\\
& \text{squashHorizontally} : \Delta \rightarrow \Delta \\
& \text{squashHorizontally}(\Delta) = \\
& \quad \text{fold}(\text{mergeAliases}, \Delta, \text{groupSimilarReq}(\Delta)) \\
\\
& \text{squashGlobal} : \Delta \rightarrow \Delta \\
& \text{squashGlobal} = \\
& \quad \text{squashHorizontally} \circ \text{aliasSingleHMap} \\
\\
& \text{groupSimilarReq} : \Delta \rightarrow \bar{a} \\
& \text{groupSimilarReq}(\Delta) = [\bar{a} | \bar{k} \in \text{dom}(r), \bar{a} = \text{remDiffTag}(\text{similarReq}(\bar{k}))] \\
& \text{where} \\
& \quad r = \{(\bar{k}, \bar{a}) | (\text{HMap}_o^{\{\bar{k} \tau\}}) \in \text{rng}(\Delta[A]), \bar{a} = \text{matchingReq}(\bar{k})\} \\
& \quad \text{matchingReq}(\bar{k}) = [a | (a, (\text{HMap}_o^m)) \in \Delta] \\
& \quad \text{similarReq}(\bar{k}) = [a | \bar{k}'^n \subseteq \bar{k}^m, m - n \leq \text{thres}(m), a \in r[\bar{k}']] \\
& \quad \text{remDiffTag}(\bar{a}) = [a' | a' \in \bar{a}, \text{if } (k, k') \in \text{req}(\Delta, a') \text{ and } \bigvee (k, k'') \in \text{req}(\Delta, a) \text{ then } \bar{k}' = \bar{k}'']
\end{aligned}$$

FIGURE 12.5. Definition of $\text{squashGlobal}(\Delta) = \Delta'$

merged aliases. The running time of our algorithm is proportional to the width of union types (due to the quadratic combination of unions in the join function) and this optimization greatly helped the running time of several benchmarks. To avoid introducing infinite types, top-level references to other aliases we are merging with are erased with the helper f .

The `merge?` function determines whether two types are related enough to warrant being merged. We present our current implementation, which is simplistic, but is fast and effective in practice, but many variations are possible. Aliases are merged if they are all HMaps (not contained in unions), that contain a keyword `key` in common, with possibly disjoint mapped values. For example, our opening example has the `:op` key mapped to either `:leaf` or `:node`, and so aliases for each map would be merged. Notice again, however, the join operator does not collapse differently-tagged maps, so they will occur recursively in the resulting alias, but separated by union.

Even though this implementation of `merge?` does not directly utilize the aliased union types carefully created by `aliasHMap`, they still affect the final types. For example, squashing `T` in

```

(defalias T
  (U nil '{:op :node :left '{:op :leaf ...} ...}))

```

results in

```
(defalias T
  (U nil '{:op :node :left T ...} '{:op :leaf ...}))
```

rather than

```
(defalias T2 (U '{:op :node :left T ...}
               '{:op :leaf ...}))
(defalias T (U nil T2))
```

An alternative implementation of `merge?` we experimented with included computing sets of keysets for each alias, and merging if the keysets overlapped. This, and many of our early experimentations, required expensive computations of keyset combinations and traversals over them that could be emulated with cruder heuristics like the current implementation.

12.2.3. Pass 3: Squash globally

The final step combines aliases without restriction on whether they occur “together”. This step combines type information between different positions (such as in different arguments or functions) so that any deficiencies in unit testing coverage are massaged away.

The `squashGlobal` function is the entry point in this pass, and is similar in structure to the previous pass. It first creates aliases for each HMap via `aliasSingleHMap`. Then, HMap aliases are grouped and merged in `squashHorizontally`.

The `aliasSingleHMap` function first traverses the type environment to create HMap aliases via `singleHMap`, and binds the resulting environment as Δ' . Then, alias environment entries are updated with `f`, whose first case prevents re-aliasing a top-level HMap, before we call `singleHMap` (`singleHMap`'s second argument accepts both x and a). The $\tau(\bar{\sigma})$ syntax represents a type τ whose constructor takes types $\bar{\sigma}$.

After that, `squashHorizontally` creates groups of related aliases with `groupSimilarReq`. Each group contains HMap aliases whose required keysets are similar, but are never differently-tagged. The code creates a map `r` from keysets to groups of HMap aliases with that (required) keyset. Then, for every keyset \bar{k} , `similarReq` adds aliases to the group whose keysets are a subset of \bar{k} . The number of missing keys permitted is determined by `thres`, for which we do not provide a definition. Finally, `remDiffTag` removes differently-tagged HMaps from each group, and the groups are merged via `mergeAliases` as before.

12.2.4. *Implementation*

Further passes are used in the implementation. In particular, we trim unreachable aliases and remove aliases that simply point to another alias (like a_2 in `mergeAliases`) between each pass.

CHAPTER 13

Evaluation

We performed a quantitative evaluation of our workflow on several open source programs in three experiments. We ported five programs to Typed Clojure with our workflow, and merely generated types for one larger program we deemed too difficult to port, but features interesting data types.

Experiment 1 involves a manual inspection of the types from our automatic algorithm. We detail our experience in generating types for part of an industrial-grade compiler which we ultimately decided not to manually port to Typed Clojure. This was because it uses many programming idioms beyond Typed Clojure’s capabilities (those detailed as “Further Challenges” by [8]), and so the final part of the workflow mostly involves working around its shortcomings.

Experiment 2 studies the kinds of the manual changes needed to port our five programs to Typed Clojure, starting from the automatically generated annotations. Experiment 3 enforces the initially generated annotations for these programs at runtime to check they are meaningfully underprecise.

13.1. Experiment 1: Manual inspection

For the first experiment, we manually inspect the types automatically generated by our tool. We judge our tool’s ability to use recognizable names, favor compact annotations, and not overspecify types.

We take this opportunity to juxtapose some strengths and weaknesses of our tool by discussing a somewhat problematic benchmark, a namespace from the ClojureScript compiler called `cljs.compiler` (the code generation phase). We generate 448 lines of type annotations for the 1,776 line file, and present a sample of our tool’s output as Figure 13.1. We were unable to fully complete the porting to Typed Clojure due to type system limitations, but the annotations yielded by this benchmark are interesting nonetheless.

The compiler’s AST format is inferred as `Op` (lines 1-8) with 22 recursive references (like lines 5, 5, 7) and 14 cases distinguished by `:op` (like lines 3, 6, 7), 5 of which have optional entries (like lines 4-5). To improve inference time, only the code emission unit tests were exercised (299 lines

```

1 (defalias Op ; omitted some entries and 11 cases
2   (U (HMap :mandatory
3       {:op ':binding, :info (U NameShadowMap FnScopeFnSelfNameNsMap), ...}
4       :optional
5       {:env ColumnLineContextMap, :init Op, :shadow (U nil Op), ...}))
6   ' {:op ':const, :env HMap49305, ...}
7   ' {:op ':do, :env HMap49305, :ret Op, :statements (Vec Nothing), ...}
8   ...))
9 (defalias ColumnLineContextMap
10  (HMap :mandatory {:column Int, :line Int} :optional {:context ':expr}))
11 (defalias HMap49305 ; omitted some entries
12  (U nil
13   ' {:context ':statement, :column Int, ...}
14   ' {:context ':return, :column Int, ...}
15   (HMap :mandatory {:context ':expr, :column Int, ...} :optional {...})))
16 (ann emit [Op -> nil])
17 (ann emit-dot [Op -> nil])

```

FIGURE 13.1. Sample generated types for `cljs.compiler`.

containing 39 assertions) which normally take 40 seconds to run, from which we generated 448 lines of types and 517 lines of specs in 2.5 minutes on a 2011 MacBook Pro (16GB RAM, 2.4GHz i5), in part because of key optimizations discussed in Chapter 14.

The main function of the code generation phase is `emit`, which effectfully converts a map-based AST to JavaScript. The AST is created by functions in `cljs.analyzer`, a significantly larger 4,366 line Clojure file. Without inspecting `cljs.analyzer`, our tool annotates `emit` on line 16 with a recursive AST type `Op` (lines 1-8).

Similar to our opening example `nodes`, it uses the `:op` key to disambiguate between (16) cases, and has recursive references (`Op`). We just present the first 4 cases. The first case `':binding` has 4 required and 8 optional entries, whose `:info` and `:env` entries refer to other `HMap` type aliases generated by the tool.

An important question to address is “how accurate are these annotations?”. Unlike previous work in this area [2], we do not aim for soundness guarantees in our generated types. A significant contribution of our work is a tool that Clojure programmers can use to help learn about and specify their programs. In that spirit, we strive to generate annotations meeting more qualitative criteria. Each guideline by itself helps generate more useful annotations, and they combine in interesting ways help to make up for shortcomings.

Choose recognizable names. Assigning a good name for a type increases readability by succinctly conveying its purpose. Along those lines, a good name for the AST representation on lines 1-8 might be `AST` or `Expr`. However, these kinds of names can be very misleading when incorrect, so instead of guessing them, our tool takes a more consistent approach and generates *easily recognizable* names based on the type the name points to. Then, those with a passing familiarity with the data flowing through the program can quickly identify and rename them. For example,

- `Op` (lines 1-8) is chosen because `:op` is clearly the dispatch key (the `:op` entry is also helpfully placed as the first entry in each case to aid discoverability),
- `ColumnLineContextMap` (lines 9-10) enumerates the keys of the map type it points to,
- `NameShadowMap` and `FnScopeFnSelfNameNsMap` (line 3) similarly, and
- `HMap49305` (lines 11-15) shows how our tool fails to give names to certain combinations of types (we now discuss the severity of this particular situation).

A failure of `cljs.compiler`'s generated types was `HMap49305`. It clearly fails to be a recognizable name. However, all is not lost: the compactness and recognizable names of other adjacent annotations makes it plausible for a programmer with some knowledge of the AST representation to recover. In particular 13/14 cases in `Op` have entries from `:env` to `HMap49305`, (like lines 6 and 7), and the only exception (line 5) maps to `ColumnLineContextMap`. From this information the user can decide to combine these aliases.

Favor compact annotations. Literally translating runtime observations into annotations without compacting them leads to unmaintainable and impractical types resembling `TypeWiz`'s “verbatim” annotation for `nodes`. To avoid this, we use optional keys where possible, like line 10, infer recursive types like `Op`, and reuse type aliases in function annotations, like `emit` and `emit-dot` (lines 16, 17).

One remarkable success in the generated types was the automatic inference `Op` (lines 1-8) with 14 distinct cases, and other features described in Figure 13.1. Further investigation reveals that the compiler actually features 36 distinct AST nodes—unsurprisingly, 39 assertions was not sufficient test coverage to discover them all. However, because of the recognizable name and organization of `Op`, it's clear where to add the missing nodes if no further tests are available.

These processes of compacting annotations often makes them more general, which leads into our next goal.

Don't overspecify types. Poor test coverage can easily skew the results of dynamic analysis tools, so we choose to err on the side of generalizing types where possible. Our opening example

`nodes` is a good example of this—our inferred type is recursive, despite `nodes` only being tested with a tree of height 2. This has several benefits.

- We avoid exhausting the pool of easily recognizable names by generalizing types to communicate the general role of an argument or return position. For example, `emit-dot` (line 17) is annotated to take `Op`, but in reality accepts only a subset of `Op`. Programmers can combine the recognizability of `Op` with the suggestive name of `emit-dot` (the dot operator in Clojure handles host interoperability) to decide whether, for instance, to split `Op` into smaller type aliases or add type casts in the definition of `emit-dot` to please the type checker (some libraries require more casts than others to type check, as discussed in Section 13.2).
- Generated Clojure spec annotations (an extension discussed in Section 14.3) are more likely to accept valid input with specs enabled, even with incomplete unit tests (we enable generated specs on several libraries in Section 13.3).
- Our approach becomes more amenable to extensions improving the running time of runtime observation without significantly deteriorating annotation quality, like lazy tracking (Section 14.2).

Several instances of overspecification are evident, such as the `:statements` entry of a `:do` AST node being inferred as an always-empty vector (line 7). In some ways, this is useful information, showing that test coverage for `:do` nodes could be improved. To fix the annotation, we could rerun the tool with better tests. If no such test exists, we would have to fall back to reverse-engineering code to identify the correct type of `:statements`, which is `(Vec Op)`.

Finally, 19 functions in `cljs.compiler` are annotated to take or return `Op` (like lines 16, 17). This kind of alias reuse enables annotations to be relatively compact (only 16 type aliases are used by the 49 functions that were exercised).

13.2. Experiment 2: Changes needed to type check

We used our workflow to port the following open source Clojure programs to Typed Clojure.

startrek-clojure. A reimplementaion of a Star Trek text adventure game, created as a way to learn Clojure.

math.combinatorics. The core library for common combinatorial functions on collections, with implementations based on Knuth’s Art of Computer Programming, Volume 4.

fs. A Clojure wrapper library over common file-system operations.

Library	Lines of code	Lines of Generated Global/Local Types	Lines manually added/removed	Casts/Instantiations	Polymorphic annotation	Local annotation	Type System Workaround/no-check	Overprecise argument/return type	Uncalled function (bad test coverage)	Variable-arity/keyword arg type	Add occurrence typing annotation	Erase or upcast HVec annotation	Add missing case in defalias
startrek	166	133/3	70/41	5 / 0	0	2	13/1	1 / 2	5	1 / 0	0	0	0
math.comb	923	395/147	124/120	23 / 1	11	19	2 / 9	5 / 2	0	3 / 4	1	3	0
fs	588	157/1	119/86	50 / 0	0	2	3 / 11	4 / 9	4	2 / 0	0	0	0
data.json	528	168/9	94/125	6 / 0	0	2	4 / 5	11/7	5	0 / 20	0	0	0
mini.occ	530	49/1	46/26	7 / 0	0	2	5 / 2	4 / 2	6	0 / 0	0	1	5

FIGURE 13.2. Lines of generated annotations, git line diff for total manual changes to type check the program, and the kinds of manual changes.

data.json. A library for working with JSON.

mini.occ. A model of occurrence typing by an author of the current paper. It utilizes three mutually recursive ad-hoc structures to represent expressions, types, and propositions.

In this experiment, we first generated types with our algorithm by running the tests, then amended the program so that it type checks. Figure 13.2 summarizes our results. After the lines of code we generate types for, the next two columns show how many lines of types were generated and the lines manually changed, respectively. The latter is a git line diff between commits of the initial generated types and the final manually amended annotations. While an objectively fair measurement, it is not a good indication of the effort needed to port annotations (a 1 character changes on a line is represented by 1 line addition and 1 line deletion) The rest of the table enumerates the different kinds of changes needed and their frequency.

Uncalled functions. A function without tests receives a broad type annotation that must be amended. For example, the startrek-clojure game has several exit conditions, one of which is running out of time. Since the tests do not specifically call this function, nor play the game long enough to invoke this condition, no useful type is inferred.

`(ann game-over-out-of-time AnyFunction)`

In this case, minimal effort is needed to amend this type signature: the appropriate type alias already exists:

```
(defalias CurrentKlingonsCurrentSectorEnterpriseMap
  (HMap :mandatory
    {:current-klingons (Vec EnergySectorMap),
     :current-sector (Vec Int), ...}
    :optional {:lrs-history (Vec Str)})))
```

So we amend the signature as

```
(ann game-over-out-of-time
  [(Atom1 CurrentKlingonsCurrentSectorEnterpriseMap)
   -> Boolean])
```

Over-precision. Function types are often too restrictive due to insufficient unit tests.

There are several instances of this in `math.combinatorics`. The `all-different?` function takes a collection and returns true only if the collection contains distinct elements. As evidenced in the generated type, the tests exercise this functions with collections of integers, atoms, keywords, and characters.

```
(ann all-different?
  [(Coll (U Int (Atom1 Int) ':a ':b Character))
   -> Boolean])
```

In our experience, the union is very rarely a good candidate for a Typed Clojure type signature, so a useful heuristic to improve the generated types would be to upcast such unions to a more permissive type, like `Any`. When we performed that case study, we did not yet add that heuristic to our tool, so in this case, we manually amend the signature as

```
(ann all-different? [(Coll Any) -> Boolean])
```

Another example of overprecision is the generated type of `initial-perm-numbers` a helper function taking a *frequency map*—a hash map from values to the number of times they occur—which is the shape of the return value of the core `frequencies` function.

The generated type shows only a frequency map where the values are integers are exercised.

```
(ann initial-perm-numbers
  [(Map Int Int) -> (Coll Int)])
```

A more appropriate type instead takes `(Map Any Int)`. In many examples of overprecision, while the generated type might not be immediately useful to check programs, they serve as valuable starting points and also provide an interesting summary of test coverage.

Missing polymorphism. We do not attempt to infer polymorphic function types, so these amendments are expected. However, it is useful to compare the optimal types with our generated ones.

For example, the `remove-nth` function in `math.combinatorics` returns a functional delete operation on its argument. Here we can see the tests only exercise this function with collections of integers.

```
(ann remove-nth [(Coll Int) Int -> (Vec Int)])
```

However, the overall shape of the function is intact, and the manually amended type only requires a few keystrokes.

```
(ann remove-nth
  (All [a] [(Coll a) Int -> (Vec a)]))
```

Similarly, `iter-perm` could be polymorphic, but its type is generated as

```
(ann iter-perm [(Vec Int) -> (U nil (Vec Int))])
```

We decided this function actually works over any number, and bounded polymorphism was more appropriate, encoding the fact that the elements of the output collection are from the input collection.

```
(ann iter-perm
  (All [a]
    [(Vec (I a Num)) -> (U nil (Vec (I a Num)))]))
```

Missing argument counts. Often, variable argument functions are given very precise types. Our algorithm does not apply any heuristics to approximate variable arguments — instead we emit types that reflect only the arities that were called during the unit tests.

The `math.combinatorics` experiment contains a good example of this phenomenon in the type inferred for the `plus` helper function. From the generated type, we can see the tests exercise this function with 2, 6, and 7 arguments.

```
(ann plus (IFn [Int Int Int Int Int Int Int -> Int]
               [Int Int Int Int Int Int -> Int])
```

```
[Int Int -> Int]))
```

Instead, `plus` is actually variadic and works over any number of arguments. It is better annotated as the following, which is easy to guess based on both the annotated type and manually viewing the function implementation.

```
(ann plus [Int * -> Int])
```

A similar issue occurs with `mult`.

```
(ann mult [Int Int -> Int]) ;; generated  
(ann mult [Int * -> Int])   ;; amended
```

A similar issue is inferring keyword arguments. Clojure implements keyword arguments with normal variadic arguments. Notice the generated type for `lex-partitions-H`, which takes a fixed argument, followed by some optional integer keyword arguments.

```
(ann lex-partitions-H  
  (IFn [Int -> (Coll (Coll (Vec Int)))]  
    [Int ':min Int ':max Int  
      -> (Coll (Coll (Coll Int)))]))
```

While the arity of the generated type is too specific, we can conceivably use the type to help us write a better one.

```
(ann lex-partitions-H  
  [Int & :optional {:min Int :max Int}  
    -> (Coll (Coll (Coll Int)))]))
```

Weaknesses in Typed Clojure. We encountered several known weaknesses in Typed Clojure’s type system that we worked around. The most invasive change needed was in `startrek-clojure`, which strongly updated the global mutable configuration map on initial play. We instead initialized the map with a dummy value when it is first created.

Missing defalias cases. With insufficient test coverage, our tool can miss cases in a recursively defined type. In particular, `mini.occ` features three recursive types—for the representation of types `T`, propositions `P`, and expressions `E`. For `T`, three cases were missing, along with having to upcast the `:params` entry from the singleton vector `' [NameTypeMap]`. Two cases were missing from `E`. The manual changes are highlighted (`P` required no changes with five cases).

Library	LOC	Lines of specs	Recursive	Instance	Het. Map	Passed Tests?
startrek	166	25	0	10	0	Yes
math.comb	923	601	0	320	0	Yes
fs	588	543	0	215	0	Yes
data.json	528	401	0	174	0	No (1/79 failed)
mini.occ	530	131	3	25	15	Yes

FIGURE 13.3. Summary of the quantity and kinds of generated specs and whether they passed unit tests when enabled. The one failing test was related to pretty-printing JSON, and seems to be an artifact of our testing environment, as it still fails with all specs removed.

```
(defalias T
  (U '{:T ':not, :type T}
    '{:T ':refine, :name t/Sym, :prop P}
    '{:T ':union, :types (t/Set T)}
    '{:T ':false}
    '{:T ':fun,
      :params (t/Vec NameTypeMap),
      :return T}
    '{:T ':intersection, :types (Set T)}
    '{:T ':num}))

(defalias E
  (U '{:E ':add1}
    '{:E ':n?}
    '{:E ':app, :args (Vec E),
      :fun E}
    '{:E ':false}
    '{:E ':if, :else E,
      :test E, :then E}
    '{:E ':lambda, :arg Sym,
      :arg-type T, :body E}
    '{:E ':var, :name Sym}))
```

13.3. Experiment 3: Specs pass unit tests

Our final experiment uses our tool to generate specs (Section 14.3) instead of types. Specs are checked at runtime, so to verify the utility of generated specs, we enable spec checking while rerunning the unit tests that were used in the process of creating them.

At first this might seem like a trivial property, but it serves as a valuable test of our inference algorithm. The aggressive merging strategies to minimize aliases and maximize recognizability, while unsound transformations, are based on hypotheses about Clojure idioms and how Clojure programs are constructed. If, hypothetically, we generated singleton specs for numbers like we do for keywords and did not eventually upcast them to `number?`, the specs might be too strict to pass its unit tests. Some function specs also perform generative testing based on the argument and return types provided. If we collapse a spec too much and include it in such a spec, it might feed a function invalid input.

Thankfully, we avoid such pitfalls, and so our generated specs pass their tests for the benchmarks we tried. Figure 13.3 shows our preliminary results. All inferred specs pass the unit tests when enforced, which tells us they are at least well formed. We had some seemingly unrelated difficulty with a test in `data.json` which we explain in the caption. Since hundreds of invariants are checked—mostly “instance” checks that a value is of a particular class or interface—we can also be more confident that the specs are useful.

CHAPTER 14

Extensions

Two optimizations are crucial for practical implementations of the collection phase. First, space-efficient tracking efficiently handles a common case with higher-order functions where the same function is tracked at multiple paths. Second, instead of tracking a potentially large value by eagerly traversing it, lazy tracking offers a pay-as-you-go model by wrapping a value and only tracking subparts as they are accessed. Both were necessary to collect samples from the compiler implementation we instrumented for Experiment 1 (Section 13.1) because it used many higher-order functions and its AST representation can be quite large which made it intractable to eagerly traverse each time it got passed to one of dozens of functions.

14.1. Space-efficient tracking

To reduce the overhead of runtime tracking, we can borrow the concept of “space-efficient” contract checking from the gradual typing literature [40]. Instead of tracking just one path at once, a space-efficient implementation of track threads through a set of paths. When a tracked value flows into another tracked position, we extract the unwrapped value, and then our new tracked value tracks the paths that is the set of the old paths with the new path.

To model this, we introduce a new kind of value $[e, \rho]_{c\bar{\pi}}^v$ that tracks old value v as new value $[e, \rho]_c$ with the paths $\bar{\pi}$. Proxy expressions are introduced when tracking functions, where instead of just returning a new wrapped function, we return a proxy. We can think of function proxies as a normal function with some extra metadata, so we can reuse the existing semantics for function application—in fact we can support space-efficient function tracking just by extending **track**.

We present the extension in Figure 14.1. The first two **track** rules simply make inference results for each of the paths. The next rule says that a bare closure reduces to a proxy that tracks the domain and range of the closure with respect to the list of paths. Attached to the proxy is everything needed to extend it with more paths, which is the role of the final rule. It extracts the original closure from the proxy and creates a new proxy with updated paths via the previous rule.

$$\begin{aligned}
v ::= \dots & \mid \overline{\{k \ v\}}^{\overline{\{k \ \{m \ \overline{\pi}}\}}} && \text{Values} \\
\text{track}(\overline{\{k \ \overline{k'} \ \overline{k''} \ v\}}, \overline{\pi}) & = \overline{\{k \ \overline{k'} \ \overline{k''} \ v\}}^{\overline{\{k \ \overline{\mathbf{t}} \ \overline{k''} \ \overline{\mathbf{t}}\}}} ; \{\} \\
& \text{where } \mathbf{t} = \{\overline{\{k \ \overline{k'} \ \overline{k''} \ ?\}} \ \overline{\pi}\} \\
\text{track}(\overline{\{k \ v\}}^{\overline{\{k' \ \{m \ \overline{\pi'}\}}}}, \overline{\pi}) & = \overline{\{k \ v\}}^{\overline{\{k' \ \{m \ (\overline{\pi} \cup \overline{\pi'})\}}}}} ; \{\} \\
\delta(\text{assoc}, \overline{\{k \ v\}}^{\overline{\{k' \ \overline{\mathbf{t}}, \overline{k''} \ \overline{\mathbf{t}}\}}}, k', v') & = \overline{\{k \ v\}}[k' \mapsto v']^{\overline{\{k'' \ \overline{\mathbf{t}}\}}} ; \{\} \\
\delta(\text{assoc}, \overline{\{k \ v\}}^{\overline{\{k'' \ \overline{\mathbf{t}}\}}}, k', v') & = \overline{\{k \ v\}}[k' \mapsto v']^{\overline{\{k'' \ \overline{\mathbf{t}}\}}} ; \{\} \\
\delta(\text{get}, \{k \ v, \overline{k'} \ v'\}^{\overline{\{k \ \mathbf{t}, \overline{k''} \ \overline{\mathbf{t}}\}}}, k) & = \text{track}(v, \overline{\pi}) \\
& \text{where } \overline{\pi} = [\pi :: \mathbf{key}_m(k) \mid (m, \overline{\pi}) \in \mathbf{t}, \pi \in \overline{\pi}] \\
\delta(\text{get}, \{k \ v, \overline{k'} \ v'\}^{\overline{\{k'' \ \overline{\mathbf{t}}\}}}, k) & = v \\
\delta(\text{dissoc}, \{k \ v, \overline{k'} \ v'\}^{\overline{\{k \ \mathbf{t}, \overline{k''} \ \overline{\mathbf{t}}\}}}, k) & = \overline{\{k' \ v'\}}^{\overline{\{k'' \ \overline{\mathbf{t}}\}}} ; \{\} \\
\delta(\text{dissoc}, \{k \ v, \overline{k'} \ v'\}^{\overline{\{k'' \ \overline{\mathbf{t}}\}}}, k) & = \overline{\{k' \ v'\}}^{\overline{\{k'' \ \overline{\mathbf{t}}\}}} ; \{\}
\end{aligned}$$

FIGURE 14.2. Lazy tracking extensions (**changes**)

For example, say functions `f` and `g` operate on the same types of (deeply nested) arguments, and `f` has complete test coverage (but does not traverse all of its arguments), and `g` has incomplete test coverage (but fully traverses its arguments). Eagerly tracking `f` would give better inference results, but lazily tracking `g` is more efficient. Forcing several layers of tracking helps strike this balance, which our implementation exposes as a parameter.

This can be achieved in our formal system by adding fuel arguments to **track** that contain depth and breadth tracking limits, and defer to lazy tracking when out of fuel.

14.3. Automatic contracts with `clojure.spec`

While we originally designed our tool to generate Typed Clojure annotations, it also supports generating “specs” for `clojure.spec`, Clojure’s runtime verification system. There are key similarities between Typed Clojure and `clojure.spec`, such as extensive support for potentially-tagged keyword maps, however `spec` features a global registry of names via `s/def` and an explicit way to declare unions of maps with a common dispatch key in `s/multi-spec`. These require differences in both type and name generation.

The following generated specs correspond to the first `Op` case of Figure 13.1 (lines 2-5).

```
1 (defmulti op-multi-spec :op) ;dispatch on :op key
2 (defmethod op-multi-spec :binding ;match :binding
3   [_] ;s/keys matches keyword maps
4   (s/keys :req-un [::op ...] ;required keys
5           :opt-un [::column ...])) ;optional keys
6 (s/def ::op #{:js :let ...}) ;:op key maps to keywords
7 (s/def ::column int?) ;:column key maps to ints
8 ; register ::Op as union dispatching on :op entry
9 (s/def ::Op (s/multi-spec op-multi-spec :op))
10 ; emit's first argument :ast has spec ::Op
11 (s/fdef emit :args (s/cat :ast ::Op) :ret nil?)
```

CHAPTER 15

Conclusion

This paper shows how to generate recursive heterogeneous type annotations for untyped programs that use plain data. We use a novel algorithm to “squash” the observed structure of program values into named recursive types suitable for optional type systems, all without the assistance of record, structure, or class definitions. We test this approach on thousands of lines of Clojure code, optimizing generated annotations for programmer comprehensibility over soundness.

In our experience, our guidelines to automatically name, group, and reuse types yield insightful annotations for those with some familiarity with the original programs, even if the initial annotations are imprecise, incomplete, and always require some changes to type check. Most importantly, many of these changes will involve simply rearranging or changing parts of existing annotations, so programmers are no longer left alone with the daunting task of reverse-engineering such programs completely from scratch.

Part III

Typed Clojure Implementations

CHAPTER 16

Background

Clojure is a dialect of Lisp, and so supports metaprogramming via macros. This immediately poses an interesting problem for Clojure type systems: how do we check a macro call? Ideally, we don't want to require special typing rules for each macro, since that imposes additional burden on the programmer to define special rules for their own macros. On the other hand, sometimes it's helpful to write custom rules for customized error messages, or a higher-level specification for a macro's usage.

In this part we explore several solutions to this problem, from the standard approach of expanding macros to primitive forms before checking, to more involved solutions that allow extensible typing rules for each macro.

Several constraints guide us through our designs. There is a question of soundness: does what we actually check match up with the code being evaluated? There is a natural tension between soundness and user extensibility. Allowing custom rules for macros gives a kind of flexibility that makes it hard to relate type checking semantics with the running code—which is the whole idea behind a soundness result. On the other hand, expanding code before checking ensures we check the actual code being run. In all of these cases, wrappers that communicate information to the type system are needed, but they interact with evaluated code differently.

We also consider the experience of using these solutions. Error messages can be unrelated to the source problem if pre-expanding code, but we may miss actual errors by using a poorly written typing rule. We are interested in the difficulty of extending each system, including any additional annotation burden, additional knowledge needed to manage evaluation semantics in typing rules, and additional type system knowledge required to write typing rules. Finally, we also consider implications to type checking performance and amenability to iterative development.

The following chapters present several designs of Typed Clojure, their extensibility stories, and general implementation concerns for Clojure type system designers.

CHAPTER 17

Expand before checking

Typed Clojure’s initial design was inspired by Typed Racket, which checks Racket code by first expanding until it consists of only primitives, and then checking using fixed rules for each primitive. This chapter goes into this design in more detail, starting with our choice of analyzer and then how to handle extensibility.

17.1. Upfront Analysis with `tools.analyzer`

Instead of using Clojure’s compiler to analyze code, we opted to use `tools.analyzer`, a standalone nano-pass analyzer providing an idiomatic map-based AST format providing passes for hygienic transformations and Java reflection resolution.

Figure 17.1 demonstrates how Typed Clojure checks code using the pre-expansion approach. To simplify presentation we assume `tools.analyzer` uses only 2 passes. The first pass `analyze` creates a bare AST with no platform specific information. The second pass is composed of two tree traversals. The first is a pre-traversal `pre-passes` which is called before we visit the children of an AST node. The second is a post-traversal `post-passes` which is called after we visit the children of an AST node.

This arrangement is convenient as a type system implementer, insofar as there is a clean separation of concerns: the analyzer handles expansion and evaluation, while the type system merely checks. However, much contextual information is lost from the expansion process that is needed for checking. We now present how we surmount this challenge while still preserving the pre-expanded checking model.

17.2. Extensibility

Now that we have outlined how we use `tools.analyzer` to pre-expand code before type checking, we describe Typed Clojure’s approach to sharing information between the programs it checks and the type system. We deviate significantly from Typed Racket’s approach [22] mostly because of differences in compilation models between Clojure and Racket.

Time	(let [...]	(cond ...	(+ ...)))
0	analyze>		
1		analyze>	
2			analyze>
3			analyze<
4		analyze<	
5	analyze<		
6	pre-passes>		
7		pre-passes>	
8			pre-passes>
9			post-passes<
10		post-passes<	
11	post-passes<		
12	check>		
13		check>	
14			check>
15			check<
16		check<	
17	check<		

FIGURE 17.1. Illustrative control flow when using `tools.analyzer` to expand code via `analyze` and several passes, followed by Typed Clojure checking. The partial expression `(let [...] (cond ... (+ ...)))` was chosen since it has at least 3 levels of nesting. Many more levels will be revealed after expansion by `analyze`, which we do not picture. `>` and `<` indicate work done to a node before and after processing its children, respectively.

One constraint we must consider in Typed Clojure is that a “typed” Clojure program must evaluate unchanged under normal Clojure compilation. In Racket, we could instead specify the language under which a module is compiled using the `#lang` directive—this is Typed Racket’s approach. In Clojure, there is just one language and no built-in facilities to extend the compilation process, so Typed Clojure provides a suite of macros for communicating with the type system that users must explicitly load and use.

These macros come in several flavors:

- syntax-based communication to type checker,
- side-effectful communication to type checker, and
- wrappers for existing untyped macros.

We discuss each in the following sections.

```

(defmacro tc-ignore
  "Ignore forms in body during type checking"
  [& body]
  `(do :clojure.core.typed.special-form/special-form
       :clojure.core.typed/tc-ignore
       ~@(or body [nil])))

```

FIGURE 17.2. Public facing macro definition for `tc-ignore`.

```

(defmethod internal-special-form :clojure.core.typed/tc-ignore
  [expr expected]
  (tc-ignore/check-tc-ignore check-expr expr expected))

```

FIGURE 17.3. Registering a corresponding typing rule for `tc-ignore` via the `do-special-form` protocol.

17.2.1. *Syntax-based communication*

A simple macro provided by Typed Clojure that communicates to the checker via syntax is `tc-ignore`, which takes a number of forms, places them in a `do` form, and tells the checker to ignore the resulting form and assign it type `Any`.

Figure 17.2 shows the implementation of the `tc-ignore` macro. It demonstrates the `do-special-form` protocol: if the first member of a `do` is the keyword

```
:clojure.core.typed.special-form/special-form,
```

the following keyword names a special typing rule to use to check the entire form. A corresponding typing rule must then be registered with the type checker under this name, like in Figure 17.3.

Clojure’s compilation and runtime models make `do` statements an excellent candidate for the basis of an extensible syntax-based communication protocol. First, it naturally inherits the top-level characteristics of `do`, which is key to defining wrapper macros that operate at the top-level. A usage of `tc-ignore` that relies on this is demonstrated in Figure 17.5. Second, it avoids the need to pre-expand its arguments to attach information, or have special cases for particular arguments. On the other hand, a communication protocol based on attaching metadata properties would require pre-expanding arguments, since metadata is lost on macroexpansion, and in some cases would not be possible, since many common Clojure forms do not support metadata (such as keywords, numbers, and `nil`). Third, the information can be compiled away using standard techniques, since

```

(defmacro ann-form
  "Annotate a form with an expected type."
  [form ty]
  `(do :clojure.core.typed.special-form/special-form
       :clojure.core.typed/ann-form
       {:type '~ty}
       ~form))

```

FIGURE 17.4. The definition of `ann-form` shows how to communicate extra information to the type checker

```

(tc-ignore
 (defmacro reverse-app [a f] `(~f ~a))
 (reverse-app 1 inc)) ;=> 2

```

FIGURE 17.5. Example top-level usage of `tc-ignore` where the second form must expand after the first evaluates. It works because `tc-ignore` wraps only with `do`.

they are constant statements—extra information can be provided via a map of constant values placed after the typing rule name, as in the definition of `ann-form` (Figure 17.4).

While a strong choice, there are some downsides to basing our communication protocol on `do` statements. There is no guarantee the information will be compiled away at runtime, and thus may contribute to bloating the runtime. On the other hand, `tools.analyzer` must be carefully configured to not erase these constant values before Typed Clojure can access them.

Alternative `do`-based protocols could be similarly effective such as attaching metadata directly to the symbol `do` or list `(do ...)`. We felt embedding the information directly in programs had the best chance of forward-compatibility, since the interaction between metadata and compilation is not well documented and can be platform-dependent (in our experience ClojureScript has handled some cases differently, like evaluating metadata instead of simply quoting it as in Clojure).

17.2.2. *Side-effectful communication*

Racket has a sophisticated system for managing compile-time side effects to accompany its module system. Clojure does not have a module system, and instead relies on conventions and a simple compilation model to write effective programs.

The unit of compilation in Clojure is a top-level form. A top-level Clojure form is guaranteed to have all previous top-level forms fully expanded and evaluated before it is expanded and evaluated

itself. This blurs the lines between compile-time and runtime, compared to the distinct phases of Racket compilation.

When checking a file with Typed Clojure, we have similar guarantees: when checking a top-level form, we can depend on the fact that all previous top-level forms have been expanded, evaluated, and checked, and that the current form has been fully expanded.

Thus, we have a choice of (at least) three times to send side-effectful communication to the type checker: expansion-time, evaluation-time, and checking-time. Figure 17.6 shows the most frequently used side-effectful macro **ann**, which registers the type of a var in the global environment. It expands to code that uses internal function **ann***, which does the registering. This is a *evaluation-time* side effect, and we similarly perform most communication at this time. We now elaborate on why this is a good choice.

A previous implementation of Typed Clojure (which was used by CircleCI in Section 7.2) only collected top-level annotations from **ann** at checking-time. This forced Typed Clojure to recursively check other files just to collection annotations. We decided the natural behavior of rechecking a file would be to recheck its dependencies so, among other benefits, top-level annotations would be kept up-to-date. Unfortunately, the checker was much slower at evaluating files than the Clojure compiler, meaning iterative development was hampered. To fix this, we made checking of transitive file dependencies optional, and so dependencies containing top-level annotations would potentially only be evaluated by the Clojure compiler. Evaluation-time was then the natural time to collect these annotations.

A side-effect of this design choice is that it is no longer a sound idea to infer types for unannotated top-level bindings. In the aforementioned implementation, if the checker finds an unannotated top-level **def** like `(def a 1)`, it will update the global environment with the inferred type of the right-hand-side. Now that transitive dependencies are optionally checked, it is not guaranteed the checker will infer these annotations, and so more top-level annotations via **ann** are needed to recover consistent checking behavior. This unfortunately increases the annotation burden even more, however the rewards are great. We believe that Clojure programmers will enjoy the ability to rapidly recheck small parts of their code base, just like they are used to in untyped Clojure.

Now, we discuss the merits of collection at evaluation-time over expansion-time. We avoid side-effects at expansion-time because Clojure code can be evaluated in two ways: from the original source code in on-the-fly compilation mode, and from precompiled JVM bytecode in ahead-of-time compilation mode. In the latter, code is expanded ahead-of-time (potentially in a different

```

(defmacro ann
  "Register top-level var with type."
  [varsym typesyn]
  (let [qsym (qualify-in-current-ns varsym)
        opts (meta varsym)
        check? (not (:no-check opts))]
    `(tc-ignore (ann* `~qsym `~typesyn `~check? `~&form))))
(defn ann*
  "Internal use only. Use ann."
  [qsym typesyn check? form]
  ; omitted - registers `qsym` at type `typesym`
  )

```

FIGURE 17.6. Implementation of `ann`, which expands to code that registers types at evaluation-time.

environment) and thus expansion-time side-effects are lost. We applied the standard solution to this problem: remove the side-effect from the macro itself and move it to the evaluation of the code it expands into.

17.2.3. *Wrapper macros*

Several situations call for wrapper macros for existing untyped macros. In practice, this often means the type system author provides an alternative implementation for a macro, and the type system user replaces any usages of the original macro in type-checked code with the alternative implementation. Sometimes this choice is aesthetic, providing a prettier way to write annotations. For example, the `fn` wrapper enables writing annotations like `(fn [a :-Int] ...)` instead of the more verbose `(ann-form (fn [a] ...) [Int ->Any])`.

The more pressing need for wrapper macros when checking pre-expanded code is to manage complex expansions. Some macro expansions are too complex for Typed Clojure to reason about, so it becomes necessary to rewrite these expansions to be more palatable for the checker. For example, the `for` macro is a lazy sequence builder using a list-comprehension syntax—however it expands into local loops using local mutable state, which are problematic to check. The wrapper macro for `for` expands (and thus evaluates) similarly, but inserts user-provided type annotations strategically into the expansion so it more easily type checks.

The problem with this kind of wrapper macros is that large amounts of implementation code must be copied to preserve the original semantics. Instead of checking a higher-level specification of the

macro's behavior, we are tied closely to a particular implementation. This has the advantage of checking the actual code that gets evaluated, but unfortunately requires the type system writer to closely follow the original implementations (hampering both backwards- and forwards-compatibility with versions of the original macro). Furthermore, users not only must use wrapper macros where necessary, but also recognize when they are required—usually attempting to check a complex expansion yields an incomprehensible error as Typed Clojure fails to check it. It is rarely apparent that a wrapper macro is needed from such an error message.

CHAPTER 18

Interleaved expansion and checking

The previous chapter outlined a design for Typed Clojure that fully expands code before checking. We identified several problems with the user experience of Typed Clojure’s initial design, including bad error messages, and excessive copying of macro implementations for wrapper macros. Additionally, we identified several issues with `tools.analyzer` that we have not yet discussed.

First, `tools.analyzer`’s goals of being mostly platform-agnostic made analysis particularly slow, and so added an undesirable performance overhead to type checking. In particular, a copy of the global scope is maintained for every namespace. While it enables a convenient platform-agnostic API for symbol resolution, it comes at a performance cost since it must be updated (from scratch) frequently. Furthermore, some macroexpansion side effects are not (yet) recognized by the analyzer which means analysis sometimes deviates from Clojure compiler, an undesirable situation since Typed Clojure intends to model how code runs *outside* of type checking. Unfortunately, fixing some of these differences would require even more frequent costly updates.

Second, it is impractical to recover contextual information lost via analysis. This is both because `tools.analyzer` has no way of representing unanalyzed code (so there is no choice but to expand immediately), and because `tools.analyzer` uses at least 2 passes over the AST (so there is no obvious place to recover contextual information since pre-traversal passes run *after* the entire program has been expanded). For example, Figure 17.1 illustrates `tools.analyzer`’s control flow with just 2 traversals. Say at time 1 we wished to take advantage of the unexpanded `cond` form with a special rule (before it expands and contextual information is lost). In fact, `tools.analyzer` provides the extension point `macroexpand-1` for just this purpose, which allows the user to specify exactly how a form is expanded. Unfortunately, time 0 introduced local bindings that are unhygienic, and the hygienic transformation pass (required for checking because occurrence typing’s propositions do not recognize variable shadowing) happens at time 6 with `pre-passes`. So, there is no room for a checking rule for `cond` until time 13, well after the `cond` is expanded away.

Fortunately, `tools.analyzer`'s design and implementation is otherwise brilliant and innovative, and forms a great base to build a new Clojure analyzer better suited to help solve many of the aforementioned analysis and checking problems—we did exactly that in `core.typed.analyzer`.

18.1. Interleaved Analysis with `core.typed.analyzer`

To replace `tools.analyzer`, we built `core.typed.analyzer`. In this section, we describe how `core.typed.analyzer` works, and outline both the ideas we repurposed from `tools.analyzer` and those specific to `core.typed.analyzer`.

18.1.1. *Overview*

The main feature of `core.typed.analyzer` is the ability to stop and resume analysis at any point, while still supporting the essentials of a general-purpose Clojure analyzer. Supporting this requires several key innovations and restrictions over `tools.analyzer`. First, a new AST node type for partially expanded forms is needed to return a paused analysis. Second, the analyzer must have the ability to incrementally perform a small amount of analysis (on the order of expanding one macro) to provide fine-grained control over the AST. Third, all AST traversals must be fused into one traversal to minimize the bookkeeping needed to manage the AST.

To this end, `core.typed.analyzer` provides an API of 4 functions. First, `(unanalyzed form env)` creates an `:unanalyzed` AST node that pauses the analysis of `form` in local environment `env`. Second, `(analyze-outer ast)` analyzes the outermost form represented by `ast` further by roughly one macroexpansion if possible, otherwise it returns `ast`. Third, `(run-pre-passes ast)` and `(run-post-passes ast)` decorate `ast` with extra information, used before and after visiting its children, respectively.

To sample how it feels to use this API to implement a type checker, we now walk through checking `(let [...] (cond ... (+ ...)))` in Figure 18.1. To check the outermost `let`, we use `unanalyzed` to create an initial AST from a entire form at time 0. Then at time 1, the checker calls `analyze-outer` zero or more times, either until a special rule for partially expanded code is triggered or to a fixed point. Next at time 2 and 3 we decorate our AST node with `run-pre-passes` (adding hygienic bindings) before calling `check`. After checking its children during time 4-13, at time 14 and 15 we use `run-post-passes` to add the rest of the decorations (e.g., resolving interop reflection) before any final checks from `check`. The interleaving of operations using `core.typed.analyzer` is clear to see when compared to the same example using `tools.analyzer` (Figure 17.1).

Time	(let [...]	(cond ...	(+ ...))
0	unanalyzed>		
1	analyze-outer*		
2	run-pre-passes>		
3	check>		
4		analyze-outer*	
5		run-pre-passes>	
6		check>	
7			analyze-outer*
8			run-pre-passes>
9			check>
10			run-post-passes<
11			check<
12		run-post-passes<	
13		check<	
14	run-post-passes<		
15	check<		

FIGURE 18.1. Illustrative control flow for interleaved checking and analysis using `core.typed.analyzer`. * denotes zero or more calls.

Now with the interleaving analyzer, we can solve the problem we posed at the beginning of this chapter of wanting a custom typing rule for `cond`: we simply limit the number of expansions done via `analyze-outer` at time 4 before calling `check` (Figure 18.1). The call to `run-pre-passes` at time 2 will make any introduced `let` bindings hygienic, and so it's safe to reason about them with occurrence typing, and thus Typed Clojure.

18.1.2. Implementation

We now go into more detail about how `core.typed.analyzer` is implemented as a modification of `tools.analyzer` and the various tradeoffs that were chosen.

To support the requirement of `analyze-outer` performing as little analysis as possible, we converting the `analyze` function from a full AST traversal to a pre-traversal that only visits the current node. This mostly involved substituting recursive calls to `analyze-form` with `unanalyzed`, as we can see from porting the `parse-if` helper function in Figure 18.2.

Porting the nano-pass machinery was more involved, however we have a similar goal: passes must perform the minimum possible work so they can be easily composed as-needed. Thankfully, passes in `tools.analyzer` are written modularly, so we can straightforwardly pick a subset of them we need for `core.typed.analyzer`. To connect the passes, metadata declares dependencies on other

```

; tools.analyzer version
(defn parse-if
  "Convert a Clojure `(if <test> <then> <else>)` form to an AST."
  [_ test then else :as form] env]
  {:op      :if
   :form    form
   :env     env
   :test    (analyze-form test (assoc env :context :ctx/expr))
   :then    (analyze-form then env)
   :else    (analyze-form else env)
   :children [:test :then :else]})

; core.typed.analyzer version
(defn parse-if
  "Convert a Clojure `(if <test> <then> <else>)` form to an AST."
  [_ test then else :as form] env]
  {:op      :if
   :form    form
   :env     env
   :test    (unanalyzed test (assoc env :context :ctx/expr))
   :then    (unanalyzed then env)
   :else    (unanalyzed else env)
   :children [:test :then :else]})

```

FIGURE 18.2. Example of porting a `tools.analyzer` function to `core.typed.analyzer` using `unanalyzed` (differences highlighted in red).

```

(defn constant-lift
  "Like clojure.tools.analyzer.passes.constant-lifter/constant-lift but
  transforms also :var nodes where the var has :const in the metadata
  into :const nodes and preserves tag info"
  {:pass-info {:walk :post, :depends #{}},
   :after #{'elide-meta #'analyze-host-expr}}
  [ast]
  (merge (constant-lift* ast)
         (select-keys ast [:tag :o-tag :return-tag :arglists])))

```

FIGURE 18.3. Passes in `tools.analyzer` are defined as regular functions, with `:pass-info` metadata (red) declaring dependencies on other passes and tree walking strategy.

passes and the traversal strategy. We can see this in action for `constant-lift` (Figure 18.3), which is declared to be part of a post-traversal that must run after `elide-meta` and `analyze-host-expr`.

```

1 (defn unanalyzed
2   "Create an unanalyzed AST node from form and env"
3   [form env]
4   {:op :unanalyzed
5    :form form
6    :env env
7    ;; ::config will be inherited by whatever node
8    ;; this :unanalyzed node becomes when analyzed
9    ::config {}})
10
11 (defn analyze-outer
12   "If ast is :unanalyzed, call analyze-form on it, otherwise return ast"
13   [ast]
14   (case (:op ast)
15     :unanalyzed (assoc (analyze-form (:form ast) (:env ast))
16                        ::config (:config ast))
17     ast))

```

FIGURE 18.4. The initialization and propagation of `::config` (relevant parts **highlighted**)

A scheduler compiles the passes according to this metadata into as few traversals as possible. We reuse this setup of scheduled passes in `core.typed.analyzer`, with the restriction that all passes compile into one traversal. We could convert many existing pre- and post-traversal passes without much modification. Only the most crucial pass required much modification: the hygienic transformation pass `uniquify-locals`. It must be a pre-traversal in `core.typed.analyzer` (for reasons we have already discussed), and was modified from a full tree walk.

To help support `:unanalyzed` AST nodes, a `:clojure.core.typed.analyzer/config` entry (abbreviated `::config`) was added to all nodes to attach data that applies to AST nodes even after they are expanded. For example, a top-level expression is still top-level after it is expanded. The implementations of `unanalyzed` and `analyze-outer` in Figure 18.4 show their propagation—`unanalyzed` initializes `::config` on line 9, and `analyze-outer` propagates it on line 16 after further analysis.

Finally, we revised to platform-agnostic parts of the `tools.analyzer` API to allow better performance. Symbol and namespace resolution are now platform-dependent, which allows us to remove the global environment mirroring we identified as a performance issue at the beginning of this chapter. This added a slight burden to platform implementers of `core.typed.analyzer`—the JVM support added a dozen lines of code, although it took several revisions and testing to recover the original behavior.

```

1 (defn check
2   "Check an analyzed AST node has the expected type."
3   [expr expected]
4   (case (:op expr)
5     :if (let [ctest (check-expr (:test expr) <omitted>)]
6           <omitted>)
7     :lambda <omitted>
8     <omitted other cases>))
9 (defn check-expr
10  "Check an AST node has the expected type."
11  [expr expected]
12  (if (= :unanalyzed (:op expr))
13      (case <resolved-op-sym-for-expr>
14        clojure.core/cond (check-special-cond expr expected)
15        ; default case
16        (check-expr (analyze-outer expr) expected))
17      (run-post-passes
18        (check (run-pre-passes expr)
19              expected))))
20 (defn check-form
21  "Check a Clojure expression has the expected type"
22  [form expected]
23  (check-expr (unanalyzed form (empty-env))
24            expected))

```

FIGURE 18.5. The driver function `check-form` for a type system using `core.typed.analyzer`, which dispatches to a special typing rule for an unexpanded `cond` (red).

18.2. Extensibility in Interleaved checking

Now we present the most significant type system feature enabled by `core.typed.analyzer`: custom typing rules. We already hinted at how this support works in Figure 18.1—in this section we make that explicit with a small type system implementation.

We now present the sample type system in Figure 18.5. The main entry point is `check-form` (line 20), and we can check our running example has type `expected` with:

```

(check-form '(let [...] (cond ... (+ ...)))
           expected)

```

A pair of mutually recursive helpers assist the main driver: `check-expr` (line 9) handles the analysis machinery along with unanalyzed forms, and `check` which type checks an analyzed AST node.

Once `check-expr` has found a fully analyzed AST, it calls `check` (line 18) in between running the analyzer passes. Correspondingly, any recursive checking of children performed in `check` could trigger a special rule for unanalyzed forms, and so calls `check-expr` (for example, checking `:if`'s test on line 5).

Finally, custom typing rules are dispatched by `check-expr`—we have included an example dispatch to a `cond` rule on line 14. The `check-special-cond` function now has the ability to define a robust typing rule for `cond`: it has full access to both the unexpanded `cond` form and its hygienic type context.

This is a far cry from what was possible with `tools.analyzer`, and so `core.typed.analyzer` is a success in that light. However, with great power comes great responsibility: handing users the ability to control the order of analysis via custom typing rules requires careful planning in the face of compile-time side effects. The next chapter is dedicated to discussing this caveat.

Managing Analysis Side effects

To change Clojure’s order-of-macroexpansion is to change the semantics of Clojure—in theory. This chapter will give an overview of Clojure’s evaluation model so that the full implications of giving Typed Clojure users the responsibility to handle macroexpansion via custom typing rules becomes apparent. We also present how both `tools.analyzer` and `core.typed.analyzer` attempt to preserve these semantics. We will then compare our issues with those in other systems that allow typing rules.

19.1. Clojure’s Evaluation Model

In this section, we describe the subtleties of evaluating Clojure code. To evaluate a string of Clojure code, it is first parsed (via `read`) into a Clojure data representation and then macroexpanded until it consists of only language primitives. This is then compiled to JVM bytecode which is executed to produce the result of evaluation. Loading a file of Clojure code is mostly equivalent to evaluating each form in the file from top-to-bottom.

A form is given a special status when considered *top-level*: it will be completely evaluated before the next top-level form is expanded. Under evaluation, a form is considered top-level unless it is nested under another form. For example, `(query)` in `(cond (query) ...)` is not considered top-level, and the entire `cond` form is top-level (unless nested in a larger form). The exception to this rule is nesting under `do` expressions: arguments of a top-level `do` form inherits its top-level status. That is, in the top-level expression `(do (def a ...) (def b ...))`, `a` will be completely defined before `b` is expanded. This arrangement allows the expansion of one top-level form to depend on the evaluation (and thus expansion) of all preceding top-level forms.

As described above, Clojure is always compiled (it has no interpreter). Clojure offers two modes of compilation: on-the-fly and ahead-of-time. The main distinction is that on-the-fly mode discards the generated bytecode after executing it, whereas ahead-of-time mode both executes and saves the bytecode (as JVM `.class` files) for later execution. This is different from other Lisps like Chez Scheme [25] and Common Lisp [73], which has distinct semantics for interpreted and compiled

modes. In these languages, there is an implicit assumption that expressions are only compiled development machines, and so compilation mode in these languages skips the evaluation of certain expressions to avoid production-only side effects (e.g., initializing databases). Programmers must use `eval-when` to opt-in to different behavior. In contrast, Clojure evaluates all code during compilation (and Clojure is always compiled). Programmers rely on on Java-style `main` methods (invoked from the command line) to trigger initialization steps only applicable in production.

The most important consequence of Clojure’s ahead-of-time compilation is that macros are expanded in a different environment than the program is executed in, and thus state is not necessarily preserved between them. This is a well-known problem in most Lisps like Chez Scheme and Common Lisp—to work around it, Steele [73] suggests the convention of moving compile-time side effects into the code that the macro expands to. This way, the side effects are evaluation-time, and thus always visible in every mode of compilation. Clojure also recommends this convention—without it, it is possible to have accidental dependencies on expansion side-effects that only cause bugs under ahead-of-time compilation (usually performed only as the last step of software deployment). Racket’s module system, on the other hand, avoids these latent bugs [30] by erasing compile-time state before evaluation. This emulates the conditions of ahead-of-time compilation in Racket’s interpreted mode, at the cost of repeated module reinitializations.

19.2. Is order-of-expansion defined in Clojure?

Order of evaluation in Clojure is usually specified where it makes sense [41]. For example, invocations (`f arg*`) are evaluated left-to-right starting from `f`, whereas the order of evaluation for elements of unordered set literals `#{k*}` is undefined. On the other hand, the order of *expansion* is not addressed at all in the Clojure documentation. It would be extremely convenient for the writers and users of Typed Clojure to avoid micromanaging the order of expansion, and would make writing custom typing rules and other Typed Clojure extensions more viable. With those biases in mind, we now attempt to give a balanced account of expansion order in Clojure.

It is worth distinguishing between order of expansion of top-level forms and inner forms. Common Lisp asserts [73] that the order of macroexpansion for inner forms is unspecified. This gives flexibility not only to both platform implementors but also macro writers, because it grants macros the flexibility to expand their arguments, a pattern used by the Clojure core library `core.async` [43]. This seems to work in practice for `core.async` users without any special instruction or warnings. Also, `core.async` was designed by the same team that develops Clojure itself, so it gives us more

confidence that changing expansion order (by manually expanding a macro's arguments) is a sound choice.

Even more reason to doubt the importance of expansion order is its seeming lack of preservation across platforms for core macros. For instance, Clojure and ClojureScript share the same infrastructure for writing macros, but only share a subset of core macro *definitions*. That is, some macros are redefined in ClojureScript to cater to the JavaScript host—furthermore, some functions in Clojure are turned into macros in ClojureScript. While the order of evaluation must be preserved for compatibility with Clojure, it seems unlikely that any special measures were taken to preserve expansion order of arguments—especially for more complicated, platform-specific macros. In practice, however, most macros probably do preserve expansion-order: idiomatic macros do not expand their arguments and merely forward them to more primitive operators (often **do** or **let**) that have more consistent expansions across platforms. This might be coincidental, since we are not aware of any special effort to force this style, and might more be a consequence of following general Clojure idioms.

The popular general-purpose code analyzer `tools.analyzer` ignores particular expansion-time side-effects, without any apparent downsides. Specifically, changes to the current namespace are ignored during macroexpansion. This is a common *runtime* side-effect in Clojure, and is crucial for an analyzer to adhere to because analyzing a form in the wrong namespace is incorrect. Even so, we are not aware of any cases in practice in which this is a problem. Given that `tools.analyzer` is thoroughly tested and used in industry, it might then be reasonable to conclude that expansion-time side effects are rare. On the other hand, changing namespaces is a very specific side-effect whose conventions are perhaps not generalizable to other side-effects. Usually, changing namespaces is only triggered by the expansion of the `ns` macro, and `ns` is almost always used exactly once at the top of every Clojure file (to declare namespace dependencies). There could be other, more common expansion-time side-effects that are compatible with `tools.analyzer` that we are not aware of.

Relatedly, to help measure the practicality of an alternative design of Typed Clojure that expands macros multiple times, we talked to Clojure and Racket developers about repeated macro expansions. One Clojure developer felt that avoiding repeated expansion was important in Clojure, but could not show an example of real-world code that would fail in these circumstances. In contrast, a Racket researcher was quick to demonstrate complex assumptions in their macros that would be violated in these conditions (for example, global counters for identifying specific expansions).

Reflecting on these anecdotes, Clojure developers routinely reload code (REPL’s can take minutes to start and so encourage developers to leave them open for days) and so are guided to write code that can be re-evaluated in almost any order (after an initial load), encouraged further by the late-binding semantics of Clojure Vars. On the other hand, Racket’s module system is much stricter and reinitializes entire modules in their own sandboxes. Racket programmers can (and do) rely on these restrictions to support complex top-level invariants.

In both Clojure and Racket, it is idiomatic to avoid “double expansions” when writing macros by binding intermediate results to names in a macro’s expansion, usually for performance reasons. This also prevents double *evaluation*, an even more serious performance concern since expressions are run many more times than they are expanded. The issue of a third party (like Typed Clojure) expanding macros multiple times is tangentially related to this idiom, since the cost of double expansions must be paid. The main difference, at least in the design we proposed, was that the extra expansions by the third party would eventually be discarded and not evaluated, thus avoiding the cost of double evaluation.

It’s interesting to note that the backgrounds and daily obligations of each groups varied significantly, with the Racket programmers being mostly from academia (studying language extensibility) and the Clojure developers mostly from industry. While these opinions about repeated expansions are useful to help contextualize our larger discussion of order-of-expansion for inner forms, there are important details to take into account before prematurely linking the two subjects. We must avoid using the fact that Clojure programmers reload expressions out-of-order as direct evidence to support changing the expansion order of inner forms. This is because only *top-level* expressions are reloaded—intuitively, this does not change the expansion order of inner forms. We also note that reloading Clojure expressions can be notoriously buggy in certain circumstances, resulting in desynchronization between code on disk and code loaded into memory. Many disparate Clojure libraries and conventions have been developed to help manage these situations and there is no centralized solution. On the other hand, this problem is recognized and addressed by Racket’s module system. We do not want Typed Clojure contributing yet another source of desynchronization, which is the main reason behind this extended discussion.

19.3. Preserving evaluation order during type checking

Now that we have discussed some details of Clojure’s evaluation model, we demonstrate how to write a type checker that correctly preserves these semantics. Most of the details concern top-level

Time	(do	(defmacro mac [] 42)	(mac))
	analyze+eval> macroexpand-1*	analyze+eval> macroexpand-1* analyze+passes check eval analyze+eval<	analyze+eval> macroexpand-1* analyze+passes check eval analyze+eval<
	analyze+eval<		

FIGURE 19.1. Using `tools.analyzer` to interleave evaluation of top-level forms during the checking of `(do (defmacro mac [] 42) (mac))`.

expressions. In particular, a top-level `do` gives top-level status to its arguments, and that top-level forms must be completely evaluated in order.

The most interesting case to consider is a top-level `(do e1 e2)` form, where the *expansion* of `e2` depends on the *evaluation* of `e1`. For example,

```
1 (do (defmacro mac [] 42)
2     (mac))
```

defines the macro `mac` and then uses it to expand the macro call on the the last line (the final result is `42`). This will only execute correctly if top-level semantics of `do` are faithfully preserved. We will use this as a running example in our exploration of each analyzer.

19.3.1. *tools.analyzer*

An `analyze+eval` function handles top-level evaluation concerns in `tools.analyzer`, as Figure 19.1 demonstrates with our running example. The type checker’s main obligation is to provide a `check` function that checks a completely analyzed AST before it is evaluated. When passed to `analyze+eval`, it will only call `check` on the smallest top-level forms by expanding macros until a non-`do` expression is reached.

There are several other details that must be handled that led us to write our own `analyze+eval` variant for Typed Clojure—for example, expected type propagation and handling the `do`-special-form protocol. We do not discuss those details here, and hope that the original implementation of `analyze+eval` combined with the descriptions in this section should be sufficient to create other variants.

19.3.2. *core.typed.analyzer*

We have just described how to preserve evaluation order with `tools.analyzer`. The main distinction to keep in mind is that `tools.analyzer` controls analysis and checking for us, and so it handles its own internal bookkeeping to keep track of top-level forms. On the other hand, this section describes the same problem with `core.typed.analyzer`, which has comparatively very little control of when analysis is performed (since the provided `analyze-outer` function only performs a single expansion). Some other party—in this case, the type checker—must incorporate analysis into its main loop.

To handle this, `core.typed.analyzer` uses markers to distinguish between different kinds of top-level expressions. The first marker is performed by `mark-top-level` and is added to every kind of top-level expression. The second marker, `mark-eval-top-level`, further marks a top-level expression for evaluation. The `eval-top-level` function then evaluates AST nodes marked by `mark-eval-top-level` (with other corner cases we will describe later).

We use Figure 19.2 to demonstrate this communication with our running example. The entry point here is `check-top-level` at time 0, which has been enhanced to handle top-level evaluation order. To set up the rest of checking, at time 2 the outer `do` expression is marked with `mark-top-level`. Then, once `analyze-outer` determines the outer expression is a `do`, its children are then marked with `mark-top-level` (times 4 and 5).

For the `defmacro` form, let us assume for simplicity that it expands to a non-`do` expression. Under those circumstances, once `analyze-outer` at time 8 reaches a fixed point, the resulting AST node is marked for evaluation with `mark-eval-top-level` at time 9. Then (after it has been checked) at time 14 `eval-top-level` evaluates the `defmacro` expression (because it was marked with `mark-eval-top-level`). This repeats similarly for the final expression (`mac`).

There are a few notable cases to where `eval-top-level` does *not* trigger evaluation. The common case where evaluation is skipped is when an AST node has no top-level markings, like during time 11 where the main checking loop should not recursively evaluate expressions. A less-obvious

T	(do	(defmacro mac [] 42)	(mac))
0	check-top-level>		
1	unanalyzed		
2	mark-top-level		
3	analyze-outer*		
4		mark-top-level	
5			mark-top-level
6	run-pre-passes>		
7	check>		
8		analyze-outer*	
9		mark-eval-top-level	
10		run-pre-passes>	
11		check>	
12		run-post-passes<	
13		check<	
14		eval-top-level<	
15			analyze-outer*
16			mark-eval-top-level
17			run-pre-passes>
18			check>
19			run-post-passes<
20			check<
21			eval-top-level<
22	run-post-passes<		
23	check<		
24	eval-top-level<		
25	check-top-level<		

FIGURE 19.2. Using `core.typed.analyzer` to interleave checking with the evaluation of top-level forms.

case is a node marked with `mark-top-level`, but *not* `mark-eval-top-level`. For example, the outer-most `do` is only marked by `mark-top-level` (time 2). Evaluation is then skipped at time 24 because the top-level markers have been passed along to its children (at times 4 and 5) and so they themselves have been evaluated (at times 14 and 21). If we evaluated at time 24, it would result in an incorrect double-evaluation of the `defmacro` and `(mac)` expressions—hence evaluation is skipped by `eval-top-level` in this case.

Incorporating correct top-level evaluation into a `core.typed.analyzer`-based type checking loop is a mostly-straightforward extension, as demonstrated in Figure 19.3. The main differences are highlighted. First, `unanalyzed-top-level` combines `unanalyzed` and `mark-top-level` (times 1 and 2 in our previous discussion). Then, `eval-top-level` should be added as the final operation in the main checking loop. Finally, the handling of unanalyzed nodes (line 5) must carefully

```

1 (defn check-expr
2   "Check an AST node has the expected type."
3   [expr expected]
4   (if (= :unanalyzed (:op expr))
5       ...
6       (eval-top-level
7         (run-post-passes
8           (check (run-pre-passes expr)
9                 expected))))))
10 (defn check-top-level [form expected]
11   (check-expr (unanalyzed-top-level form (empty-env))
12              expected))

```

FIGURE 19.3. Handling top-level forms in a checker based on `core.typed.analyzer`

manage markers to prevent double evaluations. In particular, `eval-top-level` has a corner case to handle AST nodes that are not fully analyzed: nodes that are both `:unanalyzed` and are marked with `mark-top-level` are evaluated. The corresponding functions `unmark-top-level` and `unmark-eval-top-level` can be used to undo the markers if these semantics are undesirable for particular situations.

Part IV

Symbolic Closures

CHAPTER 20

Background

As is inevitable for an optional type system, there are many Clojure programs that Typed Clojure was not designed to type check. These programs contain Clojure idioms that are often either intentionally not supported by Typed Clojure’s initial design, or were introduced to Clojure at a later date. Regardless, programmers will inevitably want to use these features in their Typed Clojure programs—but crucially without breaking support for existing idioms. In this part, we explore what kinds of idioms are missing support in Typed Clojure, and propose solutions in the form of backwards-compatible extensions.

As we discussed in Part I, Typed Clojure’s initial design is strongly influenced by Typed Racket. In particular, Typed Clojure’s static semantics of combining local type inference and occurrence typing to check fully-expanded code comes directly from Typed Racket. This shared base is appropriate, given the similarities between the base Clojure and Racket languages. It is also effective, seamlessly handling many control flow idioms, capturing many polymorphic idioms, and often yielding predictable type error messages. However, there are important tradeoffs to consider in this design—in the following sections we introduce them and propose extensions to attempt to nullify their downsides.

20.1. Enhancing Local Type Inference

“Local Type Inference” [66] refers to the combination of bidirectional type propagation and local type argument synthesis. Concerning the limitations of local type inference, Hosoya and Pierce [45] isolate two drawbacks. The first is dealing with “hard-to-synthesize arguments”. To understand this, we must appreciate a key ingredient of local type inference called *bidirectional propagation*, which we use the example of type checking (`inc 42`) to demonstrate. If we have already checked `inc` to have type `[Int ->Int]`, we now have a choice of how to check the argument `42` is an `Int`. The first is to ascribe an expected type to `42` of `Int` and rely on bidirectional *checking mode* to ensure `42` has the correct type once we check it. The second is to infer the type of `42` (without an expected type) using bidirectional *synthesis mode*, and then ensure the inferred type is compatible

with `Int` after the fact. A useful analogy in terms of expressions is that checking mode propagates information outside-in, and synthesis mode propagates inside-out. A similar analogy in terms of a type derivation tree (that grows upwards) relates checking and synthesis modes to information being passed up and down the tree, respectively.

To best serve the purposes of local type inference, it is crucial to stay in bidirectional *checking* mode as much as possible. The “hard-to-synthesize arguments” problem occurs when type argument inference interferes with the ability to stay in checking mode, and thus forces the bidirectional propagator into synthesis mode for arguments that require checking mode. For example, to type check

```
(map (fn [x] (inc x)) [1 2 3]),
```

where `map` has type

```
(All [a b] [[a ->b] (Seqable a) ->(Seqable b)]),
```

we use type argument inference to determine how to instantiate type variables `a` and `b` based on `map`'s arguments. Unfortunately, to answer this question, the naive local type inference algorithm [66] uses synthesis mode to retrieve the argument types, and so checks `(fn [x] (inc x))` in synthesis mode. No information is propagated about the type of `x`, so this expression will fail to type check, demonstrating why functions are hard-to-synthesize.

The second drawback noted by Hosoya and Pierce are cases where there is no “best” type argument to infer. This occurs when there is not enough information available to determine how to instantiate a type such that the program has the best chance of type checking, and so it must be guessed. A representative case where this occurs is inferring the type of a reference from just its instantiation, such that optimal types are given to reads and writes. For example, the following code creates a Clojure Atom (a reference type) with initial value `nil`, writes 0 to the Atom, and then increments the Atom's value.

```
(let [r (atom nil)]
  (reset! r 0)
  (inc @r))
```

What type should `r` be assigned? From its initial binding, `(Atom nil)` seems appropriate, but the subsequent write would fail. Alternatively, assigning `(Atom Any)` would allow the write to succeed, but the the final read would fail because it expects `Int`. This demonstrates difficulties of the “no-best-type-argument” problem.

Hosoya and Pierce report unsatisfactory results in their attempts to fix these issues, in both the effectiveness and complexity in their solutions. They speculate that these difficulties might be better addressed at the language-design level—rather than algorithmically—in ways that keep the bidirectional propagator in checking mode. For the “no-best-type-argument” problem, we agree with this assessment, since addressing the problem mostly amounts to annotating all reference constructors. To this end, Typed Clojure offers several wrappers for common functions where this problem is common—the previous example might use the “typed” constructor

```
(t/atom :-(U nil Int), nil).
```

However, the “hard-to-synthesize arguments” problem is a deeper and more pervasive issue when checking Clojure code. We don’t have the luxury, desire, nor do we think it would be particularly successful to introduce new core idioms to Clojure, and so we attempt to solve this problem algorithmically.

Hosoya and Pierce outline the two main challenges that must be addressed to solve the “hard-to-synthesize arguments” problem. First, we must provide a strategy for identifying which arguments should be avoided. For instance, they provide a simple grammar for identifying hard-to-synthesize arguments, which includes (for Standard ML) unannotated functions and unqualified constructors. Second, an alternative (probably more complicated) algorithm for inferring type arguments is needed that also handles avoided arguments. Their experiments show that the naive approach does not suffice, and hint at the delicate handling needed to effectively maximize or minimize instantiated types to stay in checking mode. We will now use these challenges as a presentational framework to outline our own approach.

In our experience, the most common hard-to-synthesize expression in Clojure code is the function. Clojure’s large standard library of higher-order functions and encouragement of functional programming result in many usages of anonymous functions, which almost always require annotations to check with Typed Clojure. So, to answer Hosoya and Pierce’s first challenge, we avoid checking hard-to-synthesize function expressions by introducing a new function type: a *symbolic closure type*. A symbolic closure does not immediately check the function body. Instead, the function’s code along with its local type context is saved until enough information is available to check the function body in checking mode. We present more details about symbolic closures in Chapter 21.

Now that we have delayed the checking of hard-to-check arguments, Hosoya and Pierce’s second challenge calls for an enhanced type argument reconstruction algorithm to soundly handle them.

We delegate this challenge to future work, and study symbolic closures with an off-the-shelf type argument reconstruction algorithm.

Delayed checking for Unannotated Local Functions

Using bidirectional type checking, functions are hard-to-synthesize types for. Put another way, to check a function body successfully using only locally available information, types for its parameters are needed upfront. For top-level function definitions, this is not a problem in many optional type systems since top-level annotations would be provided for each function. However, for anonymous functions it's a different story. The original local type inference algorithm [66] lacks a synthesis rule for unannotated functions, instead relying on bidirectional propagation of types, but due to the prevalence of hard-to-synthesize anonymous functions in languages like JavaScript, Racket, and Clojure, optional type systems for the languages add their own rules.

Typed Racket and Typed Clojure implement a simple but sound strategy to check unannotated functions. The body of the function is checked in a type context where its parameters are of type **Any**, the **Top** type. This helps check functions that don't use their arguments, or only use them in positions that require type **Any**. For example, both `(fn [x] "a")` and `(fn [x] (str "x: " x))` synthesize to `[Any ->String]` in Typed Clojure. The downsides to this strategy are that unannotated functions are never inferred as polymorphic, and functions that use their arguments at types more specific than **Any** are common.

TypeScript [77], an optional type system for JavaScript, takes a similar approach, but instead of annotating parameters with TypeScript's least permissive type called **unknown**, by default it assigns parameters the unsound dynamic type **any**. In TypeScript, **any** can be implicitly cast to any other type, so the type checker will (unsoundly) allow any usage of unannotated arguments. If this behavior is unsatisfactory, the `noImplicitAny` flag removes special handling for unannotated functions altogether, and TypeScript will demand explicit annotations for all arguments.

In this chapter, we present an alternative approach to checking unannotated functions based on the insight that a function's body need only be type checked if and when it is called. For example, the program `(fn [x] (inc x))` cannot throw a runtime error because the function is never called, and so a type system may soundly treat the function body as unreachable code. On the other hand,

wrapping the same program in the invocation `((fn [x] (inc x)) nil)` makes the runtime error possible, and so a sound static type system must flag the error.

Exploiting this insight in the context of a bidirectional type checker using local type inference requires many considerations. First, we must decide in which situations is it desirable to delay checking a function. Second, we must identify the information that must be saved in order to delay checking a function, and then choose a suitable format for packaging that information. Third, we must identify how a function is deemed “reachable”, and then which component of the type system is responsible for checking a function body. Fourth, it is desirable to identify and handle the ways in which infinite loops are possible, such as the checking of a delayed function triggering another delayed function to check, which triggers another delayed check, ad nauseam. Fifth, we must determine how delayed functions interact with polymorphic types during type argument reconstruction.

We address all these considerations in the following sections, except for the final one, which we delegate to future work.

21.1. Overview

In this section, we explore some of the implications that come with delayed checks for local functions, by example. To explain the essence of the challenges we actually address, we avoid polymorphic functions in this section and restrict ourselves to non-recursive monomorphic functions.

First, let `inc` be of type `[Int ->Int]`. The following, then, is well typed because `1` is an `Int`.

```
(inc 1)
```

Using the standard bidirectional application type rule, `inc` is checked first, followed by `1`. However, eta-expanding the operator does not behave as nicely.

```
((fn [x] (inc x)) 1)
```

Like usual, the standard application rule checks the function first. However, there is no annotation for `x`, so the function body will fail to check. This is unfortunate, especially in a type system that claims to be “bidirectional”, since the information that `x` is an `Int` is adjacent to the function in the form of an argument. One strategy to alleviate this problem is to always check arguments first [83]. However, that nullifies the ability for the operator to propagate information to its arguments, whose advantages are exploited to good effect in Colored Local Type Inference [62]

```

1 (let [f (let [y 1]
2         (fn [x] (+ x y)))]
3   (let [y nil]
4     (f 1)))

```

FIGURE 21.1. This example evaluates to 2 with lexically scoped variables.

We combine both flavors by keeping the standard operator-first checking order but delay the checking of unannotated functions. Then, an additional application rule handles applications of unannotated functions to force their checking. So in this case, the checking of `(inc x)` is delayed until the argument 1 is inferred as `Int`, after which this information is used to check `(inc x)` in the extended type context where `x : Int`.

We could imagine hard-coding a type rule that manually delays direct applications of unannotated functions until after checking its arguments. However, that does not generalize to more complicated examples. Take the following illustrative code, identical the previous example, except the function is let-bound as `f`.

```

1 (let [f (fn [x] (inc x))]
2   (f 1))

```

Instead of following the brittle strategy of creating yet-another special rule to delay checking let-bound functions, we generalize the idea. We make a delayed function check a first-class concept in our type-system by creating a new type for it. Roughly, `f` would have a delayed function type—introduced by a type rule for unannotated functions—and `(f 1)` would force a check for the delayed function—by an application rule that handles delayed function *types* (not syntax-driven).

Now we must decide what a delayed function type consists of. Clearly, the *code* of the function must be preserved until it is checked, otherwise the application rule would have nothing to work with. We note that our static semantics of saving the code of a function to check later is analogous to the runtime strategy of evaluating a function as *closure*, and using beta-reduction to extract the original function from the closure and apply it to its arguments.

The trick in maintaining lexical scope during beta-reduction for closures is to apply the function under the *function definition's* environment, instead of the application site's. For example, Figure 21.1 evaluates to 2 because the occurrence of `y` on line 2 is bound to 1 by line 1. If we used the local environment at the application site (line 4), `y` would be bound on line 3 to `nil`, and would throw a runtime error.

The crucial insight is that the same trick applies to *checking* delayed function types, except at the *type-level*. Specifically, the occurrence of `y` on line 2 must be checked as type `Int` (from line 1), and not type `nil` (from line 3). So, a delayed function type pairs a function’s code with the type environment at the function definition site. This strongly resembles a “type-level” closure that is reduced symbolically, and so we call this new type a *symbolic closure*.

We can use symbolic closures to inline higher-order-function definitions. In the following example, `app` would normally need a higher-order or polymorphic annotation to handle the application on the final line. Instead, with symbolic closures, type checking reduces in a few steps to simply checking `(inc x)` where `x : Int`.

```
(let [f (fn [x] (inc x))
      app (fn [g y] (g y))]
  (app f 1))
```

As alluded to in the previous section, we must identify all type system components who are responsible for checking symbolic closures, and ensure they perform their obligations correctly. The following example uses a higher-order function `app-int` to increment the value 1. Since `app-int` is annotated, it will be checked by the standard application rule. However, its first argument will be delayed as a symbolic closure—now we must identify who is responsible for checking it.

```
(ann app-int [[Int -> Int] Int -> Int])
(defn app-int [f x] (f x))
...
(app-int (fn [x] (inc x)) 1)
```

The type signature of `app-int`, clearly says that its first argument may be called with an `Int`. Therefore, to maintain soundness, applications of `app-int` must ensure its first argument accepts `Int`. The standard application type rule uses subtyping to ensure provided arguments are compatible with the formal parameter types of the operator. To handle symbolic closures, we preserve the standard application rule and instead add a subtyping case for symbolic closures.

In this case, the subtyping relation would be asked to verify if “the symbolic closure type representing `(fn [x] (inc x))` is a subtype of `[Int ->Int]`”. This can be answered by checking the symbolic closure returns `Int` when `x` is type `Int`—and so this subtyping case delegates to checking if the symbolic closures inhabits the given type. The subtype relationship is true if the check succeeds without type error, otherwise it is false.

The correct “contravariant subtyping left-of-the-arrow” is naturally preserved. In this case, the left-of-the-arrow check is “**Int** is a subtype of **x**’s type”, and annotating **x** as **Int** turns this statement into the reflexively true “**Int** is a subtype of **Int**”. At a glance, it may seem that we are wasting the benefits of this contravariant rule—after all, it enables **x** to be *any* supertype of **Int**, such as **Num** or even **Any**. However, it is in our interest to propagate the most precise parameter types so then function bodies have the best chance to check without error. Since symbolic closures are designed to support rechecking their bodies at different argument types, a symbolic function can simply be rechecked with the less-precise types when it comes time to broaden its domain.

This scheme extends to subtyping with arbitrarily-nested function types. To demonstrate nesting to the right of an arrow, the following code sums 1 with itself via `curried-app-int`, which accepts a curried function of two arguments **f** and a number **x**, and provides **x** as both arguments to **f**.

```
(ann curried-app-int [[Int -> [Int -> Int]] Int -> Int])
(defn curried-app-int [f x] ((f x) x))
...
(curried-app-int (fn [y] (fn [x] (+ x y)))) 1)
```

The standard application rule will ensure “the symbolic closure of `(fn [y] (fn [x] (+ x y)))` is a subtype of `[Int ->[Int ->Int]]`”, which involves assuming `y : Int` and then checking the *code* `(fn [x] (+ x y))` at type `[Int ->Int]`—which just uses the standard function rule.

To demonstrate nesting to the left of an arrow, `app-inc` again computes `(inc 1)` in an even more convoluted way with `app-inc`—by accepting a function **f** that it passes both `inc` and its second argument to.

```
(ann app-inc [[[Int -> Int] Int -> Int] Int -> Int])
(defn app-inc [f x] (f inc x))
...
(app-inc (fn [g y] (g y)) 1)
```

Importantly, `app-inc`’s first argument has a function type to the left on an arrow, in particular `[Int ->Int]`. Under these conditions, subtyping asserts “the symbolic closure `(fn [g y] (g y))` is a subtype of `[[Int ->Int] Int ->Int]`” by assuming `g : [Int ->Int]` and `y : Int` and verifying that `(g y)` checks as **Int**—which is almost immediate by the standard application rule.

We leverage some syntactic restrictions to avoid the need for further subtyping cases for symbolic closures. First, symbolic closures cannot be annotated by the programmer, and can only be introduced by the “unannotated function” typing rule. Second (as discussed in Section 17.2.2), top-level

variables are not allowed to inherit the types of their initial values, and must be explicitly annotated. These restrictions ensure symbolic closures both cannot occur to the left of an arrow type, and cannot propagate beyond the top-level form it was defined in.

21.1.1. *Performance and error messages*

While useful, allowing the type system to perform beta-reduction requires careful planning: type checking time is now proportional to the running time of the program! Unsurprisingly, this makes type checking with a naive implementation of symbolic closures undecidable. Without intervention, the next program (an infinite loop using the y-combinator that computes `(inc (inc (inc ...)))`) would send the *type system* into an infinite loop.

```
(let [Y (fn [f]
         ((fn [g] (fn [x] (f (g g) x)))
          (fn [g] (fn [x] (f (g g) x))))))
      (let [compute (Y (fn [f x] (inc (f x))))]
        (compute 1)))]
```

To prevent such loops, we limit the number of symbolic reductions done at type-checking time. As a conservative solution to the halting problem, this limit will prematurely halt some programs that would otherwise fully reduce in a finite number of steps. For example, if we set the reduction limit to 5 in the following code, during the 6th reduction of `f` the type system will throw an error.

```
(let [f (fn [x] x)]
      (f (f (f (f (f 1))))))
```

In simple cases like these, the error message can guide the user to fixing the error. For example, the type system would suggest annotating `f` as `[Int ->Int]` (by collecting argument and return types as the program is reduced), which would cause the program to check successfully under the same conditions. For cases with more heterogeneous argument and return types—like the y-combinator—the error message would just note which function caused the reduction quota to be depleted.

As Wells [81] remarks, stopgap measures such as this to circumvent undecidable type inference algorithms negatively affect program portability. For example, a different reduction limit may cause a program to fail to type check that otherwise type checked in a previous version. We hope to learn reasonable defaults for the reduction limit by experience.

21.2. Formal model

We formalize a restriction of symbolic closure types by defining an explicitly typed internal language, providing an external languages that can omit annotations, and formulating a type inference algorithm based on symbolic closures to recover omitted annotations. This approach is similar to Local Type Inference [66] and Colored Local Type Inference [62], except where they utilize bidirectional type propagation to locally determine function parameter types, we instead use symbolic closures to propagate type information (since we have a synthesis rule for functions).

21.2.1. Internal Language

e, f	$::=$	$x \mid \lambda[\bar{\alpha}](x:\tau)e \mid f[\bar{\chi}](e) \mid e.x \mid \{\bar{x} \equiv e\}$	Terms
τ, σ, χ	$::=$	$\alpha \mid \top \mid \perp \mid \tau \xrightarrow{\bar{\alpha}} \sigma \mid \{\bar{x} : \bar{\tau}\}$	Types
Γ	$::=$	$\epsilon \mid \Gamma, x : \tau \mid \Gamma, \alpha$	Type Environments

FIGURE 21.2. Internal Language Syntax

Our internal language is based on System $F_{<}$, extended with records, and is functionally identical to that used to model Colored Local Type Inference [62], except our lambda terms require full (return) type annotations. Figure 21.2 shows the syntax for the internal language. Terms e and f range over variables x , explicitly typed polymorphic functions $\lambda[\bar{\alpha}](x:\tau)e$, function application with explicit type arguments $f[\bar{\chi}](e)$, record selectors $e.x$, and record constructors $\{\bar{x} \equiv e\}$. Types τ, σ , and χ are type variables α , top type \top , bottom type \perp , polymorphic function types $\tau \xrightarrow{\bar{\alpha}} \sigma$ (where bound type variables are enumerated over the arrow), and record types $\{\bar{x} : \bar{\tau}\}$. Type environments Γ consist of the empty environment ϵ , concatenation of variable typings “ $\Gamma, x : \tau$ ”, and concatenation of type variables “ Γ, α ”.

We assume different term and type variables are distinct, and treat terms and types that are equal up to alpha-renaming as equivalent. Record terms and types have unordered fields. We treat primitive types (like **String**) as free type variables.

Figure 21.3 presents the type system for the internal language $\Gamma \vdash e : \tau$, pronounced “ e is of type τ in context Γ .” I-VAR is the normal variable lookup rule. I-SEL selects a field already present in a record. I-SEL \perp allows selecting fields from \perp . I-ABS checks a function definition at its annotated type. I-APP checks a function application with explicit type arguments. I-APP \perp allows applying operators of type \perp . I-REC checks record constructors.

$\Gamma \vdash e : \tau$ e is of type τ in context Γ .	I-VAR $\Gamma \vdash x : \Gamma(x)$	I-SEL $\frac{\Gamma \vdash e : \{x_1 : \tau_1, \dots, x_i : \tau_i, \dots, x_n : \tau_n\}}{\Gamma \vdash e.x_i : \tau_i}$	I-SEL \perp $\frac{\Gamma \vdash e : \perp}{\Gamma \vdash e.x_i : \perp}$
I-ABS $\frac{\Gamma, \bar{\alpha}, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda[\bar{\alpha}](x:\tau)e : \forall \bar{\alpha}. [\tau \rightarrow \sigma]}$	I-APP $\frac{\Gamma \vdash f : \tau \xrightarrow{\bar{\alpha}} \sigma \quad \Gamma \vdash e : \tau' \quad \tau' \leq [\bar{\chi}/\bar{\alpha}]\tau}{\Gamma \vdash f[\bar{\chi}](e) : [\bar{\chi}/\bar{\alpha}]\sigma}$	I-APP \perp $\frac{\Gamma \vdash f : \perp \quad \Gamma \vdash e : \sigma}{\Gamma \vdash f[\bar{\chi}](e) : \perp}$	I-REC $\frac{}{\Gamma \vdash \{\bar{x} \equiv e\} : \{\bar{x} : \bar{\tau}\}}$

FIGURE 21.3. Internal language type system

$\tau \leq \sigma$ τ is a subtype of σ .	S-TVAR $\alpha \leq \alpha$	S-TOP $\tau \leq \top$	S-BOT $\perp \leq \tau$	S-REC $\frac{}{\{\bar{x} : \bar{\tau}, \bar{x}' : \bar{\tau}'\} \leq \{\bar{x} : \bar{\sigma}\}}$
S-FN $\frac{\sigma \leq \sigma' \quad \tau \leq \tau'}{\sigma' \xrightarrow{\bar{\alpha}} \tau \leq \sigma \xrightarrow{\bar{\alpha}} \tau'}$				

FIGURE 21.4. Internal language subtyping

$$e, f ::= \dots \mid \lambda(x)e \mid f(e) \quad \text{Terms}$$

FIGURE 21.5. External Language Syntax (extends Figure 21.2)

E-UAPP $\frac{\Gamma \vdash f : \tau \xrightarrow{\bar{\alpha}} \sigma \quad \Gamma \vdash e : \sigma' \quad \bar{\alpha} > 0 \quad \forall \bar{\chi}'. \left(\sigma' \leq [\bar{\chi}'/\bar{\alpha}]\tau \text{ implies } [\bar{\chi}/\bar{\alpha}]\sigma' \leq [\bar{\chi}'/\bar{\alpha}]\sigma' \right)}{\Gamma \vdash f(e) : [\bar{\chi}/\bar{\alpha}]\sigma}$	E-UAPP \perp $\frac{\Gamma \vdash f : \perp \quad \Gamma \vdash e : \sigma}{\Gamma \vdash f(e) : \tau}$	E-UABS $\frac{\Gamma, \bar{\alpha}, x : \tau \vdash e : \sigma \quad \bar{\alpha} \cap \text{tv}(e) = \emptyset}{\Gamma \vdash \lambda(x)e : \tau \xrightarrow{\bar{\alpha}} \sigma}$
---	--	--

FIGURE 21.6. External Language Specification (extends Figure 21.3)

Figure 21.4 presents the subtyping for the internal language $\tau \leq \sigma$, pronounced “ τ is a subtype of σ .” S-TVAR says type variables are subtypes of themselves. S-TOP and S-BOT establish \top and \perp as maximal and minimal types. S-REC says record types may forget or upcast their fields. S-FN relates types contravariantly to the left of an arrow and covariantly to the right.

21.2.2. External language

The syntax for the external language Figure 21.5 is a superset of the internal language, with unannotated functions $\lambda(x)e$, and “lightweight” applications with implicit type arguments $f(e)$.

The external language type system is (declaratively) specified in Figure 21.6 as a superset of the (algorithmic) internal language. E-UAPP says we must pick the most general type arguments when elaborating an application without explicit type arguments. This is identical to the corresponding rule in Local Type Inference. E-UABS says that an untyped function must type check at the interface chosen for its elaboration. Since we also infer type parameters, the rule also requires the type variables chosen must not capture type variables that occur free in the body of the function. A similar rule is included in Colored Local Type Inference, except colored types enforce that parameter types and type parameters only be inherited from its surrounding context. In contrast, our rule uses an oracle to synthesize both. This is because our type inference algorithm based on symbolic closures is not restricted to local reasoning.

$$\begin{array}{llll}
[\bar{\tau}^n \rightarrow \sigma] & \Leftrightarrow & [\{\overrightarrow{\text{arg}i : \tau_i}^{1 \leq i \leq n}\} \rightarrow \sigma] & \text{where } n \neq 1 \quad \text{Type abbreviations} \\
\lambda(\bar{x}^n)e & \Leftrightarrow & \lambda(x')[\overrightarrow{x'.\text{arg}i/x_i}^{1 \leq i \leq n}]e & \text{where } n \neq 1, x' \notin \text{fv}(e) \quad \text{Term abbreviations} \\
\lambda[\bar{\alpha}](\bar{x}^n : \bar{\tau}^n)e & \Leftrightarrow & \lambda[\bar{\alpha}](x' : \{\overrightarrow{\text{arg}i : \tau_i}^{1 \leq i \leq n}\}) & \\
& & [\overrightarrow{x'.\text{arg}i/x_i}^{1 \leq i \leq n}]e & \text{where } n \neq 1, x' \notin \text{fv}(e) \\
f(\bar{e}^n) & \Leftrightarrow & f(\{\overrightarrow{\text{arg}i = e_i}^{1 \leq i \leq n}\}) & \text{where } n \neq 1 \\
\text{let } \bar{x} = \bar{e} \text{ in } f & \Leftrightarrow & (\lambda(\bar{x})f)(\bar{e}) & \\
\lambda(\bar{x} : \bar{\tau})e & \Leftrightarrow & \lambda[](\bar{x} : \bar{\tau})e & \\
\lambda[\bar{\alpha}](\bar{x} : \bar{\tau}) : \sigma e & \Leftrightarrow & \lambda[\bar{\alpha}](\bar{x} : \bar{\tau})(e : \sigma) & \\
(e : \sigma) & \Leftrightarrow & (\lambda(x : \sigma)x)(e) & \\
\lambda(\bar{x} : \bar{\tau}) : \sigma e & \Leftrightarrow & \lambda[](\bar{x} : \bar{\tau}) : \sigma e &
\end{array}$$

FIGURE 21.7. External Language Syntax abbreviations

We use several syntax abbreviations, enumerated in Figure 21.7. The first four abbreviations use record terms and types to represent multi-parameter functions and applications. For example, the function type $[\tau_1, \tau_2 \rightarrow \sigma]$ stands for $[\{\text{arg}1 : \tau_1, \text{arg}2 : \tau_2\} \rightarrow \sigma]$, the function term $\lambda(x, y)e$ stands for $\lambda(x')[\overrightarrow{x'.\text{arg}1/x, x'.\text{arg}2/y}]e$ (where x' does not occur free in e) and the application term $f(e_1, e_2)$ stands for $f(\{\text{arg}1 = e_1, \text{arg}2 = e_2\})$. Symbolic closures permit the lambda-encoding of let, so $\text{let } x = e \text{ in } f$ stands for $(\lambda(x)f)(e)$. We allow the type parameters to be omitted from function terms if they are empty. Type ascription $(e : \sigma)$ stands for $(\lambda(x : \sigma)x)(e)$, which we use to represent return types for functions $\lambda[\alpha](x : \tau) : \sigma e$ as $\lambda[\alpha](x : \tau)(e : \sigma)$.

21.2.3. Type Inference Algorithm

We now define a type inference algorithm based on symbolic closures that recovers types from terms written in the external language. First, we give the syntax for symbolic closures, then we describe the organization of type inference, and finally fill in the missing details.

e, f	::= ... $\lambda_{\mathbb{I}}(x)e$	Terms
τ, σ, χ	::= ... $\Gamma@^{\mathbb{I}}\lambda(x)e$	Types
\mathbb{I}	::= α	Symbolic Closure Identifiers
\mathbb{F}	::= n	Symbolic Reduction Fuel
\mathbb{E}	::= $\overline{\mathbb{I} \Rightarrow \Gamma@e}$	Elaboration Caches
Δ	::= $\mathbb{F}; \mathbb{E}$	Threaded Environments

FIGURE 21.8. Symbolic Closure Language (SCL) Syntax (extends Figure 21.5)

The syntax for the Symbolic Closure Language (SCL) is given in Figure 21.8, which is a superset of the external language syntax. We introduce a new term and type, both which act as a sort of placeholder for an explicitly typed function term or type (respectively) which will be filled in after type checking. The term $\lambda_{\mathbb{I}}(x)e$ is a tagged function, which says the unannotated function $\lambda(x)e$ was assigned the symbolic closure type identified by \mathbb{I} . A symbolic closure type $\Gamma@^{\mathbb{I}}\lambda(x)e$, then, says the unannotated function term $\lambda(x)e$ is closed under definition type context Γ , with identifier \mathbb{I} . We say $\lambda(x)e$ is *closed* because all free type and term variables in $\lambda(x)e$ are bound by Γ . In terms of the internal language, a loose first-intuition of a symbolic closure type's meaning is a function type $\tau \xrightarrow{\bar{\alpha}} \sigma$ where $\Gamma, \bar{\alpha}, x : \tau \vdash e : \sigma$. This analogy is inadequate because (in small part) Γ , τ , e , and σ can contain SCL types and terms.

An important part of using SCL for type inference is making the meaning of symbolic closure types explicit by replacing them with concrete internal types and terms. To this end, the remaining syntax is in service to the bookkeeping necessary to decide how to achieve this. To prevent infinite loops when checking symbolic closures, we introduce symbolic reduction fuel \mathbb{F} , a natural number that represents the remaining number of symbolic reductions allowed. The elaboration of an unannotated function checked with symbolic closures could be determined at any time, so an elaboration cache \mathbb{E} is maintained that associates a symbolic closure \mathbb{I} with its scoped elaboration $\Gamma@e$, which says SCL term e is closed under SCL type environment Γ . For convenience, we use threaded environments Δ to stand for the pair $\mathbb{F}; \mathbb{E}$.

We now describe the organization of type inference. The typing judgment for SCL is written $\mathbb{F}; \mathbb{E}; \Gamma \vdash e : \tau; \mathbb{F}'; \mathbb{E}'; e'$ and says with initial fuel \mathbb{F} and elaboration cache \mathbb{E} , external term e is of SCL type τ in SCL context Γ , elaborating to SCL term e' with updated fuel \mathbb{F}' and elaboration cache \mathbb{E}' . It performs a depth-first traversal of the syntax tree.

$$\frac{\text{INFER} \quad \exists \mathbb{F}. \mathbb{F}; \emptyset; \Gamma \vdash e : \sigma; \mathbb{F}'; \mathbb{E}; e' \quad \text{elim}(\emptyset, \mathbb{E}, \sigma) = \tau \quad \text{elab}(\mathbb{E}, e') = f}{\Gamma \vdash e : \tau \hookrightarrow f}$$

$\Delta; \Gamma \vdash e : \tau; \Delta'; e'$

Given symbolic closure environment Δ and SCL context Γ , external term e has SCL type τ in environment Δ' , with SCL elaboration e' (omitted when obvious from subderivations).

$$\begin{array}{c}
\text{SC-VAR} \\
\Delta; \Gamma \vdash x : \Gamma(x); \Delta \\
\text{SC-REC} \\
\frac{\Delta_{i-1}; \Gamma \vdash f_i : \tau_i; \Delta_i \quad \rightarrow 1 \leq i \leq n}{\Delta_0; \Gamma \vdash \{x = f^n\} : \{\bar{x} : \bar{\tau}^n\}; \Delta_n} \\
\text{SC-ABS} \\
\frac{\left(\begin{array}{l} |\bar{\alpha}| > 0 \text{ implies } \Gamma \text{ and } \sigma \\ \text{contain no symbolic closures} \end{array} \right) \quad \Delta; \Gamma, \bar{\alpha}, x : \tau \vdash e : \sigma; \Delta'}{\Delta; \Gamma \vdash \lambda[\bar{\alpha}](x:\tau)e : \tau \xrightarrow{\bar{\alpha}} \sigma; \Delta'} \\
\text{SC-SEL} \\
\frac{\Delta; \Gamma \vdash f : \{x_1 : \tau_1, \dots, x_i : \tau_i, \dots, x_n : \tau_n\}; \Delta'}{\Delta; \Gamma \vdash f.x_i : \tau_i; \Delta'} \\
\text{SC-APP} \\
\frac{\Delta_1; \Gamma \vdash f : \tau \xrightarrow{\bar{\alpha}} \sigma; \Delta_2 \quad \Delta_2; \Gamma \vdash e : \tau'; \Delta_3 \quad \Delta_3 \vdash \tau' \leq [\bar{\chi}/\bar{\alpha}]\tau; \Delta_4}{\Delta_1; \Gamma \vdash f[\bar{\chi}](e) : [\bar{\chi}/\bar{\alpha}]\sigma; \Delta_4} \quad \text{SC-APP}\perp \\
\frac{\Delta; \Gamma \vdash f : \perp; \Delta'' \quad \Delta''; \Gamma \vdash e : \sigma; \Delta'}{\Delta; \Gamma \vdash f[\bar{\chi}](e) : \tau; \Delta'} \quad \text{SC-SEL}\perp \\
\frac{\Delta; \Gamma \vdash f : \perp; \Delta' \quad \Delta''; \Gamma \vdash e : \sigma; \Delta'}{\Delta; \Gamma \vdash f.x : \perp; \Delta'} \quad \text{SC-UAPP}\perp \\
\frac{\Delta_1; \Gamma \vdash f : \forall \bar{\alpha}. [\tau \rightarrow \sigma]; \Delta_2 \quad \Delta_2; \Gamma \vdash e : \tau'; \Delta_3 \quad \tau, \sigma, \tau' \text{ contain no symbolic closures} \quad |\bar{\alpha}| > 0 \quad \emptyset \vdash_{\bar{\alpha}} \tau' <: \tau \Rightarrow C \quad \text{genSubst}(C, [\tau \rightarrow \sigma]) = S}{\Delta_1; \Gamma \vdash f(e) : S\sigma; \Delta_3} \quad \text{SC-APPINFPT} \\
\text{SC-UABS} \\
\frac{\Delta = \mathbb{F}; \mathbb{E} \quad \mathbb{I} \notin \text{dom}(\mathbb{E}) \quad \Delta' = \mathbb{F}; \mathbb{E}[\mathbb{I} \mapsto \Gamma @ \lambda(x)e]}{\Delta; \Gamma \vdash \lambda(x)e : \Gamma @^{\mathbb{I}} \lambda(x)e; \Delta'; \lambda_{\mathbb{I}}(x)e} \quad \text{SC-UAPPCLO} \\
\frac{\Delta_1; \Gamma \vdash f : \Gamma @^{\mathbb{I}} \lambda(x)e'; \Delta_2; f' \quad \Delta_2; \Gamma \vdash e : \tau; \mathbb{F}_3; \mathbb{E}_3; e'' \quad 0 < \mathbb{F}_3 \quad \mathbb{F}_3 - 1; \mathbb{E}_3; \Gamma', x : \tau \vdash e' : \sigma; \mathbb{F}_4; \mathbb{E}_4; f''}{\Delta_1; \Gamma \vdash f(e) : \sigma; \mathbb{F}_4; \text{pick}_{\mathbb{E}_4}(\mathbb{I}, \lambda[] (x:\tau): \sigma f''); f' [] (e'')}
\end{array}$$

FIGURE 21.9. Type inference algorithm

The top-level driver for type inference $\Gamma \vdash e : \tau : f \hookrightarrow$, presented above, says external term e has internal type τ in external environment Γ , with internal elaboration f . It requires an initial fuel \mathbb{F} to be provided to SCL, and then uses the output elaboration cache \mathbb{E} to erase SCL terms and types using the metafunctions `elab` and `elim`. Next, we present the SCL type system and subtyping then provide definitions for the elaboration metafunctions.

The SCL type system is given in Figure 21.9. We abbreviate $\mathbb{F}; \mathbb{E}; \Gamma \vdash e : \tau; \mathbb{F}'; \mathbb{E}'; e'$ as $\Delta; \Gamma \vdash e : \tau; \Delta'; e'$ where $\Delta = \mathbb{F}; \mathbb{E}$ and $\Delta' = \mathbb{F}'; \mathbb{E}'$. Further, we sometimes omit the elaborated term as $\Delta; \Gamma \vdash e : \tau; \Delta'$, but only when e' can be obviously derived from subderivations. The first seven rules correspond to the internal language type system, straightforwardly extended with threaded environments. The extra condition in SC-ABS helps ensure a symbolic closure type only reasons about type variables in its definition scope. The first rule for lightweight applications SC-UAPP \perp implements E-UAPP \perp . The SC-APPINFPT rule uses Pierce and Turner's type argument synthesis

algorithm [66] off-the-shelf. Since it does not handle them, we ensure that it cannot be fed a symbolic closure.

The remaining rules are more interesting. The symbolic closure introduction rule SC-UABS creates a symbolic closure type with a fresh identifier \mathbb{I} for the unannotated function term $\lambda(x)e$. The return type is symbolic closure type $\Gamma@^{\mathbb{I}}\lambda(x)e$, which holds enough information to both check its body at some later time and link its elaboration to the originating term. The elaboration cache entry for \mathbb{I} is initialized with an unannotated function term, signifying that the body has yet to be type checked, and is passed on as part of Δ' . Finally, the elaboration $\lambda_{\mathbb{I}}(x)e$ tags the original term with its symbolic closure identifier \mathbb{I} . Intuitively, this rule is sound because, from the type checker’s perspective, there is no witness (yet) to $\lambda(x)e$ being called, and so there is no opportunity to “get stuck” or “go wrong”. The next rule handles one kind of witness to its invocation: application.

The application rule for symbolic closures SC-UAPPCLO checks term $f(e)$ where f has symbolic closure type $\Gamma'@^{\mathbb{I}}\lambda(x)e'$ and e has type τ . As with the normal application rule, we must ensure f ’s domain is permissive enough to be applied to terms of type τ , which we verify with a symbolic reduction. After consuming fuel, the final premise checks \mathbb{I} ’s function body e' in its definition context Γ' , extended with $x : \tau$ (to account for e of type τ being passed as an argument), giving result type σ and elaboration f'' . The type of the entire application is simply σ , since the condition in SC-ABS (and in other rules, given later) ensure that Γ and Γ' share the same type variable scope—there is no opportunity for σ to introduce an out-of-scope type variable. We now pick the elaboration for \mathbb{I} to be $\lambda[](x:\tau):\sigma f''$ (an abbreviation for a function with return type σ , defined in Figure 21.7) using the `pick` metafunction. Since we choose \mathbb{I} to be monomorphic, it does not require type arguments and so the entire application elaborates to $f'[](e'')$. A symbolic closure’s elaboration may be picked exactly once, so it is an error for any past or future elaborations decide on different numbers of type arguments. We elide the almost-identical rule SC-APPCLO for applying a symbolic closure with explicit type arguments, since it can only be of the form $f[](e)$.

To illustrate how SC-UABS and SC-UAPPCLO interact, we give simplified definitions for each, using judgments resembling the internal language and a simplified symbolic closure type $\Gamma@\lambda(x)f$ that omits identifiers. SIMP-UABS immediately packages up a function with its environment in a symbolic closure type. SIMP-UAPPCLO unpacks the symbolic closure, and checks its body in a context extended with the argument’s type. Highlighting is used to convey how a symbolic closure is assembled, disassembled, and checked.

$$\begin{array}{c}
\boxed{\text{SIMP-UABS}} \\
\Gamma \vdash \lambda(x)f : \Gamma@{\lambda(x)}f
\end{array}
\quad
\frac{
\boxed{\text{SIMP-UAPPCLO}} \quad
\Gamma' \vdash e_1 : \Gamma@{\lambda(x)}f \quad
\Gamma' \vdash e_2 : \sigma \quad
\Gamma, x : \sigma \vdash f : \tau
}{
\Gamma' \vdash e_1(e_2) : \tau
}$$

Using these rules, we illustrate inferring the *obfuscated* term “let $g = \{d = \lambda(x)f\}$ in $g.d(e)$ ”, which is like the *unobfuscated* term “ $(\lambda(x)f)(e)$ ”, except the function is briefly stored in a let-bound record. Bidirectional propagation of types is insufficient to recover the type of x , however symbolic closures provide the necessary indirection to successfully infer the term. To simplify its presentation, the following derivation uses the standard explicit typing rule for let, where the left premise checks the bound term and the right premise checks the body in an extended context—we use the lambda encoding of lets everywhere else (Figure 21.7). We also liberally elide Γ (when uninteresting) and the subderivation trees of SIMP-UAPPCLO.

$$\frac{
\boxed{\text{SIMP-UABS}} \quad \Gamma \vdash \lambda(x)f : \Gamma@{\lambda(x)}f \quad
\boxed{\text{SIMP-UAPPCLO}} \quad
\text{g.d} : \Gamma@{\lambda(x)}f \quad e : \sigma \quad \Gamma, x : \sigma \vdash f : \tau
}{
\frac{
\{d = \lambda(x)f\} : \{d : \Gamma@{\lambda(x)}f\} \quad
\text{g} : \{d : \Gamma@{\lambda(x)}f\} \vdash \text{g.d}(e) : \tau
}{
\text{let } g = \{d = \lambda(x)f\} \text{ in } \text{g.d}(e) : \tau
}
}$$

The derivation proceeds as follows. First, the record term “ $\{d = \lambda(x)f\}$ ” is checked using the left subderivation, which ends with SIMP-UABS. There, to delay its checking, $\lambda(x)f$ is packaged with its definition context Γ in the symbolic closure $\Gamma@{\lambda(x)}f$, which finds itself in the resulting record type of the left subderivation “ $\{d : \Gamma@{\lambda(x)}f\}$ ”. The right subderivation then binds this record type as g in the type environment to check the body “ $g.d(e)$ ”. The first rule in the right subderivation is SIMP-UAPPCLO, since the operator $g.d$ has our symbolic closure type $\Gamma@{\lambda(x)}f$ via a rule like I-SEL. Next, the argument e is checked as type σ . Now, Γ , x , and f are extracted from the symbolic closure and, along with σ , used to check (effectively) our type-decorated unobfuscated term “ $(\lambda(x:\sigma)f)(e)$ ” with the derivation “ $\Gamma, x : \sigma \vdash f : \tau$.” The result type τ is then propagated to be the type of the entire derivation.

Local Type Inference and Colored Local Type Inference would have failed to infer the obfuscated term exactly at SIMP-UABS, because it requires the type of x to be known from its context at the point $\lambda(x)f$ is checked. Indeed, they also fail to check our unobfuscated term “ $(\lambda(x)f)(e)$ ” for

$\Delta \vdash \sigma \leq \tau; \Delta'$ With symbolic closure environment Δ , σ is a subtype of τ in updated environment Δ' .			
SCS-TVAR $\Delta \vdash \alpha \leq \alpha; \Delta$	SCS-TOP $\Delta \vdash \tau \leq \top; \Delta$	SCS-BOT $\Delta \vdash \perp \leq \tau; \Delta$	SCS-REC $\frac{\Delta_{i-1} \vdash \tau \leq \sigma; \Delta_i^{\rightarrow 1 \leq i \leq n}}{\Delta_0 \vdash \{\bar{x} : \tau^n, \bar{x}' : \tau'\} \leq \{\bar{x} : \sigma^n\}; \Delta_n}$
SCS-FN $\left(\begin{array}{l} \bar{\alpha} > 0 \text{ implies } \tau, \tau', \sigma, \sigma' \\ \text{contain no symbolic closures} \end{array} \right)$ $\frac{\Delta \vdash \sigma \leq \sigma'; \Delta'' \quad \Delta'' \vdash \tau \leq \tau'; \Delta'}{\Delta \vdash \sigma' \xrightarrow{\bar{\alpha}} \tau \leq \sigma \xrightarrow{\bar{\alpha}} \tau'; \Delta'}$		SCS-CLO $\frac{\begin{array}{l} \Delta_1 = \mathbb{F}_1; \mathbb{E}_1 \quad \text{tv}(e) \cap \bar{\alpha} = \emptyset \\ 0 < \mathbb{F}_1 \quad \mathbb{F}_1 - 1; \mathbb{E}_1; \Gamma, \bar{\alpha}, x : \tau \vdash e : \sigma'; \mathbb{F}_2; \mathbb{E}_2; e' \\ \mathbb{F}_2; \text{pick}_{\mathbb{E}_2}(\mathbb{I}, \lambda[\bar{\alpha}](x:\tau):\sigma'e') \vdash \sigma' \leq \sigma; \Delta_3 \end{array}}{\Delta_1 \vdash \Gamma @^{\mathbb{I}} \lambda(x)e \leq \tau \xrightarrow{\bar{\alpha}} \sigma; \Delta_3}$	

FIGURE 21.10. Symbolic Closure Language Subtyping

similar reasons. In that case, their application rules require a type for the operator before checking its operand, and so no information may be propagated from operand to operator. This is fatal to checking the unobfuscated term, since they cannot synthesize the type of functions from nothing (unlike SIMP-UABS).

Subtyping for SCL is given in Figure 21.10. The judgment $\mathbb{F}; \mathbb{E} \vdash \sigma \leq \tau; \mathbb{F}'; \mathbb{E}'$ says with fuel \mathbb{F} and elaboration cache \mathbb{E} , σ is a subtype of τ with updated fuel \mathbb{F}' and elaboration cache \mathbb{E}' . Similar to the typing judgment, we abbreviate subtyping with threaded environments as $\Delta \vdash \sigma \leq \tau; \Delta'$. The first five rules correspond to the internal language subtyping rules, extended with threaded environments. The extra condition in SCS-FN helps contain symbolic closures to the type-variable scope they were defined in. The rule SCS-CLO relates symbolic closures with polymorphic function types. It follows the idea that $\Gamma @^{\mathbb{I}} \lambda(x)e$ is a subtype of $\tau \xrightarrow{\bar{\alpha}} \sigma$ if $\lambda[\bar{\alpha}](x:\tau):\sigma'e$ is well typed under Γ and σ' is a subtype of σ . The rule proceeds similarly to SC-UAPPCLO, except we may choose a polymorphic type for \mathbb{I} , and we must check the return type is under σ . Adding a type binder to a term invites the possibility of unintentional variable capture, and so the condition on $\bar{\alpha}$ avoids capturing free type variables in e .

To illustrate SCS-CLO's role, we again aggressively simplify our presentation. The simplified rule SIMPS-CLO extracts symbolic closure's terms and definition environment to check it with the provided input type and type arguments.

$$\text{SIMPS-CLO} \frac{\Gamma, \bar{\alpha}, x : \tau \vdash e : \sigma' \quad \sigma' \leq \sigma}{\Gamma @ \lambda(x) e \leq \tau \xrightarrow{\bar{\alpha}} \sigma}$$

We now illustrate how to check $(\{d = \lambda(x)f\} : \tau \xrightarrow{\bar{\alpha}} \sigma)$ a slightly more complicated version of $((\lambda(x)f) : \tau \xrightarrow{\bar{\alpha}} \sigma)$. To streamline presentation, we temporarily ignore the fact that type ascription is syntactic sugar (defined Figure 21.7) and use its standard typing rule, which first synthesizes a type for the term, then checks it is a subtype of the provided type.

$$\frac{\frac{\text{SIMPS-CLO} \quad \Gamma, \bar{\alpha}, x : \tau \vdash f : \sigma' \quad \sigma' \leq \sigma}{\Gamma @ \lambda(x) f \leq \tau \xrightarrow{\bar{\alpha}} \sigma} \quad \frac{\text{SIMP-UABS} \quad \Gamma \vdash \lambda(x)f : \Gamma @ \lambda(x)f}{\{d = \lambda(x)f\} : \{d : \Gamma @ \lambda(x)f\}}}{\text{let } g = \{d = \lambda(x)f\} \text{ in } (g.d : \tau \xrightarrow{\bar{\alpha}} \sigma) : \tau \xrightarrow{\bar{\alpha}} \sigma} \quad \frac{\Gamma @ \lambda(x) f \leq \tau \xrightarrow{\bar{\alpha}} \sigma \quad g.d : \Gamma @ \lambda(x) f}{g : \{d : \Gamma @ \lambda(x)f\} \vdash (g.d : \tau \xrightarrow{\bar{\alpha}} \sigma) : \tau \xrightarrow{\bar{\alpha}} \sigma}$$

The derivation begins in a similar fashion to the previous one, with the left subderivation using SIMP-UABS to delay the type checking of $\lambda(x)f$ via the symbolic closure $\Gamma @ \lambda(x)f$. The right subderivation begins with a rule checking type ascription, specifically that term “g.d” has function type $\tau \xrightarrow{\bar{\alpha}} \sigma$. First, a type is synthesized for “g.d”, which reveals our symbolic closure type from the left subderivation. Then, we check that the symbolic closure is a subtype of our desired function type via the symbolic reduction “ $\Gamma, \bar{\alpha}, x : \tau \vdash f : \sigma'$.” Finally, the return type of the symbolic reduction is checked to be compatible with our desired function type with $\sigma' \leq \sigma$.

Now that we have covered the type system and subtyping, we turn to the elaboration rules, which are given in Figure 21.11. They are split into two metafunctions `elab` and `elim`, elaborating terms and types respectively, along with `pick`, which manages when a symbolic closure’s elaboration may be chosen.

For terms, $\text{elab}(\mathbb{E}, e) = e'$ elaborates e to e' using elaboration cache \mathbb{E} . The case for tagged unannotated functions $\lambda_{\mathbb{I}}(x)e$ simply uses the elaboration entry for \mathbb{I} to continue elaboration. The metafunction is undefined for the external language’s unannotated terms $\lambda(x)e$ and $f(e)$. This enforces that each symbolic closure must be symbolically executed at least once to elaborate away these terms. Unannotated applications $f(e)$ are elaborated away with local type argument synthesis.

$\text{pick}_{\mathbb{E}}(\mathbb{I}, e) = \mathbb{E}'$ Pick SCL elaboration e for symbolic closure identifier \mathbb{I} .
--

$$\text{pick}_{\mathbb{E}}(\mathbb{I}, e) = \mathbb{E}[\mathbb{I} \mapsto \Gamma @ e], \quad \text{where } (\mathbb{E}[\mathbb{I}] = \Gamma @ \lambda(x)f) \text{ or } (\mathbb{E}[\mathbb{I}] = \Gamma @ e)$$

$\text{elab}(\mathbb{E}, e) = e'$ Converts symbolic closures in e to explicit types in e'

$$\begin{aligned}
\text{elab}(\mathbb{E}, x) &= x \\
\text{elab}(\mathbb{E}, f[\overline{\chi}](e)) &= \text{elab}(\mathbb{E}, f)[\overline{\text{elim}(\emptyset, \mathbb{E}, \chi)}](\text{elab}(\mathbb{E}, e)) \\
\text{elab}(\mathbb{E}, f.x) &= \text{elab}(\mathbb{E}, f).x \\
\text{elab}(\mathbb{E}, \overline{\{x = f\}}) &= \overline{\{x = \text{elab}(\mathbb{E}, f)\}} \\
\text{elab}(\mathbb{E}, \lambda[\overline{\alpha}](x:\tau)e) &= \lambda[\overline{\alpha}](x:\text{elim}(\emptyset, \mathbb{E}, \tau))\text{elab}(\mathbb{E}, e) \\
\text{elab}(\mathbb{E}, \lambda_{\mathbb{I}}(x)e) &= \text{elab}(\mathbb{E}, f), \quad \text{where } \mathbb{E}[\mathbb{I}] = \Gamma @ f
\end{aligned}$$

$\text{elim}(\overline{\mathbb{I}}, \mathbb{E}, \tau) = \tau'$ Converts symbolic closures in τ to explicit types in τ' , with seen symbolic closures $\overline{\mathbb{I}}$.

$$\begin{aligned}
\text{elim}(\overline{\mathbb{I}}, \mathbb{E}, \top) &= \top \\
\text{elim}(\overline{\mathbb{I}}, \mathbb{E}, \perp) &= \perp \\
\text{elim}(\overline{\mathbb{I}}, \mathbb{E}, \alpha) &= \alpha \\
\text{elim}(\overline{\mathbb{I}}, \mathbb{E}, \tau \xrightarrow{\overline{\alpha}} \sigma) &= \overline{\text{elim}(\overline{\mathbb{I}}, \mathbb{E}, \tau)} \xrightarrow{\overline{\alpha}} \text{elim}(\overline{\mathbb{I}}, \mathbb{E}, \sigma) \\
\text{elim}(\overline{\mathbb{I}}, \mathbb{E}, \overline{\{x:\tau\}}) &= \overline{\{x:\text{elim}(\overline{\mathbb{I}}, \mathbb{E}, \tau)\}} \\
\text{elim}(\overline{\mathbb{I}}, \mathbb{E}, \Gamma @^{\mathbb{I}'} e) &= \text{elim}(\overline{\mathbb{I}\mathbb{I}'}, \mathbb{E}, \tau) \xrightarrow{\overline{\alpha}} \text{elim}(\overline{\mathbb{I}\mathbb{I}'}, \mathbb{E}, \sigma), \quad \text{where } \mathbb{I}' \notin \overline{\mathbb{I}}, \mathbb{E}[\mathbb{I}'] = \Gamma @ \lambda[\overline{\alpha}](x:\tau):\sigma e
\end{aligned}$$

FIGURE 21.11. Elaboration Metafunctions for SCL Terms and Types

For elaborating types, `elab` uses $\text{elim}(\overline{\mathbb{I}}, \mathbb{E}, \tau) = \tau'$, which elaborates symbolic closures in τ using \mathbb{E} . Some extra bookkeeping is needed to prevent infinitely generating types. Since symbolic closures may be passed to other symbolic closures, a seen-set $\overline{\mathbb{I}}$ handles the case where it is passed to itself. Since we do not model equi-recursive types, we simply disallow that situation here.

To highlight the elaboration rules, we demonstrate the lambda-encoding of `let` in our system by checking term “`let $x = 42$ in f` ”—which desugars to “ $(\lambda(x)f)(42)$ ”—at some overall type σ with 42 having type `Int`. To streamline presentation, we assume that checking f does not introduce any symbolic closures (to keep the elaboration cache compact), liberally remove uninteresting type environments and elaboration caches, and omit symbolic reduction fuel from the derivation.

$$\begin{array}{c}
\text{SC-UABS} \\
\frac{\mathbb{E} = \mathbb{I} \Rightarrow \epsilon @ \lambda(x)f}{\varnothing; \epsilon \vdash \lambda(x)f : \epsilon @ \mathbb{I} \lambda(x)f; \mathbb{E}; \lambda_{\mathbb{I}}(x)f} \quad \begin{array}{l} 42 : \text{Int} \quad x : \text{Int} \vdash f : \sigma; f' \end{array} \\
\text{SC-UAPPCLO} \\
\frac{}{\varnothing; \epsilon \vdash (\lambda(x)f)(42) : \sigma; \mathbb{I} \Rightarrow \epsilon @ \lambda(x : \text{Int}) : \sigma f'; (\lambda_{\mathbb{I}}(x)f)(42)}
\end{array}$$

The derivation starts with the SC-UAPPCLO rule (chosen because the operator $\lambda(x)f$ has a symbolic closure type) with empty elaboration cache \varnothing and empty type environment ϵ . The distinct identifier \mathbb{I} is chosen by SC-UABS in the left subderivation, and is utilized there in three ways. First, the elaboration cache \mathbb{E} 's entry for \mathbb{I} is initialized to $\epsilon @ \lambda(x)f$, an untyped “placeholder” that signals that the final elaboration of \mathbb{I} has yet to be picked. Second, it identifies the rule's overall symbolic closure type $\epsilon @ \mathbb{I} \lambda(x)f$ for later elaboration. Third, it tags the rule's elaboration $\lambda_{\mathbb{I}}(x)f$ which, again, links the term to its elaboration cache entry. Like a traditional application rule, the middle subderivation checks the argument. The right subderivation performs a symbolic reduction $x : \text{Int} \vdash f : \sigma; f'$, where x and f come from the symbolic closure type and Int from the argument's type, with the result type σ being the type of the entire derivation. The elaboration f' will (eventually) be inserted in f 's place.

Now the goal is to stash enough information in the elaborated term and elaboration cache to eliminate unannotated terms using `elab` after type checking. For the entire derivation's elaborated term, tagged functions are preserved by combining the operator and operand elaborations in $(\lambda_{\mathbb{I}}(x)f)(42)$. For the elaboration cache, the (omitted) call `pickℙ($\mathbb{I}, \epsilon @ \lambda(x : \text{Int}) : \sigma f'$)` updates the elaboration cache's “placeholder” entry for \mathbb{I} to be $\mathbb{I} \Rightarrow \epsilon @ \lambda(x : \text{Int}) : \sigma f'$, where \mathbb{I} , ϵ , and x come from the symbolic closure type, Int from the argument's type, and σ , f' from the symbolic reduction.

The following call to `elab` then eliminates all SCL terms and types, performing a full elaboration into the internal language. Once a tagged function term is encountered, its elaboration is simply read off the cache and inserted. In this case, there is no need to also traverse f' since we assumed it does not contain symbolic closures, however in general a recursive `elab` call on f' would be needed to eliminate its symbolic closures.

$$\text{elab}(\mathbb{I} \Rightarrow \epsilon @ \lambda(x : \text{Int}) : \sigma f', (\lambda_{\mathbb{I}}(x)f)(42)) = (\lambda(x : \text{Int}) : \sigma f')(42)$$

CHAPTER 22

Symbolic Closure Metatheory Conjectures

We now outline the relationships between the internal, external, and symbolic closure languages that would be desirable as a series of conjectures. We leave the proofs as future work.

Ideally, SCL always infers sound annotations and types for external terms.

CONJECTURE 22.1 (SCL Soundness). *If there exists fuel \mathbb{F} such that $\mathbb{F}; \emptyset; \epsilon \vdash e : \tau; \mathbb{F}'; \mathbb{E}; e'$ for external term e , SCL type τ , and SCL term e' , then $\vdash \text{elab}(\mathbb{E}, e') : \tau'$ in the internal language, where $\tau' \leq \text{elim}(\emptyset, \mathbb{E}, \tau)$.*

Completeness for SCL says that there always exists some amount of annotations we can add to a term that type checks in the external language so it can type check in SCL.

CONJECTURE 22.2 (SCL Weak Completeness). *If $\vdash e : \tau$ for external term e , then there exists a term f such that e is a partial erasure of f and fuel \mathbb{F} such that $\mathbb{F}; \emptyset; \epsilon \vdash f : \tau; \mathbb{F}'; \mathbb{E}$.*

Since SCL essentially contains the rules of the internal language, we can always use the annotations chosen by the external language’s oracle. This way, we can erase all symbolic closure introduction rules and reuse previous results for systems based on $F_{<}$, so this theorem does not say much about symbolic closures. It would be nice to prove a stronger theorem based on fuel, such as the following.

CONJECTURE 22.3 (SCL Strong Completeness). *If $\vdash e : \tau$ for external term e , then either, for all initial fuel \mathbb{F} , SCL gets stuck at a symbolic reduction with zero fuel when checking e , or there exists fuel \mathbb{F} such that $\mathbb{F}; \emptyset; \epsilon \vdash e : \tau'; \mathbb{F}'; \mathbb{E}$, and $\text{elim}(\emptyset, \mathbb{E}, \tau') \leq \tau$.*

Unfortunately, this does not hold in the presented model of SCL, at least in part because of the restrictions placed on symbolic closures and our use of “off-the-shelf” type argument synthesis. Furthermore, we would need to distinguish type errors from running out of fuel. We now discuss some specific issues we face when trying to achieve a system with Strong Completeness, and speculate on how to fix them.

The restriction in SC-ABS disallows symbolic closures to cross into a new type variable scope, like “let $f = \lambda(x)e$ in $\lambda[\alpha]x^\alpha.f(x)$.” We have considered two fixes, both with their own tradeoffs. The

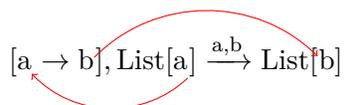
first is to infer a polymorphic type for f —we simply quantify over each type variable that occurs out-of-scope with respect to f 's definition context. In the present example, instead of recording f 's type as “ $\alpha \rightarrow \dots$ ” at the application site, we would record the polymorphic type “ $\alpha \xrightarrow{\alpha} \dots$ ”, since α is not in scope at f 's definition. The complication here is related to type argument synthesis. Since f now has a polymorphic type, the application must elaborate to $f[\alpha](x)$. Furthermore, if f is passed as an argument to a polymorphic function such as $\text{map}(f, e)$, we would have to also infer type arguments for operands (like CDuce [14]).

Another approach to handling out-of-scope type variables is to enrich types with type contexts. Here, f would have type “ $(\alpha' \vdash \alpha' \rightarrow \dots)$ ”, which says “in a type context that starts with some type variable α , expands to $\alpha \rightarrow \dots$ ”. This way, checking the body of “ $\lambda[\alpha]x^\alpha.f(x)$ ” would effectively update f 's type to “ $\alpha \rightarrow \dots$ ”. This approach resembles contextual subtyping [24], except their dependence of types on type contexts is purely syntactic. Type contexts are eliminated too early to check our example, because the f 's annotation must occur at its definition, but there the type environment is empty.

Another major restriction is that we may only pick a single elaboration for each symbolic closure. This disallows simple programs like $\text{let } f = \lambda(x)e \text{ in } \{\text{left} = f(1), \text{right} = f(\text{“a”})\}$. We have considered several approaches to lifting this restriction. First is to infer an *intersection type* for f . That way, f 's type would be $(\cap[\text{Int} \rightarrow \text{Int}], [\text{Str} \rightarrow \text{Str}])$, which says “returns an Int when given an Int , and returns a Str when given a Str .” Now choosing the final elaboration for e becomes more complicated, but the intersection type checking literature presents several solutions [82, 24, 13].

We could also try and guess a polymorphic type for f based on its use sites, in an approach resembling Trace Typing [3]. Here, say we know f is type $[\text{Int} \rightarrow \text{Int}]$ and we learn it is also of type $[\text{Str} \rightarrow \text{Str}]$, we could guess a generalization like “ $\alpha \xrightarrow{\alpha} \alpha$ ” based on the shape of both observations. Since the definition type environment of f is always handy in the elaboration cache, we could check if f inhabits the generalized type (Trace Typing only checks use sites to verify a polymorphic type).

The restriction in SC-APPINFPT that type argument synthesis may not reason about symbolic closures rules out checking $\text{map}(\lambda(x)f, e)$. Supporting symbolic closures in type argument synthesis requires a following a notion of data flow in polymorphic types. For example, the free theorems [79] of the type for map implies that its function argument may be invoked only with elements of its collection argument. We can visualize the data flow map must adhere to.



is natural to leave symbolic closures unrestricted. While we believe unrestricted symbolic closures are type sound, we are unsure how to approach proving such a theorem.

Part V

Related and Future Work

CHAPTER 23

Related Work to Typed Clojure

Multimethods. [58] and collaborators present a sequence of systems [15, 16, 58] with statically-typed multimethods and modular type checking. In contrast to Typed Clojure, in these systems methods declare the types of arguments that they expect which corresponds to exclusively using `class` as the dispatch function in Typed Clojure. However, Typed Clojure does not attempt to rule out failed dispatches.

Record Types. Row polymorphism [80, 9, 37], 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 [18] can track similar invariants as HMaps with types for JS objects. They must deal with mutable objects, they feature refinement types and strong updates to the heap to track changes to objects.

TeJaS [53], another type system for JavaScript, also supports similar HMaps, with the ability to record the presence and absence of entries, but lacks a compositional flow-checking approach like occurrence typing.

Typed Lua [56] has *table types* which track entries in a mutable Lua table. Typed Lua changes the dynamic semantics of Lua to accommodate mutability: Typed Lua raises a runtime error for lookups on missing keys—HMaps consider lookups on missing keys normal.

Java Interoperability in Statically Typed Languages. Scala [63] has nullable references for compatibility with Java. Programmers must manually check for `null` as in Java to avoid null-pointer exceptions.

Other optional and gradual type systems. Several other gradual type systems have been developed for existing dynamically-typed languages. Reticulated Python [78] is an experimental gradually typed system for Python, implemented as a source-to-source translation that inserts dynamic checks at language boundaries and supporting Python’s first-class object system. Clojure’s nominal classes avoids the need to support first-class object system in Typed Clojure, however HMaps offer an alternative to the structural objects offered by Reticulated. Similarly, Gradualtalk [1] offers gradual typing for Smalltalk, with nominal classes.

Optional types have been adopted in industry, including Hack [35], and Flow [31] and TypeScript [77], two extensions of JavaScript. These systems support limited forms of occurrence typing, and do not include the other features we present.

Related work to Automatic Annotations

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

Examples of invariant detection tools include Daikon [27], DIDUCE [36], and Carrot [69], and typically enhance statically typed languages with more expressive types or contracts. Examples of dynamic type inference include our tool, Rubydust [2], JSTrace [70], and TypeDevil [68], and typically target untyped languages.

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

Daikon. Daikon can reason about very expressive relationships between variables using properties like ordering ($x < y$), linear relationships ($y = ax + b$), and containment ($x \in y$). It also supports reasoning with “derived variables” like fields ($x.f$), and array accesses ($a[i]$). Typed Clojure’s dynamic inference can record heterogeneous data structures like vectors and hash-maps, but otherwise cannot express relationships between variables.

There are several reasons for this. The most prominent is that Daikon primarily targets Java-like languages, so inferring simple type information would be redundant with the explicit typing disciplines of these languages. On the other hand, the process of moving from Clojure to Typed Clojure mostly involves writing simple type signatures without dependencies between variables.

Typed Clojure recovers relevant dependent information via occurrence typing [75], and gives the option to manually annotate necessary dependencies in function signatures when needed.

Reverse Engineering Programs with Static Analysis. Rigi [60] analyzes the structure of large software systems, combining static analysis with a user-facing graphical environment to allow users to view and manipulate the in-progress reverse engineering results. We instead use a static type system as a feedback mechanism, which forces more aggressive compacting of generated annotations.

Lackwit [61] uses static analysis to identify abstract data types in C programs. Like our work, they share representations between values, except they use type inference with representations encoded as types. Recursive representations are inferred via Felice and Coppos’s work on type inference with recursive types [10], where we rely on our imprecise “squashing” algorithms over incomplete runtime samples.

Soft Typing [12] uses static analysis to insert runtime checks into untyped programs for invariants that cannot be proved statically. Our approach is instead to let the user check the generated annotations with a static type system, with static type errors guiding the user to manually add casts when needed.

Schema Inference. [4] infer structural properties of JSON data using a custom JSON schema format. Their schema inference algorithm proceeds in two stages: schema inference and schema fusion. This resembles our collection and naive type environment construction phases. There are slight differences between schema fusion and our approach. Schema fusion upcasts heterogeneous array types to be homogeneous, where we maintain heterogeneous vector types until a differently-sized vector type is found in the same position. We also support function types, which JSON lacks. While they support nested data, they do not attempt to factor out common types as names or create recursive types like our squashing algorithms.

[23] present a machine learning algorithm to translate denormalized and nested data that is commonly found in NoSQL databases to traditional relational formats used by standard RDBMS. A key component is a schema generation algorithm which arranges related data into tables via a matching algorithm which discovers related attributes. Phases 1 and 2 of their algorithm are similar to our local and global squashing algorithms, respectively, in that first locally accessible information is combined, and then global information. They identify groups of attributes that have (possibly cyclic) relationships. Where our squashing algorithms for map types are based on (sets of) keysets—on the assumption that related entities use similar keysets—they also join attributes based on their similar values. This enables more effective entity matching via equivalent attributes

with different names (e.g., “Email” vs “UserEmail”). Our approach instead assumes programs are somewhat internally consistent, and instead optimizes to handle missing samples from incomplete dynamic analysis.

Other Annotation Tools. Static analyzers for JavaScript (TSInfer [51]) and for Python (Typepete [38] and PyType [33]) automatically annotate code with types. PyType and Typepete inferred nodes as `(? -> int)` and `Dict[(Sequence, object)] -> int`, respectively—our tool infers it as `[Op ->Int]` by also generating a compact recursive type. Similarly, a class-based translation of inferred both `left` and `right` fields as `Any` by PyType, and as `Leaf` by Typepete—our tool uses `Op`, a compact recursive type containing *both* `Leaf` and `Node`. This is similar to our experience with TypeWiz in Chapter 10. (We were unable to install TSInfer.)

NoRegrets [57] uses dynamic analysis to learn how a program is used, and automatically runs the tests of downstream projects to improve test coverage. Their *dynamic access paths* represented as a series of *actions* are analogous to our paths of path elements.

Related Work to Extensible Type Systems

Turnstile [17] type checks a program during expansion by repurposing the Racket macro system. Instead of the more standard approach of providing separate rules to check a macro, Turnstile typing rules specify both the expansion and checking semantics, and so ensuring the two are compatible becomes automatic. On the other hand, Typed Clojure does not have the goal of allowing users to override how language primitives type check. Instead, our goal is to provide a simple interface to write type rules for library functions and macros in a style that hides the necessary bookkeeping surrounding occurrence typing and scope management.

SugarJ [26] adds syntactic language extensibility to languages like Java, such as pair syntax, embedded XML, and closures. Desugarings are expressed as rewrite rules to plain Java. Similarly, work on *type-specific languages* [64] adds extensible systems for the definitions of specialized syntax literals to existing languages. The *type* of an expression determines how it is parsed and elaborated.

SoundX [54] presents a solution to a common dilemma in typed metaprogramming: whether to desugar before type checking, or vice-versa. The authors present a system where a form is type checked before being desugared, with a guarantee that only well-typed code is generated. Programmers specify desugarings with a combination of typing and rewriting rules, which are then connected to form a valid type derivation in a process called *forwarding*. We will explore whether we can get the same effect in Typed Clojure without requiring the user to understand typing rules.

Ziggurat [29] allows programmers to define the static and dynamic semantics of macros separately. To demonstrate its broad applicability, they choose Scheme-like macros that generate assembly code for the dynamic semantics. They advocate building towers of static analyses, so macros can be statically checked in terms the static semantics of other macros, instead of just their assembly code expansions which would otherwise be too difficult to check. This idea resembles our prototypes in defining custom typing rules for functions and macros in Typed Clojure, where the dynamic semantics are defined by runtime Clojure constructs (`defn` and `defmacro`), and towers of static semantics are progressively specified in terms of the static analysis of other Clojure forms.

Type Tailoring [34] is an approach to provide more information to a host type system than it might be capable of by itself. In particular, the authors use the host platform’s metaprogramming functionality to refine the types of calls based on the program syntax alone, as well as improve error messages by incorporating surface syntax. Their experiments are based in Typed Racket, that fully expands syntax before checking it. Since Typed Clojure recently changed to interleave macroexpansion and type checking, we could extend this technique to also refine calls based on the types of their arguments (like SoundX).

Other work is relevant to our investigations of improving the user experience of Typed Clojure. SweetT [67] automatically infers type rules for syntactic sugar. Helium [39] provides hooks into the type inference process for domain-specific type error messages.

Related Work to Symbolic Closures

Local Type Inference. Symbolic closures were originally designed as an extension of Local Type Inference [66]. Our presentation omitted their bidirectional checking (we did not propagate type information down the syntax tree using synthesis/checking rules) and so was not a superset of Local Type Inference—in particular, it does not take advantage of the relaxed optimality conditions when inferring type arguments in checking mode. However, adding back bidirectional checking should be possible by starting with the rules of Local Type Inference, adding a *synthesis* rule for functions that introduces a symbolic closure, application and subtyping rules for symbolic closures, and some side conditions to restrict how a symbolic closure may be reasoned about (like our “must not contain symbolic closures” conditions scattered in various rules). This way, a symbolic closure should only be introduced where Local Type Inference fails—when a type of a function must be synthesized—and so seems more likely to be a superset of Local Type Inference.

Colored Local Type Inference [62] extends Local Type Inference with partial information propagation. Their type inference algorithm does not use explicit synthesis/checking rules, instead passing “prototypes” P down the syntax tree that containing partial expected type information used for type checking. A prototype is a type T extended with the wildcard “?”, denoting unknown information, and the specific shape of a prototype denotes which type rule to use. The rule inferring unannotated functions $\lambda(x)e$ requires a prototype $T \rightarrow P$, where T is the fully known expected type for x . A symbolic closure could be introduced when checking an unannotated function with prototype $P \rightarrow P'$ or “?” (the equivalent of Local Type Inference’s “synthesis” rules). In the more complicated case of $P \rightarrow P'$, a symbolic reduction of $\lambda(x)e$ is required to ensure it at least conforms its the prototype. For example, inferring the type of $\text{map}(\lambda(x)e, [1, 2, 3])$ with Colored Local Type Inference (where map has type “[$a \rightarrow b$], List[a] $\xrightarrow{a,b}$ List[b]”) checks $\lambda(x)e$ with prototype $? \rightarrow ?$. We can be optimistic and check the function at the largest (most specific) subtype of $\perp \rightarrow \top$ that matches $? \rightarrow ?$, which is $\perp \rightarrow \top$. This ensures that the function at least conforms to the most optimistic interpretation of its prototype, and then by returning a symbolic closure type instead of $\perp \rightarrow \top$ allows us to check more specific requirements later. Of course, to fully check this example,

it requires that we specify how type argument synthesis works with symbolic closures, but it at least illustrates how symbolic closures relate to the rest of the system.

Spine-local type inference [47] explores Local Type Inference in the context of System F (without subtyping). They present a greedy type argument synthesis algorithm which more aggressively propagates type information to an application’s arguments. To check arguments, type variable instantiations are guessed based on the expected type of the application. For example, when checking $\text{id}(\lambda(x)e)$ with expected type $[\tau \rightarrow \sigma]$, where id has type $\alpha \xrightarrow{\alpha} \alpha$, α would be guessed to have type $[\tau \rightarrow \sigma]$ and then $\lambda(x)e$ would be checked at that type. This would fail if the application was in synthesis mode. In this specific example, symbolic closures would allow the checking of $\lambda(x)e$ to be delayed to when more type information is available, in either checking or synthesis modes. Unfortunately, it does not seem that their algorithm can check $\text{map}(\lambda(x)e, [1, 2, 3])$ even in checking mode, and so does not seem to assist us in solving similar problems with symbolic closures. This case does not check because only the type of e would be apparent from an expected type, not the type of x .

Mixing Symbolic Execution and Type Checking. Mix [49] allows an interplay of symbolic execution [50] with type checking by providing syntactic regions, with terms $\{t \ e \ t\}$ signaling to use type checking for e , and $\{s \ e \ s\}$ for symbolic execution. In Mix, for example, the term

$$\{s \ \text{let id} = \lambda(x)x \ \text{in} \ \{t \ \dots \ \{s \ \text{id}(3) \ s\} \ \dots \ \{s \ \text{id}(3.0) \ s\} \ \dots \ t\} \ s\}$$

symbolically executes $\{s \ \text{id}(3) \ s\}$ and $\{s \ \text{id}(3.0) \ s\}$, propagating result types `Int` and `Real` back to the typed regions, respectively. Comparatively, symbolic closures integrates only a small amount of symbolic execution with a type system, but in such a way that delayed symbolic computations may pass between typed regions. Since Mix cleanly separates symbolic execution and type checking and its formalism does not support function types, it is difficult to compare the two approaches. In rough terms, symbolic closures use typed regions by default and automatically adds symbolic regions around unannotated functions.

$$\{t \ \text{let id} = \{s \ \lambda(x)x \ s\} \ \text{in} \ \dots \ \text{id}(3) \ \dots \ \text{id}(3.0) \ \dots \ t\}$$

Typing rules are then added to introduce a symbolic closure type and also to apply them, which involves checking the delayed body in a typed region.

$$\Gamma \vdash \{\mathbf{s} \lambda(x)e \mathbf{s}\} : \Gamma @ \lambda(x)e \quad \frac{\Gamma \vdash \{\mathbf{t} f \mathbf{t}\} : \Gamma' @ \lambda(x)e' \quad \Gamma \vdash \{\mathbf{t} e \mathbf{t}\} : \sigma \quad \Gamma', x : \sigma \vdash \{\mathbf{t} e' \mathbf{t}\} : \tau}{\Gamma \vdash \{\mathbf{t} f(e) \mathbf{t}\} : \tau}$$

When forced to delineate type checking from symbolic execution like this, it interesting to ask to what degree symbolic closures even uses symbolic execution. Our view is that (at least) symbolic closures symbolically execute the runtime-closure introduction rule.

$$\rho \vdash \lambda(x)e \Downarrow [\rho, \lambda(x)e]$$

The symbolic closure type $\Gamma @ \lambda(x)e$ is then the symbolic value of the runtime closure $[\rho, \lambda(x)e]$, related by the following typing rule.

$$\frac{\overline{\vdash \rho(y) : \Gamma(y)}^{y \in \text{dom}(\rho)}}{\Gamma' \vdash [\rho, \lambda(x)e] : \Gamma @ \lambda(x)e}$$

As evidenced by the lack of symbolic regions in the above application rule, a “symbolic reduction” of a symbolic closure is not particularly related to symbolic execution—it merely kicks off some delayed type checking. However, Γ , σ , and τ in that rule may contain symbolic closure types, so symbolic values are being used to reason about the program.

Mix also uses symbolic execution to enhance simple type systems with flow-sensitivity. For example, the following Mix program uses symbolic execution to flow-sensitively reason about `int?`, a predicate that returns true only for integer values.

`\{\mathbf{s} \text{ let } f = \lambda(x)(\text{if int?}(x) \text{ then } x \text{ else nil}) \text{ in } \{\mathbf{t} \dots \{\mathbf{s} f(3) \mathbf{s}\} \dots \{\mathbf{s} f(3.0) \mathbf{s}\} \dots \mathbf{t}\} \mathbf{s}\}`

The symbolic regions determine `\{\mathbf{s} f(3) \mathbf{s}\}` has type `Int` and `\{\mathbf{s} f(3.0) \mathbf{s}\}` type `Nil` via symbolic execution. Symbolic closures are instead designed to be compatible with flow-sensitive type systems like occurrence typing [75]. Here is the analogous program using symbolic closures.

`\{\mathbf{t} \text{ let } f = \{\mathbf{s} \lambda(x)(\text{if int?}(x) \text{ then } x \text{ else nil}) \mathbf{s}\} \text{ in } \dots f(3) \dots f(3.0) \dots \mathbf{t}\}`

Now, let us assume occurrence typing is also used to check this program, and that `int?` is typed as a predicate for integers. The call to `f(3)` triggers the symbolic reduction

$$x : \text{Int} \vdash (\text{if } \text{int?}(x) \text{ then } x \text{ else nil}) : \text{Int}$$

which has type `Int`, because the `else`-branch is unreachable, and similarly `f(3.0)` triggers

$$x : \text{Real} \vdash (\text{if } \text{int?}(x) \text{ then } x \text{ else nil}) : \text{Nil}$$

which has type `Nil`, because the `then`-branch is unreachable.

Intersection Type Checking. In hindsight, the idea behind symbolic closures resembles intersection type checking, where the same code may be checked at multiple types. Carlier and Wells [11] give an approachable explanation of “expansion”, a mechanism that informs an intersection type system when it should check the same term at different types. This is achieved by splicing typing rules (like intersection-introduction) into existing typing derivations that are derived from the principal typings of subterms. In contrast, symbolic closures do not assume principal types are available, and delays the construction of typing derivation(s) for a delayed term altogether until it is obvious how to construct it. Then, it is matter of combining a symbolic closure’s typing derivations to recover the (intersection) type it was used at.

Higher-order Control Flow Analysis. Closure analysis [71] approximates the set of arguments which a given function may be applied, as well as which functions a given term may evaluate to. Each function term is labelled $\lambda^\ell x.e$, where the label ℓ abstracts over the set of all runtime closures $[\rho, \lambda x.e]$ where runtime environment ρ can choose arbitrary bindings for e ’s free term variables. In contrast, a “tagged” symbolic closure term $\lambda_{\mathbb{I}}(x)e$ uses identifier \mathbb{I} to stand for $[\rho, \lambda(x)e]$ where the bindings in the runtime environment ρ are of the types given in the type environment Γ where the term was encountered by the type checker. In an unrestricted setting of symbolic closures, the same term may be used with different identifiers. For example, in

$$\text{let } f = \lambda(x)\lambda(y)x \text{ in } \dots f(3)\dots f(3.0)\dots$$

the first call to `f` tags the inner function as $\lambda_{\mathbb{I}_1}(y)x$ with \mathbb{I}_1 standing for the set of closures whose runtime environments bind `x` to a value of type `Int`, and the second call tags it as $\lambda_{\mathbb{I}_2}(y)x$ with \mathbb{I}_2 standing for the set of closures who similarly bind `x` to a value of type `Real`.

Giannini and Rocca [32] provide the following strongly-normalizing term is not typable in System F, which we write in Clojure and refer to as **GR**.

```

(let [I (fn [a] a)
      K (fn [b] (fn [c] b))
      D (fn [d] (d d))]
  ((fn [x] (fn [y] ((y (x I))
                    (x K))))
   D))

```

Palsberg [65] uses GR to motivate program analyses that answer basic questions like:

- For every application point, which abstractions can be applied?
- For every abstraction, to which arguments can it be applied?

Symbolic closures answer neither of these questions. Instead, they provide answers relevant to checking and inferring types:

- Can GR accept an argument of type τ ?
- When given an argument of type τ , what type is the value returned by GR?
- Does GR inhabit $[\tau \rightarrow \sigma]$?

To illustrate, we turn to our preliminary implementation of symbolic closures¹ to explore GR. It exposes the type checking query `(tc p e)`, which returns the type of checking `e` at expected prototype `p`, where a prototype is a type that can contain “wildcards” `?`. Now we can query the type of GR as if it had a type. The caveat: without a rich enough prototype, a benign symbolic closure type may be provided as an answer—you only get out what you put in (for this reason, symbolic closures perform particularly well when top-level types are always provided).

For example, `(tc ? GR)` asks to synthesize a type for GR. Unsurprisingly, a symbolic closure type greets us (below).

$$\Gamma @ (\text{fn } [y] ((y (x I)) (x K))), \text{ where } \begin{array}{l} \Gamma = I : I_c, K : K_c, D : D_c, x : D_c, \\ I_c = \Gamma_I @ I \\ K_c = \Gamma_k @ K \\ D_c = \Gamma_D @ D \end{array}$$

The term of GR_c is the (call-by-value) normal form of GR, derived by applying the symbolic closure of `(fn [x] ...)` to D. The type environment Γ captures the type environment at the point the `(fn [y] ...)` term was type checked. There, the bindings I, K, and D are all symbolic closure types I_c , K_c , and D_c , respectively, with `x` also having type D_c as a result of the application.

¹<https://github.com/frenchy64/lti-model>

As explained in Section 21.2, two ways to query a symbolic closure are by applying it or using subtyping. We can experimentally discover what shape of argument `GR` accepts by querying it at different prototypes, and using error messages and visual inspections of `GR` and its normal form (calculated by `GRc`) as guidance. We started with the query `(tc [Any ->?] GR)`, which gives the error message `Cannot invoke Any`. Then we inspected `GRc`, and noticed `y` must have shape `[? ->[? ->?]]` based on its usage. Incorporating that information results in our first interesting query result.

```
(tc [[? -> [? -> ?]] -> ?]
  GR)
;=> [[Any -> [Any -> Nothing]] -> Nothing]
```

This was calculated by observing a result type of `Nothing` from the application of `GRc` to an argument of type `[Any ->[Any ->Nothing]]` (derived by minimizing/maximising wildcards in covariant/contravariant positions, respectively, with respect to the relevant part of the prototype).

We can find other interesting types `GRc` inhabits by varying the query.

```
(tc [[? -> [? -> Int]] -> ?]
  GR)
;=> [[Any -> [Any -> Int]] -> Int]

(tc (All [a] [[? -> [? -> a]] -> ?])
  GR)
;=> (All [a] [[Any -> [Any -> a]] -> a])
```

With the last query, we stumble on how to use `GR` as a glorified identity function. We can verify this by evaluating a few terms.

```
(GR (fn [_] (fn [_] 42))) ;=> 42
(GR (fn [_] (fn [_] 24))) ;=> 24
```

We can also synthesize the types of these calls.

```
(tc ? (GR (fn [_] (fn [_] 42))))
;=> Int
```

The original use case of symbolic closures is to type check top-level functions against provided types, but whose bodies are too difficult to type check with traditional means. The following (extreme) example shows how checking the definition of a simple identity function can be thwarted, and how symbolic closures can make checking the definition of functions much more flexible.

```

(tc (All [a] [a -> a])
  (fn [z]
    (GR (fn [_] (fn [_] z))))))
;=> (All [a] [a -> a])

```

This illustrates the promise of symbolic closures to treat previously untypable terms as “black-boxes” during type checking, especially in a setting where top-level type information is always provided.

Banerjee [6] achieves a similar effect by instrumenting the rank 2 fragment of the intersection type discipline with flow information of closure values. Function terms are labelled and arrow types are annotated with sets of labels which denote which functions it may represent. They demonstrate by analyzing the following term $(\lambda f.(\lambda x.fI)(f0))I$ where I represents the identity function. They label the term $(\lambda^1 f.(\lambda^2 x.f(\lambda^3 u.u))(f0))(\lambda^4 v.v)$ and infer overall type $t \xrightarrow{\{3\}} t$, which says values of this type originate from the lambda labeled 3, with fresh type variable t .

Their system inherits the principal typing property of intersection types, which we lack in Typed Closure. To compensate for this, our prototype for unrestricted symbolic closures types returns the full code and type environment for the corresponding closure of lambda 3, so it may be further checked later when more type information is available. For example, plugging this example into our prototype gives the following symbolic closure type (using their labelled lambda syntax) $\Gamma @ \lambda^3 u.u$, where $\Gamma = \{f : \{\} @ \lambda^4 v.v, x : \text{Int}\}$.

Hindley-Milner and Let-polymorphism. Kanellakis and Mitchell [48] provide a set of (pathological) ML programs that exhibit exponential growth in the size of their principal type schemes. We use their benchmarks to compare symbolic closures with global type inference in the style of Milner [59].

Example 3.1 of [48] uses a lambda-encoding of pairs to create an ML principal type which appears to grow exponentially in length, however has a linear time representation as a directed acyclic graph. It is designed to avoid ML’s let-construct to remove the influence of let-polymorphism. The idea behind the program is to duplicate types σ by placing them in (lambda-encoded) tuples, following the pattern $\sigma, \langle \sigma, \sigma \rangle, \langle \langle \sigma, \sigma \rangle, \langle \sigma, \sigma \rangle \rangle$, and so on. To compare with symbolic closures, let P stand for $(\text{fn } [x] (\text{fn } [z] (z x x)))$ and P_z for $(\text{fn } [z] (z x x))$ in the following, where the left hand side term has the right hand side type. We can see the size of the type grows linearly in the number of occurrences of P —specifically, the outermost symbolic closure’s *environment* increases in size.

$$\begin{aligned}
(P\ 1) & : x: \text{Int} \quad @P_z \\
(P\ (P\ 1)) & : x: (x:\text{Int}@P_z) \quad @P_z \\
(P\ (P\ (P\ 1))) & : x:(x:(x:\text{Int}@P_z)@P_z)@P_z
\end{aligned}$$

Example 3.4 of [48] exhibits exponential growth in the size of ML principal types by exploiting let-polymorphism’s ability to copy type variables and assign new names to them. It follows the following pattern, which is similar to the previous benchmark, except intermediate values are let-bound.

```

(let [x0 1
      x1 (P x0)]
  x1)
      (let [x0 1
            x1 (P x0)
            x2 (P x1)]
        x2)
          (let [x0 1
                x1 (P x0)
                x2 (P x1)
                x3 (P x2)]
            x3)

```

Unlike ML, we find that symbolic closures are rather sensitive to whether P is let-bound or copied. If let-bound at the top of each term, the types of each term are identical to the previous example, and so grow linearly in size. This is because the resulting type of each term is a symbolic closure of the P_z term occurring in P , whose definition type environment never increases to include new variables (in particular, x_1 , x_2 , and x_3 are never in-scope there). If P is copied, however, the number of symbolic closures types reachable from the innermost occurrence of P grows exponentially, and so the resulting type also grows exponentially. We can reduce this to linear growth with sharing as below, where x_i has type Pc_i , because the exponential growth happens by duplicating symbolic closure types. Each P_z term comes from a different copy of P , in particular the P_z term of symbolic closure type Pc_i originates from the P occurring on the right-hand-side of x_i .

$$\begin{aligned}
Pc_1 & = \{x_0 \text{Int}, \quad x \text{Int}\}@P_z \\
Pc_2 & = \{x_0 \text{Int}, x_1 Pc_1, \quad x Pc_1\}@P_z \\
Pc_3 & = \{x_0 \text{Int}, x_1 Pc_1, x_2 Pc_2, \quad x Pc_2\}@P_z \\
Pc_4 & = \{x_0 \text{Int}, x_1 Pc_1, x_2 Pc_2, x_3 Pc_3, \quad x Pc_3\}@P_z
\end{aligned}$$

Example 3.5 of [48] gives a series of terms whose ML principal type is doubly-exponential in the size of the term, reduced to exponential when converted to a directed acyclic graph. The pattern is below, which, for $i > 1$ and $j = i - 1$, binds x_i to $(\text{fn } [y] \ (x_j \ (x_j \ y)))$.

CHAPTER 27

Future work

The most pressing future work for Typed Clojure is part of the ongoing work presented after Part I.

27.1. Future work for Automatic Annotations

A larger scale investigation of Clojure usage patterns is now possible by repurposing the automatic annotation tool described in Part II to generate and enforce `clojure.spec` annotations. As well as testing the robustness of the tool's design, the resulting data would be useful in investigating general questions like how effectively Clojure users utilize unit and generative testing, how Clojure code evolves of code over time, and the prevalence of idioms that Typed Clojure and `clojure.spec` have (and have not) been designed around.

27.2. Future work for Extensible Types

Part III outlines a code analyzer that paves the way to a future implementation of extensible typing rules for Typed Clojure. The next steps in this direction involve deciding the user interface for such a system and performing a survey of commonly used macros to determine which features must be supported.

27.3. Future work for Symbolic Closures

Symbolic closures (Part IV) show much promise in improving the user-experience of Typed Clojure. However, our preliminary work is still not well understood. Chapter 22 outlines several conjectures we hope to first prove. Finally, the problem of integrating symbolic closures with type argument synthesis is a crucial piece of future work, that (we hope) will prove symbolic closures as indispensable in checking many common Clojure problems.

Bibliography

- [1] Esteban Allende et al. “Gradual typing for Smalltalk”. In: *Science of Computer Programming* 96 (2014), pp. 52–69.
- [2] Jong-hoon (David) An et al. “Dynamic Inference of Static Types for Ruby”. In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 459–472. ISSN: 0362-1340. DOI: 10.1145/1925844.1926437. URL: <https://doi.acm.org/10.1145/1925844.1926437>.
- [3] Esben Andreasen et al. “Trace Typing: An Approach for Evaluating Retrofitted Type Systems”. In: *ECOOP*. 2016.
- [4] Mohamed-Amine Baazizi et al. “Schema inference for massive JSON datasets”. In: *Extending Database Technology (EDBT)*. 2017.
- [5] Phil Bagwell. “Ideal hash trees”. In: *Es Grands Champs* 1195 (2001), pp. 5–2.
- [6] Anindya Banerjee. “A modular, polyvariant and type-based closure analysis”. In: *ACM SIGPLAN Notices*. Vol. 32. 8. ACM. 1997, pp. 1–10.
- [7] Ambrose Bonnaire-Sergeant and contributors. *core.typed*. URL: <https://github.com/clojure/core.typed>.
- [8] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. “Practical Optional Types for Clojure”. In: *European Symposium on Programming Languages and Systems*. Springer. 2016, pp. 68–94.
- [9] Luca Cardelli and John C. Mitchell. “Operations on records”. In: *Mathematical Structures in Computer Science*. 1991, pp. 3–48.
- [10] Felice Cardone and Mario Coppo. “Type inference with recursive types: Syntax and semantics”. In: *Information and Computation* 92.1 (1991), pp. 48–80.
- [11] Sébastien Carlier and Joe B Wells. “Expansion: the crucial mechanism for type inference with intersection types: A survey and explanation”. In: *Electronic Notes in Theoretical Computer Science* 136 (2005), pp. 173–202.
- [12] Robert Cartwright and Mike Fagan. “Soft Typing”. In: *Proc. PLDI*. 1991.

- [13] G. Castagna et al. “Polymorphic Functions with Set-Theoretic Types. Part 1: Syntax, Semantics, and Evaluation”. In: *POPL '14, 41st ACM Symposium on Principles of Programming Languages*. 2014, pp. 5–17.
- [14] G. Castagna et al. “Polymorphic Functions with Set-Theoretic Types. Part 2: Local Type Inference and Type Reconstruction”. In: *POPL '15, 42nd ACM Symposium on Principles of Programming Languages*. 2015, pp. 289–302.
- [15] Craig Chambers. “Object-Oriented Multi-Methods in Cecil”. In: *Proc. ECOOP*. 1992.
- [16] Craig Chambers and Gary T. Leavens. “Typechecking and Modules for Multi-methods”. In: *Proc. OOPSLA*. 1994.
- [17] Stephen Chang, Alex Knauth, and Ben Greenman. “Type Systems As Macros”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 694–705. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009886. URL: <https://doi.acm.org/10.1145/3009837.3009886>.
- [18] Ravi Chugh, David Herman, and Ranjit Jhala. “Dependent Types for JavaScript”. In: *Proc. OOPSLA*. 2012.
- [19] CircleCI. *CircleCI*. URL: <https://circleci.com>.
- [20] CircleCI. *Why we're no longer using core.typed*. 2015. URL: <https://blog.circleci.com/why-were-no-longer-using-core-typed/>.
- [21] CircleCI. *Why we're supporting Typed Clojure, and you should too!* 2013. URL: <https://blog.circleci.com/supporting-typed-clojure/>.
- [22] Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. “Advanced macrology and the implementation of Typed Scheme”. In: *IN PROC. 8TH WORKSHOP ON SCHEME AND FUNCTIONAL PROGRAMMING*. ACM Press, 2007, pp. 1–14.
- [23] Michael DiScala and Daniel J Abadi. “Automatic generation of normalized relational schemas from nested key-value data”. In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 295–310.
- [24] Joshua Dunfield and Frank Pfenning. “Tridirectional Typechecking”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: ACM, 2004, pp. 281–292. ISBN: 1-58113-729-X. DOI: 10.1145/964001.964025. URL: <http://doi.acm.org/10.1145/964001.964025>.
- [25] R Kent Dybvig. “Chez Scheme Version 9 User’s Guide”. In: *Cisco Systems, Inc.* (2018).
- [26] Sebastian Erdweg et al. “SugarJ: Library-based Syntactic Language Extensibility”. In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems*

- Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 391–406. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048099. URL: <https://doi.acm.org/10.1145/2048066.2048099>.
- [27] Michael D Ernst et al. “Dynamically discovering likely program invariants to support program evolution”. In: *IEEE Transactions on Software Engineering* 27.2 (2001), pp. 99–123.
- [28] Robert Bruce Findler and Matthias Felleisen. “Contracts for higher-order functions”. In: *ACM SIGPLAN Notices*. Vol. 37. 9. ACM. 2002, pp. 48–59.
- [29] David Fisher. “Static analysis for syntax objects”. In: *In ACM SIGPLAN International Conference on Functional Programming*. 2006.
- [30] Matthew Flatt. “Composable and Compilable Macros: You Want it When?” In: *ACM SIGPLAN Notices*. Vol. 37. 9. ACM. 2002, pp. 72–83.
- [31] *Flow*. 2018. URL: <https://flow.org>.
- [32] Paola Giannini and Simona Ronchi Della Rocca. “Characterization of typings in polymorphic type discipline”. In: *Proceedings Third Annual Symposium on Logic in Computer Science*. IEEE. 1988, pp. 61–70.
- [33] Google. *PyType*. 2018. URL: <https://github.com/google/pytype>.
- [34] Ben Greenman, Stephen Chang, and Matthias Felleisen. *Type Tailoring (Unpublished manuscript)*. URL: <https://www.ccs.neu.edu/home/types/resources/type-tailoring.pdf>.
- [35] *Hack Language*. 2018. URL: <https://hacklang.org>.
- [36] Sudheendra Hangal and Monica S Lam. “Tracking down software bugs using automatic anomaly detection”. In: *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. IEEE. 2002, pp. 291–301.
- [37] Robert Harper and Benjamin Pierce. “A Record Calculus Based on Symmetric Concatenation”. In: *Proc. POPL*. 1991.
- [38] Mostafa Hassan et al. “MaxSMT-Based Type Inference for Python 3”. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 12–19. ISBN: 978-3-319-96142-2.
- [39] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. “Scripting the Type Inference Process”. In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP '03. Uppsala, Sweden: ACM, 2003, pp. 3–13. ISBN: 1-58113-756-7. DOI: 10.1145/944705.944707. URL: <https://doi.acm.org/10.1145/944705.944707>.

- [40] David Herman, Aaron Tomb, and Cormac Flanagan. “Space-efficient Gradual Typing”. In: *Higher Order Symbol. Comput.* 23.2 (June 2010), pp. 167–189. ISSN: 1388-3690. DOI: 10.1007/s10990-011-9066-z. URL: <https://dx.doi.org/10.1007/s10990-011-9066-z>.
- [41] Rich Hickey. *Clojure Evaluation Documentation*. 2018. URL: <https://clojure.org/reference/evaluation>.
- [42] Rich Hickey. *Clojure sequence Documentation*. 2015. URL: <https://clojure.org/reference/sequences>.
- [43] Rich Hickey. *core.async*. 2018. URL: <https://github.com/clojure/core.async>.
- [44] Rich Hickey. “The Clojure programming language”. In: *Proc. DLS*. 2008.
- [45] Haruo Hosoya and Benjamin C Pierce. “How Good is Local Type Inference?” In: *Technical Reports (CIS)* (1999), p. 180.
- [46] Suresh Jagannathan, Stephen Weeks, and Andrew Wright. “Type-directed flow analysis for typed intermediate languages”. In: *International Static Analysis Symposium*. Springer. 1997, pp. 232–249.
- [47] Christopher Jenkins and Aaron Stump. “Spine-local Type Inference”. In: *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*. IFL 2018. Lowell, MA, USA: ACM, 2018, pp. 37–48. ISBN: 978-1-4503-7143-8. DOI: 10.1145/3310232.3310233. URL: <http://doi.acm.org/10.1145/3310232.3310233>.
- [48] Paris C Kanellakis and John C Mitchell. “Polymorphic unification and ML typing”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1989, pp. 105–115.
- [49] Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. “Mixing Type Checking and Symbolic Execution”. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’10. Toronto, Ontario, Canada: ACM, 2010, pp. 436–447. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806645. URL: <https://doi.acm.org/10.1145/1806596.1806645>.
- [50] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: <https://doi.acm.org/10.1145/360248.360252>.
- [51] Erik Krogh Kristensen and Anders Møller. “Inference and Evolution of TypeScript Declaration Files”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2017, pp. 99–115.
- [52] Jukka Lehtosalo. *mypy*. URL: <http://mypy-lang.org/>.

- [53] Benjamin S. Lerner et al. “TeJaS: Retrofitting Type Systems for JavaScript”. In: *Proceedings of the 9th Symposium on Dynamic Languages*. DLS '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 1–16. ISBN: 978-1-4503-2433-5. DOI: 10.1145/2508168.2508170. URL: <https://doi.acm.org/10.1145/2508168.2508170>.
- [54] Florian Lorenzen and Sebastian Erdweg. “Sound Type-dependent Syntactic Language Extension”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: ACM, 2016, pp. 204–216. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837644. URL: <https://doi.acm.org/10.1145/2837614.2837644>.
- [55] John M. Lucassen and David K. Gifford. “Polymorphic effect systems”. In: *Proc. POPL*. 1988.
- [56] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. “Typed Lua: An Optional Type System for Lua”. In: *Proc. Dyla*. 2014.
- [57] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. “Type Regression Testing to Detect Breaking Changes in Node.js Libraries”. In: *Proc. 32nd European Conference on Object-Oriented Programming (ECOOP)*. 2018.
- [58] Todd Millstein and Craig Chambers. “Modular Statically Typed Multimethods”. In: *Information and Computation*. Springer-Verlag, 2002, pp. 279–303.
- [59] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.
- [60] Hausi A Müller et al. “A reverse engineering environment based on spatial and visual software interconnection models”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 17. 5. ACM, 1992, pp. 88–98.
- [61] Robert O’Callahan and Daniel Jackson. “Lackwit: A program understanding tool based on type inference”. In: *In Proceedings of the 19th International Conference on Software Engineering*. Citeseer. 1997.
- [62] Martin Odersky, Matthias Zenger, and Christoph Zenger. “Colored Local Type Inference”. In: *Proc. ACM Symposium on Principles of Programming Languages*. 2001, pp. 41–53.
- [63] Martin Odersky et al. *An overview of the Scala programming language (second edition)*. Tech. rep. EPFL Lausanne, Switzerland, 2006.
- [64] Cyrus Omar et al. “Safely composable type-specific languages”. In: *European Conference on Object-Oriented Programming*. Springer. 2014, pp. 105–130.

- [65] Jens Palsberg. “Closure Analysis in Constraint Form”. In: *ACM Trans. Program. Lang. Syst.* 17.1 (Jan. 1995), pp. 47–62. ISSN: 0164-0925. DOI: 10.1145/200994.201001. URL: <http://doi.acm.org/10.1145/200994.201001>.
- [66] Benjamin C. Pierce and David N. Turner. “Local Type Inference”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’98. San Diego, California, USA: ACM, 1998, pp. 252–265. ISBN: 0-89791-979-3. DOI: 10.1145/268946.268967. URL: <https://doi.acm.org/10.1145/268946.268967>.
- [67] Justin Pombrio and Shriram Krishnamurthi. “Inferring type rules for syntactic sugar”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2018, pp. 812–825.
- [68] Michael Pradel, Parker Schuh, and Koushik Sen. “TypeDevil: Dynamic type inconsistency analysis for JavaScript”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 314–324.
- [69] Brock Pytlik et al. “Automated fault localization using potential invariants”. In: *arXiv preprint cs/0310040* (2003).
- [70] Claudiu Saftoiu. *JSTrace: Run-time type discovery for JavaScript*. Tech. rep. Technical Report CS-10-05, Brown University, 2010.
- [71] Peter Sestoft. “Analysis and efficient implementation of functional programs”. PhD thesis. DIKU, University of Copenhagen, 1991.
- [72] Uri Shaked. *TypeWiz*. 2018. URL: <https://github.com/urish/typewiz>.
- [73] Guy Steele. *Common Lisp: The Language (Second Edition)*. Elsevier, 1990.
- [74] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. “Practical Variable-Arity Polymorphism”. In: *Proc. ESOP*. 2009.
- [75] Sam Tobin-Hochstadt and Matthias Felleisen. “Logical Types for Untyped Languages”. In: *Proc. ICFP*. ICFP ’10. 2010.
- [76] Sam Tobin-Hochstadt and Matthias Felleisen. “The Design and Implementation of Typed Scheme”. In: *Proc. POPL*. 2008.
- [77] *TypeScript*. 2018. URL: <https://www.typescriptlang.org>.
- [78] Michael M. Vitousek et al. “Design and Evaluation of Gradual Typing for Python”. In: *Proc. DLS*. 2014.
- [79] Philip Wadler. “Theorems for free!” In: *FPCA*. Vol. 89. 4. 1989, pp. 347–359.
- [80] Mitchell Wand. *Type Inference for Record Concatenation and Multiple Inheritance*. 1989.

- [81] Joe B Wells. “Typability and type checking in the second-order lambda/-calculus are equivalent and undecidable”. In: *Proceedings of the Symposium on Logic in Computer Science, 1994*. IEEE. 1994, pp. 176–185.
- [82] Joe B Wells and Christian Haack. “Branching types”. In: *European Symposium on Programming*. Springer. 2002, pp. 115–132.
- [83] Ningning Xie and Bruno C d S Oliveira. “Let Arguments Go First”. In: *European Symposium on Programming*. Springer. 2018, pp. 272–299.

APPENDIX A

Full rules for λ_{TC}

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

FIGURE A.01. Syntax of Terms, Types, Propositions, and Objects

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

FIGURE A.02. Type abbreviations

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

FIGURE A.03. Judgment abbreviations

$$\begin{array}{c}
\text{T-LOCAL} \\
\frac{\Gamma \vdash \tau_x}{\sigma = (\cup \mathbf{nil} \mathbf{false})} \\
\frac{}{\Gamma \vdash x : \tau ; \bar{\sigma}_x | \sigma_x ; x}
\end{array}
\quad
\begin{array}{c}
\text{T-CONST} \\
\Gamma \vdash c : \delta_\tau(c) ; \mathbf{tt} | \mathbf{ff} ; \emptyset
\end{array}
\quad
\begin{array}{c}
\text{T-TRUE} \\
\Gamma \vdash \mathbf{true} : \mathbf{true} ; \mathbf{tt} | \mathbf{ff} ; \emptyset
\end{array}$$

$$\begin{array}{c}
\text{T-FALSE} \\
\Gamma \vdash \mathbf{false} : \mathbf{false} ; \mathbf{ff} | \mathbf{tt} ; \emptyset
\end{array}
\quad
\begin{array}{c}
\text{T-NIL} \\
\Gamma \vdash \mathbf{nil} : \mathbf{nil} ; \mathbf{ff} | \mathbf{tt} ; \emptyset
\end{array}
\quad
\begin{array}{c}
\text{T-NUM} \\
\Gamma \vdash n : \mathbf{N} ; \mathbf{tt} | \mathbf{ff} ; \emptyset
\end{array}$$

$$\begin{array}{c}
\text{T-DO} \\
\frac{\Gamma \vdash e_1 : \tau_1 ; \psi_{1+} | \psi_{1-} ; o_1}{\Gamma, \psi_{1+} \vee \psi_{1-} \vdash e : \tau ; \psi_+ | \psi_- ; o} \\
\frac{}{\Gamma \vdash (\mathbf{do} \ e_1 \ e) : \tau ; \psi_+ | \psi_- ; o}
\end{array}
\quad
\begin{array}{c}
\text{T-IF} \\
\frac{\Gamma \vdash e_1 : \tau_1 ; \psi_{1+} | \psi_{1-} ; o_1 \quad \Gamma, \psi_{1+} \vdash e_2 : \tau ; \psi_+ | \psi_- ; o \quad \Gamma, \psi_{1-} \vdash e_3 : \tau ; \psi_+ | \psi_- ; o}{\Gamma \vdash (\mathbf{if} \ e_1 \ e_2 \ e_3) : \tau ; \psi_+ | \psi_- ; o}
\end{array}$$

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

$$\begin{array}{c}
\text{T-APP} \\
\frac{\Gamma \vdash e : x : \sigma \xrightarrow[o_f]{\psi_{f+} | \psi_{f-}} \tau ; \psi_+ | \psi_- ; o \quad \Gamma \vdash e' : \sigma ; \psi'_+ | \psi'_- ; o'}{\Gamma \vdash (e \ e') : \tau[o'/x] ; \psi_{f+} | \psi_{f-} [o'/x] ; o_f[o'/x]}
\end{array}
\quad
\begin{array}{c}
\text{T-ABS} \\
\frac{\Gamma, \sigma_x \vdash e : \sigma' ; \psi_+ | \psi_- ; o}{\Gamma \vdash \lambda x^\sigma . e : x : \sigma \xrightarrow[o]{\psi_+ | \psi_-} \sigma' ; \mathbf{tt} | \mathbf{ff} ; \emptyset}
\end{array}$$

$$\begin{array}{c}
\text{T-CLOS} \\
\frac{\exists \Gamma . \rho \models \Gamma \text{ and } \Gamma \vdash \lambda x^\tau . e : \sigma ; \psi_+ | \psi_- ; o}{\vdash [\rho, \lambda x^\tau . e]_c : \sigma ; \psi_+ | \psi_- ; o}
\end{array}
\quad
\begin{array}{c}
\text{T-ERROR} \\
\Gamma \vdash \mathbf{err} : \perp ; \mathbf{ff} | \mathbf{ff} ; \emptyset
\end{array}$$

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

FIGURE A.04. Standard Typing Rules

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

$$\frac{\text{T-NEWSTATIC} \quad \overrightarrow{\text{JT}(C_i) = \tau_i} \quad \text{JT}(C) = \tau \quad \overrightarrow{\Gamma \vdash e_i \Rightarrow e'_i : \tau_i}}{\Gamma \vdash (\text{new}_{[\vec{C}_i]} C \vec{e}_i) : \tau; \text{tt|ff}; \emptyset}$$

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

$$\frac{\text{T-FIELDSTATIC} \quad \text{JT}(C_1) = \sigma \quad \vdash \sigma <: \mathbf{Object} \quad \text{JT}_{\text{nil}}(C_2) = \tau \quad \Gamma \vdash e \Rightarrow e' : \sigma}{\Gamma \vdash (. e fld_{C_2}^{C_1}) : \tau; \text{tt|tt}; \emptyset}$$

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

$$\frac{\text{T-METHODSTATIC} \quad \overrightarrow{\text{JT}(C_i) = \tau_i} \quad \text{JT}(C_1) = \sigma \quad \vdash \sigma <: \mathbf{Object} \quad \text{JT}_{\text{nil}}(C_2) = \tau \quad \Gamma \vdash e \Rightarrow e' : \sigma \quad \overrightarrow{\Gamma \vdash e_i \Rightarrow e'_i : \tau_i}}{\Gamma \vdash (. e (mth_{[[\vec{C}_i], C_2]}^{C_1} \vec{e}_i)) : \tau; \text{tt|tt}; \emptyset}$$

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

$$\text{T-INSTANCE} \quad \Gamma \vdash C \{fld : v\} : C; \text{tt|ff}; \emptyset$$

FIGURE A.05. Java Interop Typing Rules

$$\begin{array}{c}
\text{T-DEFMULTI} \\
\frac{\sigma = x:\tau \xrightarrow{o}^{\psi_+|\psi_-} \tau' \quad \sigma' = x:\tau \xrightarrow{o'}^{\psi'_+|\psi'_-} \tau'' \quad \Gamma \vdash e \Rightarrow e' : \sigma'}{\Gamma \vdash (\text{defmulti } \sigma \ e) : (\mathbf{Multi} \ \sigma \ \sigma') ; \text{tt|fff} ; \emptyset}
\\
\\
\text{T-DEFMETHOD} \\
\frac{\tau_m = x:\tau \xrightarrow{o}^{\psi_+|\psi_-} \sigma \quad \tau_d = x:\tau \xrightarrow{o'}^{\psi'_+|\psi'_-} \sigma' \quad \Gamma \vdash e_m \Rightarrow e'_m : (\mathbf{Multi} \ \tau_m \ \tau_d) \quad \Gamma \vdash e_v \Rightarrow e'_v : \tau_v}{\text{IsAProps}(o', \tau_v) = \psi''_+|\psi''_- \quad \Gamma, \tau_x, \psi''_+ \vdash e_b : \sigma ; \psi_+|\psi_- ; o \quad e' = (\text{defmethod } e'_m \ e'_v \ \lambda x^\tau . e_b)} \\
\Gamma \vdash (\text{defmethod } e_m \ e_v \ \lambda x^\tau . e_b) : (\mathbf{Multi} \ \tau_m \ \tau_d) ; \text{tt|fff} ; \emptyset
\\
\\
\text{T-ISA} \\
\frac{\Gamma \vdash e : \sigma ; \psi'_+|\psi'_- ; o \quad \Gamma \vdash e' \Rightarrow e'_1 : \tau \quad \text{IsAProps}(o, \tau) = \psi_+|\psi_-}{\Gamma \vdash (\text{isa? } e \ e') : \mathbf{B} ; \psi_+|\psi_- ; \emptyset}
\\
\\
\text{T-MULTI} \\
\frac{\vdash v \Rightarrow v' : \tau \quad \overline{\vdash v_k \Rightarrow v'_k : \tau} \quad \overline{\vdash v_v \Rightarrow v'_v : \sigma}}{\vdash [v, \{\overline{v_k \mapsto v'_k}\}]_m : (\mathbf{Multi} \ \sigma \ \tau) ; \text{tt|fff} ; \emptyset}
\\
\\
\text{FIGURE A.06. Multimethod Typing Rules}
\\
\\
\text{T-HMAP} \quad \text{T-KW} \\
\frac{\overline{\vdash v_k \Rightarrow v'_k : (\mathbf{Val} \ k)} \quad \overline{\vdash v_v \Rightarrow v'_v : \tau_v} \quad \mathcal{M} = \{k \mapsto \tau_v\}}{\vdash \{\overline{v_k \mapsto v'_k}\} : (\mathbf{HMap}^c \ \mathcal{M}) ; \text{tt|fff} ; \emptyset} \quad \Gamma \vdash k : (\mathbf{Val} \ k) ; \text{tt|fff} ; \emptyset
\\
\\
\text{T-GETHMAP} \\
\frac{\Gamma \vdash e : (\bigcup (\mathbf{HMap}^e \ \mathcal{M} \ \mathcal{A}))^i ; \psi_{1+}|\psi_{1-} ; o \quad \Gamma \vdash e_k \Rightarrow e'_k : (\mathbf{Val} \ k) \quad \overline{\mathcal{M}[k] = \tau}}{\Gamma \vdash (\text{get } e \ e_k) : (\bigcup \overline{\tau}^i) ; \text{tt|tt} ; \mathbf{key}_k(x)[o/x]}
\\
\\
\text{T-GETHMAPABSENT} \\
\frac{\Gamma \vdash e : (\mathbf{HMap}^e \ \mathcal{M} \ \mathcal{A}) ; \psi_{1+}|\psi_{1-} ; o \quad \Gamma \vdash e_k \Rightarrow e'_k : (\mathbf{Val} \ k) \quad k \in \mathcal{A}}{\Gamma \vdash (\text{get } e \ e_k) : \mathbf{nil} ; \text{tt|tt} ; \mathbf{key}_k(x)[o/x]}
\\
\\
\text{T-GETHMAPPARTIALDEFAULT} \\
\frac{\Gamma \vdash e : (\mathbf{HMap}^p \ \mathcal{M} \ \mathcal{A}) ; \psi_{1+}|\psi_{1-} ; o \quad \Gamma \vdash e_k \Rightarrow e'_k : (\mathbf{Val} \ k) \quad k \notin \text{dom}(\mathcal{M}) \quad k \notin \mathcal{A}}{\Gamma \vdash (\text{get } e \ e_k) : \top ; \text{tt|tt} ; \mathbf{key}_k(x)[o/x]}
\\
\\
\text{T-ASSOCHMAP} \\
\frac{\Gamma \vdash e \Rightarrow (\text{assoc } e' \ e'_k \ e'_v) : (\mathbf{HMap}^e \ \mathcal{M} \ \mathcal{A}) \quad \Gamma \vdash e_k \Rightarrow e'_k : (\mathbf{Val} \ k) \quad \Gamma \vdash e_v \Rightarrow e'_v : \tau \quad k \notin \mathcal{A}}{\Gamma \vdash (\text{assoc } e \ e_k \ e_v) : (\mathbf{HMap}^e \ \mathcal{M}[k \mapsto \tau] \ \mathcal{A}) ; \text{tt|fff} ; \emptyset}
\end{array}$$

FIGURE A.07. Map Typing Rules

$$\begin{array}{c}
\text{SO-REFL} \\
\vdash o <: o \\
\\
\text{SO-TOP} \\
\vdash o <: \emptyset \\
\\
\text{S-UNIONSUPER} \quad \text{S-UNIONSUB} \quad \text{S-FUNMONO} \\
\frac{\exists i. \vdash \tau <: \sigma_i}{\vdash \tau <: (\bigcup \vec{\sigma}^i)} \quad \frac{\vdash \tau_i <: \sigma}{\vdash \tau_i <: \sigma} \quad \vdash x:\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau <: \mathbf{Fn} \\
\\
\text{S-OBJECT} \\
\vdash C <: \mathbf{Object} \\
\\
\text{S-SBOOL} \\
\vdash (\mathbf{Val} b) <: \mathbf{B} \\
\\
\text{S-SCLASS} \\
\vdash (\mathbf{Val} C) <: \mathbf{Class} \\
\\
\text{S-SKW} \\
\vdash (\mathbf{Val} k) <: \mathbf{K} \\
\\
\\
\text{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'} \\
\\
\text{S-PMULTIFN} \quad \text{S-PMULTIFN} \\
\frac{\vdash \sigma_t <: x:\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau \quad \vdash \sigma_d <: x:\sigma \xrightarrow[o']{\psi'_+|\psi'_-} \tau' \quad \vdash \sigma_t <: x:\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau \quad \vdash \sigma_d <: x:\sigma \xrightarrow[o']{\psi'_+|\psi'_-} \tau'}{\vdash (\mathbf{Multi} \sigma_t \sigma_d) <: x:\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau \quad \vdash (\mathbf{Multi} \sigma_t \sigma_d) <: x:\sigma \xrightarrow[o']{\psi'_+|\psi'_-} \tau'} \\
\\
\text{S-PMULTI} \quad \text{S-MULTIMONO} \\
\frac{\vdash \sigma <: \sigma' \quad \vdash \tau <: \tau'}{\vdash (\mathbf{Multi} \sigma \tau) <: (\mathbf{Multi} \sigma' \tau')} \quad \vdash (\mathbf{Multi} x:\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau \quad x:\sigma \xrightarrow[o']{\psi'_+|\psi'_-} \tau') <: \mathbf{Multi} \\
\\
\text{S-HMAPP} \quad \text{S-HMAP} \\
\frac{\forall i. \mathcal{M}[k_i] = \sigma_i \text{ and } \vdash \sigma_i <: \tau_i}{\vdash (\mathbf{HMap}^c \mathcal{M} \mathcal{A}') <: (\mathbf{HMap}^p \{k \mapsto \tau\}^i \mathcal{A})} \quad \frac{\forall i. \mathcal{M}[k_i] = \sigma_i \text{ and } \vdash \sigma_i <: \tau_i \quad \mathcal{A}_1 \supseteq \mathcal{A}_2}{\vdash (\mathbf{HMap}^e \mathcal{M} \mathcal{A}_1) <: (\mathbf{HMap}^e \{k \mapsto \tau\}^i \mathcal{A}_2)} \\
\\
\text{S-HMAPMONO} \\
\vdash (\mathbf{HMap}^e \mathcal{M} \mathcal{A}) <: \mathbf{Map}
\end{array}$$

FIGURE A.08. Subtyping rules

$$\begin{array}{l}
\text{JT}(\mathbf{Void}) = \mathbf{nil} \\
\text{JT}(C) = C \\
\text{JT}_{\mathbf{nil}}(\mathbf{Void}) = \mathbf{nil} \\
\text{JT}_{\mathbf{nil}}(C) = (\bigcup \mathbf{nil} C)
\end{array}$$

FIGURE A.09. Java Type Conversion

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

FIGURE A.010. Constant Typing

$\delta(\text{class}, C \overrightarrow{\{fld : v\}})$	$= C$	$\delta(\text{class}, \text{true})$	$= \mathbf{B}$
$\delta(\text{class}, C)$	$= \mathbf{Class}$	$\delta(\text{class}, \text{false})$	$= \mathbf{B}$
$\delta(\text{class}, [\rho, \lambda x^\tau. e]_c)$	$= \mathbf{Fn}$	$\delta(\text{class}, \text{nil})$	$= \text{nil}$
$\delta(\text{class}, [v_d, t]_m)$	$= \mathbf{Multi}$		
$\delta(\text{class}, m)$	$= \mathbf{Map}$	$\delta(n?, n)$	$= \text{true}$
$\delta(\text{class}, k)$	$= \mathbf{K}$	$\delta(n?, e)$	$= \text{false}$
$\delta(\text{class}, n)$	$= \mathbf{N}$	<i>otherwise</i>	

FIGURE A.011. Primitives

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

FIGURE A.012. Definition of isa?

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

FIGURE A.013. Definition of get-method

$$\begin{array}{c}
\text{B-LOCAL} \\
\frac{\rho(x) = v}{\rho \vdash x \Downarrow v} \\
\\
\text{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} \\
\\
\text{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} \\
\\
\text{B-VAL} \\
\rho \vdash v \Downarrow v \\
\\
\text{B-IFTRUE} \\
\frac{\rho \vdash e_1 \Downarrow v_1 \quad v_1 \neq \text{false} \quad v_1 \neq \text{nil} \quad \rho \vdash e_2 \Downarrow v}{\rho \vdash (\text{if } e_1 \ e_2 \ e_3) \Downarrow v} \\
\\
\text{B-IFFALSE} \\
\frac{\rho \vdash e_1 \Downarrow \text{false} \text{ or } \rho \vdash e_1 \Downarrow \text{nil} \quad \rho \vdash e_3 \Downarrow v}{\rho \vdash (\text{if } e_1 \ e_2 \ e_3) \Downarrow v} \\
\\
\text{B-ABS} \\
\rho \vdash \lambda x^\tau. e \Downarrow [\rho, \lambda x^\tau. e]_c \\
\\
\text{B-BETACLOSURE} \\
\frac{\rho \vdash e_f \Downarrow [\rho_c, \lambda x^\tau. e_b]_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} \\
\\
\text{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'} \\
\\
\text{B-BETAMULTI} \\
\frac{\rho \vdash e \Downarrow [v_d, t]_m \quad \rho \vdash e' \Downarrow v' \quad \rho \vdash (v_d \ v') \Downarrow v_e \quad \text{GM}(t, v_e) = v_f \quad \rho \vdash (v_f \ v') \Downarrow v}{\rho \vdash (e \ e') \Downarrow v} \\
\\
\text{B-FIELD} \\
\frac{\rho \vdash e \Downarrow v \quad \text{JVM}_{\text{getstatic}}[C_1, v_1, fld, C_2] = v}{\rho \vdash (. \ e \ fld_{C_2}^{C_1}) \Downarrow v} \\
\\
\text{B-METHOD} \\
\frac{\rho \vdash e_m \Downarrow v_m \quad \rho \vdash e_a \Downarrow v_a \quad \text{JVM}_{\text{invokestatic}}[C_1, v_m, mth, [\vec{C}_a], [\vec{v}_a], C_2] = v}{\rho \vdash (. \ e_m \ (mth_{[[\vec{C}_a], C_2]^{C_1}} \vec{e}_a)) \Downarrow v} \\
\\
\text{B-NEW} \\
\frac{\rho \vdash e_i \Downarrow v_i \quad \text{JVM}_{\text{new}}[C_1, [\vec{C}_i], [\vec{v}_i]] = v}{\rho \vdash (\text{new}_{[\vec{C}_i]} \ C \ \vec{e}_i) \Downarrow v} \\
\\
\text{B-DEFMULTI} \\
\frac{\rho \vdash e \Downarrow v_d \quad v = [v_d, \{\}]_m}{\rho \vdash (\text{defmulti } \tau \ e) \Downarrow v} \\
\\
\text{B-DEFMETHOD} \\
\frac{\rho \vdash e \Downarrow [v_d, t]_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]]_m}{\rho \vdash (\text{defmethod } e \ e' \ e_f) \Downarrow v} \\
\\
\text{B-ISA} \\
\frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e_2 \Downarrow v_2 \quad \text{IsA}(v_1, v_2) = v}{\rho \vdash (\text{isa? } e_1 \ e_2) \Downarrow v} \\
\\
\text{B-ASSOC} \\
\frac{\rho \vdash e \Downarrow m \quad \rho \vdash e_k \Downarrow k \quad \rho \vdash e_v \Downarrow v_v}{\rho \vdash (\text{assoc } e \ e_k \ e_v) \Downarrow m[k \mapsto v_v]} \\
\\
\text{B-GET} \\
\frac{\rho \vdash e \Downarrow m \quad \rho \vdash e' \Downarrow k \quad k \in \text{dom}(m)}{\rho \vdash (\text{get } e \ e') \Downarrow m[k]} \\
\\
\text{B-GETMISSING} \\
\frac{\rho \vdash e \Downarrow m \quad \rho \vdash e' \Downarrow k \quad k \notin \text{dom}(m)}{\rho \vdash (\text{get } e \ e') \Downarrow \text{nil}}
\end{array}$$

FIGURE A.014. Operational Semantics

$$\begin{array}{c}
\text{BS-METHODREFL} \\
\rho \vdash (. e \overrightarrow{mth\ \vec{e}}) \Downarrow \text{wrong}
\end{array}
\qquad
\begin{array}{c}
\text{BS-FIELDREFL} \\
\rho \vdash (. e fld) \Downarrow \text{wrong}
\end{array}
\qquad
\begin{array}{c}
\text{BS-NEWREFL} \\
\rho \vdash (. e fld) \Downarrow \text{wrong}
\end{array}$$

$$\begin{array}{c}
\text{BS-BETA} \\
\rho \vdash e_f \Downarrow v \\
v \neq c \quad v \neq [v_d, t]_m \\
v \neq [\rho_c, \lambda x^T. e_b]_c \\
\hline
\rho \vdash (e_f e_a) \Downarrow \text{wrong}
\end{array}
\qquad
\begin{array}{c}
\text{BS-BETAMULTI} \\
\rho \vdash e_f \Downarrow [v, t]_m \\
v \neq c \quad v \neq [v_d, t]_m \\
v \neq [\rho_c, \lambda x^T. e_b]_c \\
\hline
\rho \vdash (e_f e_a) \Downarrow \text{wrong}
\end{array}
\qquad
\begin{array}{c}
\text{BS-FIELDTARGET} \\
\rho \vdash e \Downarrow v_1 \\
v \neq C_1 \overrightarrow{\{fld_i : v_i\}} \\
\hline
\rho \vdash (. e fld_{C_2}^{C_1}) \Downarrow \text{wrong}
\end{array}$$

$$\begin{array}{c}
\text{BS-FIELDMISSING} \\
\rho \vdash e \Downarrow C_1 \overrightarrow{\{fld_i : v_i\}} \quad fld \notin \overrightarrow{\{fld_i\}} \\
\hline
\rho \vdash (. e fld_{C_2}^{C_1}) \Downarrow \text{wrong}
\end{array}
\qquad
\begin{array}{c}
\text{BS-METHODTARGET} \\
\rho \vdash e_m \Downarrow v \quad v \neq C_1 \overrightarrow{\{fld_i : v_i\}} \\
\hline
\rho \vdash (. e_m (mth_{[[\vec{C}_a], C_2]}^{C_1} \vec{e}_a)) \Downarrow \text{wrong}
\end{array}$$

$$\begin{array}{c}
\text{BS-METHODARITY} \\
i \neq a \\
\hline
\rho \vdash (. e_m (mth_{[[\vec{C}_i], C_2]}^{C_1} \vec{e}_a)) \Downarrow \text{wrong}
\end{array}
\qquad
\begin{array}{c}
\text{BS-METHODARG} \\
\rho \vdash e_m \Downarrow v_m \quad \rho \vdash e_a \Downarrow v_a \\
\exists a. v_a \neq C_a \overrightarrow{\{fld_i : v_i\}} \text{ or } v_a \neq \text{nil} \\
\hline
\rho \vdash (. e_m (mth_{[[\vec{C}_a], C_2]}^{C_1} \vec{e}_a)) \Downarrow \text{wrong}
\end{array}$$

$$\begin{array}{c}
\text{BS-NEWARG} \\
\rho \vdash e_i \Downarrow v_i \\
\exists i. v_i \neq C_i \overrightarrow{\{fld_i : v_i\}} \text{ or } v_i \neq \text{nil} \\
\hline
\rho \vdash (\text{new}_{[\vec{C}_i]} C \vec{e}_i) \Downarrow \text{wrong}
\end{array}
\qquad
\begin{array}{c}
\text{BS-NEWARITY} \\
i \neq a \\
\hline
\rho \vdash (\text{new}_{[\vec{C}_i]} C \vec{e}_a) \Downarrow \text{wrong}
\end{array}$$

$$\begin{array}{c}
\text{BS-ASSOCMAP} \\
\rho \vdash e_m \Downarrow v \quad v \neq \overrightarrow{\{(v_a v_b)\}} \\
\hline
\rho \vdash (\text{assoc } e_m e_k e_v) \Downarrow \text{wrong}
\end{array}
\qquad
\begin{array}{c}
\text{BS-ASSOCKEY} \\
\rho \vdash e_m \Downarrow \overrightarrow{\{(v_a v_b)\}} \quad \rho \vdash e_k \Downarrow v_k \\
v_k \neq k \\
\hline
\rho \vdash (\text{assoc } e_m e_k e_v) \Downarrow \text{wrong}
\end{array}$$

$$\begin{array}{c}
\text{BS-GETMAP} \\
\rho \vdash e_m \Downarrow v \quad v \neq \overrightarrow{\{(v_a v_b)\}} \\
\hline
\rho \vdash (\text{get } e_m e_k) \Downarrow \text{wrong}
\end{array}
\qquad
\begin{array}{c}
\text{BS-GETKEY} \\
\rho \vdash e_m \Downarrow v \quad \rho \vdash e_k \Downarrow v_k \\
v \neq k \\
\hline
\rho \vdash (\text{get } e_m e_k) \Downarrow \text{wrong}
\end{array}
\qquad
\begin{array}{c}
\text{BS-LOCAL} \\
x \notin \text{dom}(\rho) \\
\hline
\rho \vdash x \Downarrow \text{wrong}
\end{array}$$

$$\begin{array}{c}
\text{BS-DEFMETHOD} \\
\rho \vdash e_m \Downarrow v_m \quad v_m \neq [v_d, t]_m \\
\hline
\rho \vdash (\text{defmethod } e_m e_v e_f) \Downarrow \text{wrong}
\end{array}$$

FIGURE A.015. Stuck programs

$$\begin{array}{c}
\text{BE-ERRORWRONG} \\
\rho \vdash \beta \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-LET} \\
\rho \vdash e_a \Downarrow \beta \\
\hline
\rho \vdash (\text{let } [x \ e_a] \ e) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-DO1} \\
\rho \vdash e_1 \Downarrow \beta \\
\hline
\rho \vdash (\text{do } e_1 \ e) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-DO2} \\
\rho \vdash e_1 \Downarrow v_1 \\
\rho \vdash e \Downarrow \beta \\
\hline
\rho \vdash (\text{do } e_1 \ e) \Downarrow \beta
\end{array}$$

$$\begin{array}{c}
\text{BE-IF} \\
\rho \vdash e_1 \Downarrow \beta \\
\hline
\rho \vdash (\text{if } e_1 \ e_2 \ e_3) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-IFTRUE} \\
\rho \vdash e_1 \Downarrow v_1 \\
v_1 \neq \text{false} \quad v_1 \neq \text{nil} \\
\rho \vdash e_2 \Downarrow \beta \\
\hline
\rho \vdash (\text{if } e_1 \ e_2 \ e_3) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-IFFALSE} \\
\rho \vdash e_1 \Downarrow \text{false} \quad \text{or} \quad \rho \vdash e_1 \Downarrow \text{nil} \\
\rho \vdash e_3 \Downarrow \beta \\
\hline
\rho \vdash (\text{if } e_1 \ e_2 \ e_3) \Downarrow \beta
\end{array}$$

$$\begin{array}{c}
\text{BE-BETA1} \\
\rho \vdash e_f \Downarrow \beta \\
\hline
\rho \vdash (e_f \ e_a) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-BETA2} \\
\rho \vdash e_f \Downarrow v_f \\
\rho \vdash e_a \Downarrow \beta \\
\hline
\rho \vdash (e_f \ e_a) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-BETACLOSURE} \\
\rho \vdash e_f \Downarrow [\rho_c, \lambda x^\tau. e_b]_c \\
\rho \vdash e_a \Downarrow v_a \\
\rho_c[x \mapsto v_a] \vdash e_b \Downarrow \beta \\
\hline
\rho \vdash (e_f \ e_a) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-BETAMULTI1} \\
\rho \vdash e_f \Downarrow [v_d, m]_m \\
\rho \vdash e_a \Downarrow v_a \\
\rho \vdash (v_d \ v_a) \Downarrow \beta \\
\hline
\rho \vdash (e_f \ e_a) \Downarrow \beta
\end{array}$$

$$\begin{array}{c}
\text{BE-BETAMULTI2} \\
\rho \vdash e_f \Downarrow [v_d, m]_m \\
\rho \vdash e_a \Downarrow v_a \\
\rho \vdash (v_d \ v_a) \Downarrow v_e \\
\text{GM}(t, v_e) = \text{err} \\
\hline
\rho \vdash (e_f \ e_a) \Downarrow \text{err}
\end{array}
\qquad
\begin{array}{c}
\text{BE-DELTA} \\
\rho \vdash e \Downarrow c \\
\rho \vdash e' \Downarrow v \\
\delta(c, v) = \beta \\
\hline
\rho \vdash (e \ e') \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-FIELD} \\
\rho \vdash e \Downarrow \beta \\
\hline
\rho \vdash (. \ e \ \text{fld}_{C_2}^{C_1}) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-METHOD1} \\
\rho \vdash e_m \Downarrow \beta \\
\hline
\rho \vdash (. \ e_m \ (\text{mth}_{[[C_a], C_2]}^{C_1} \vec{e})) \Downarrow \beta
\end{array}$$

$$\begin{array}{c}
\text{BE-METHOD2} \\
\rho \vdash e_m \Downarrow v_m \\
\rho \vdash e_{n-1} \Downarrow v_{n-1} \\
\rho \vdash e_n \Downarrow \beta \\
\hline
\rho \vdash (. \ e_m \ (\text{mth}_{[[C_a], C_2]}^{C_1} \vec{e})) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-METHOD3} \\
\rho \vdash e_m \Downarrow v_m \quad \rho \vdash e_a \Downarrow v_a \\
\text{JVM}_{\text{invokestatic}}[C_1, v_m, \text{mth}, [C_a], [\vec{v}_a], C_2] = \text{err} \\
\hline
\rho \vdash (. \ e_m \ (\text{mth}_{[[C_a], C_2]}^{C_1} \vec{e}_a)) \Downarrow \text{err}
\end{array}$$

$$\begin{array}{c}
\text{BE-NEW1} \\
\rho \vdash e_{n-1} \Downarrow v_{n-1} \\
\rho \vdash e_n \Downarrow \beta \\
\hline
\rho \vdash (\text{new}_{[C_i]} \vec{C} \vec{e}) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-NEW2} \\
\rho \vdash e_i \Downarrow v_i \\
\text{JVM}_{\text{new}}[C_1, [C_i], [\vec{v}_i]] = \text{err} \\
\hline
\rho \vdash (\text{new}_{[C_i]} \vec{C} \vec{e}_i) \Downarrow \text{err}
\end{array}
\qquad
\begin{array}{c}
\text{BE-DEFMULTI} \\
\rho \vdash e_d \Downarrow \beta \\
\hline
\rho \vdash (\text{defmulti } \tau \ e_d) \Downarrow \beta
\end{array}$$

$$\begin{array}{c}
\text{BE-DEFMETHOD1} \\
\rho \vdash e_m \Downarrow \beta \\
\hline
\rho \vdash (\text{defmethod } e_m \ e_v \ e_f) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-DEFMETHOD2} \\
\rho \vdash e_m \Downarrow [v_d, t]_m \\
\rho \vdash e_v \Downarrow \beta \\
\hline
\rho \vdash (\text{defmethod } e_m \ e_v \ e_f) \Downarrow \beta
\end{array}
\qquad
\begin{array}{c}
\text{BE-DEFMETHOD3} \\
\rho \vdash e_m \Downarrow [v_d, t]_m \\
\rho \vdash e_v \Downarrow v_v \\
\rho \vdash e_f \Downarrow \beta \\
\hline
\rho \vdash (\text{defmethod } e_m \ e_v \ e_f) \Downarrow \beta
\end{array}$$

FIGURE A.016. Error and stuck propagation (continued in Figure A.017)

$$\begin{array}{c}
\text{BE-ISA1} \\
\frac{\rho \vdash e_1 \Downarrow \beta}{\rho \vdash (\text{isa? } e_1 e_2) \Downarrow \beta} \\
\\
\text{BE-ISA2} \\
\frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e_2 \Downarrow \beta}{\rho \vdash (\text{isa? } e_1 e_2) \Downarrow \beta} \\
\\
\text{BE-ASSOC1} \\
\frac{\rho \vdash e_m \Downarrow \beta}{\rho \vdash (\text{assoc } e_m e_k e_v) \Downarrow \beta} \\
\\
\text{BE-ASSOC2} \\
\frac{\rho \vdash e_m \Downarrow \{(v_a v_b)\} \quad \rho \vdash e_k \Downarrow \beta}{\rho \vdash (\text{assoc } e_m e_k e_v) \Downarrow \beta} \\
\\
\text{BE-ASSOC3} \\
\frac{\rho \vdash e_m \Downarrow \{(v_a v_b)\} \quad \rho \vdash e_k \Downarrow v_k \quad \rho \vdash e_v \Downarrow \beta}{\rho \vdash (\text{assoc } e_m e_k e_v) \Downarrow \beta} \\
\\
\text{BE-GET1} \\
\frac{\rho \vdash e_m \Downarrow \beta}{\rho \vdash (\text{get } e_m e_k) \Downarrow \beta} \\
\\
\text{BE-GET2} \\
\frac{\rho \vdash e_m \Downarrow \{(v_a v_b)\} \quad \rho \vdash e_k \Downarrow \beta}{\rho \vdash (\text{get } e_m e_k) \Downarrow \beta}
\end{array}$$

FIGURE A.017. Error and stuck propagation (continued from Figure A.016)

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

FIGURE A.018. Path translation

$$\begin{array}{lcl}
\text{update}((\bigcup \vec{\tau}), \nu, \pi) & = & (\bigcup \overrightarrow{\text{update}(\tau, \nu, \pi)}) \\
\text{update}(\tau, (\mathbf{Val } C), \pi :: \mathbf{class}) & = & \text{update}(\tau, C, \pi) \\
\text{update}(\tau, \nu, \pi :: \mathbf{class}) & = & \tau \\
\text{update}((\mathbf{HMap}^{\mathcal{E}} \mathcal{M} \mathcal{A}), \nu, \pi :: \mathbf{key}_k) & = & (\mathbf{HMap}^{\mathcal{E}} \mathcal{M}[k \mapsto \text{update}(\tau, \nu, \pi)] \mathcal{A}) \\
& & \text{if } \mathcal{M}[k] = \tau \\
\text{update}((\mathbf{HMap}^{\mathcal{E}} \mathcal{M} \mathcal{A}), \nu, \pi :: \mathbf{key}_k) & = & \perp \quad \text{if } \vdash \mathbf{nil} \not\prec: \nu \text{ and } k \in \mathcal{A} \\
\text{update}((\mathbf{HMap}^{\mathcal{P}} \mathcal{M} \mathcal{A}), \tau, \pi :: \mathbf{key}_k) & = & (\bigcup (\mathbf{HMap}^{\mathcal{P}} \mathcal{M}[k \mapsto \tau] \mathcal{A}) \\
& & (\mathbf{HMap}^{\mathcal{P}} \mathcal{M} (\mathcal{A} \cup \{k\}))) \\
& & \text{if } \vdash \mathbf{nil} \prec: \tau, k \notin \text{dom}(\mathcal{M}) \text{ and } k \notin \mathcal{A} \\
\text{update}((\mathbf{HMap}^{\mathcal{P}} \mathcal{M} \mathcal{A}), \nu, \pi :: \mathbf{key}_k) & = & (\mathbf{HMap}^{\mathcal{P}} \mathcal{M}[k \mapsto \text{update}(\tau, \nu, \pi)] \mathcal{A}) \\
& & \text{if } \vdash \mathbf{nil} \not\prec: \nu, k \notin \text{dom}(\mathcal{M}) \text{ and } k \notin \mathcal{A} \\
\text{update}(\tau, \nu, \pi :: \mathbf{key}_k) & = & \tau \\
\text{update}(\tau, \sigma, \epsilon) & = & \text{restrict}(\tau, \sigma) \\
\text{update}(\tau, \bar{\sigma}, \epsilon) & = & \text{remove}(\tau, \sigma) \\
\\
\text{restrict}(\tau, \sigma) & = & \perp \quad \text{if } \nexists v. \vdash v : \tau ; \psi ; o \text{ and } \vdash v : \sigma ; \psi' ; o' \\
\text{restrict}(\tau, \sigma) & = & \tau \quad \text{if } \vdash \tau \prec: \sigma \\
\text{restrict}(\tau, \sigma) & = & \sigma \quad \text{otherwise} \\
\\
\text{remove}(\tau, \sigma) & = & \perp \quad \text{if } \vdash \tau \prec: \sigma \\
\text{remove}(\tau, \sigma) & = & \tau \quad \text{otherwise}
\end{array}$$

FIGURE A.019. Type Update

$$\begin{array}{c}
\text{M-OR} \\
\frac{\rho \models \psi_1 \text{ or } \rho \models \psi_2}{\rho \models \psi_1 \vee \psi_2}
\end{array}
\quad
\begin{array}{c}
\text{M-IMP} \\
\frac{\rho \models \psi_1 \text{ implies } \rho \models \psi_2}{\rho \models \psi_1 \supset \psi_2}
\end{array}
\quad
\begin{array}{c}
\text{M-AND} \\
\frac{\rho \models \psi_1 \quad \rho \models \psi_2}{\rho \models \psi_1 \wedge \psi_2}
\end{array}
\quad
\begin{array}{c}
\text{M-TOP} \\
\rho \models \text{tt}
\end{array}$$

$$\begin{array}{c}
\text{M-TYPE} \\
\frac{\vdash \rho(\pi(x)) : \tau ; \psi_+ | \psi_- ; o}{\rho \models \tau_{\pi(x)}}
\end{array}$$

$$\begin{array}{c}
\text{M-NOTTYPE} \\
\frac{\vdash \rho(\pi(x)) : \sigma ; \psi_+ | \psi_- ; o \text{ and } \vdash v : \sigma ; \psi_{2+} | \psi_{2-} ; o_2}{\rho \models \bar{\tau}_{\pi(x)}}
\end{array}$$

FIGURE A.020. Satisfaction Relation

$$\begin{array}{c}
\text{L-ATOM} \\
\frac{\psi \in \Gamma}{\Gamma \vdash \psi}
\end{array}
\quad
\begin{array}{c}
\text{L-TRUE} \\
\Gamma \vdash \text{tt}
\end{array}
\quad
\begin{array}{c}
\text{L-FALSE} \\
\Gamma \vdash \text{ff} \\
\hline
\Gamma \vdash \psi
\end{array}
\quad
\begin{array}{c}
\text{L-ANDI} \\
\frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \wedge \psi_2}
\end{array}
\quad
\begin{array}{c}
\text{L-ANDE} \\
\frac{\Gamma, \psi_1, \psi_2 \vdash \psi}{\Gamma, \psi_1 \wedge \psi_2 \vdash \psi}
\end{array}
\quad
\begin{array}{c}
\text{L-IMPLI} \\
\frac{\Gamma, \psi_1 \vdash \psi_2}{\Gamma \vdash \psi_1 \supset \psi_2}
\end{array}$$

$$\begin{array}{c}
\text{L-IMPLE} \\
\frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_1 \supset \psi_2}{\Gamma \vdash \psi_2}
\end{array}
\quad
\begin{array}{c}
\text{L-ORI} \\
\frac{\Gamma \vdash \psi_1 \text{ or } \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \vee \psi_2}
\end{array}
\quad
\begin{array}{c}
\text{L-ORE} \\
\frac{\Gamma, \psi_1 \vdash \psi \quad \Gamma, \psi_2 \vdash \psi}{\Gamma, \psi_1 \vee \psi_2 \vdash \psi}
\end{array}
\quad
\begin{array}{c}
\text{L-SUB} \\
\frac{\Gamma \vdash \tau_{\pi(x)} \quad \vdash \tau <: \sigma}{\Gamma \vdash \sigma_{\pi(x)}}
\end{array}$$

$$\begin{array}{c}
\text{L-SUBNOT} \\
\frac{\Gamma \vdash \bar{\sigma}_{\pi(x)} \quad \vdash \tau <: \sigma}{\Gamma \vdash \bar{\tau}_{\pi(x)}}
\end{array}
\quad
\begin{array}{c}
\text{L-BOT} \\
\frac{\Gamma \vdash \perp_{\pi(x)}}{\Gamma \vdash \psi}
\end{array}
\quad
\begin{array}{c}
\text{L-UPDATE} \\
\frac{\Gamma \vdash \tau_{\pi'(x)} \quad \Gamma \vdash \nu_{\pi(\pi'(x))}}{\Gamma \vdash \text{update}(\tau, \nu, \pi)_{\pi'(x)}}
\end{array}$$

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

FIGURE A.021. Proof System

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

Substitution on types is capture-avoiding structural recursion.

FIGURE A.022. Substitution

APPENDIX B

Soundness for λ_{TC}

ASSUMPTION B.01 (JVM_{new}). If $\forall i. v_i = C_i \{\overrightarrow{fld_j : v_j}\}$ or $v_i = \text{nil}$ and v_i is consistent with ρ then either

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

ASSUMPTION B.02 ($\text{JVM}_{\text{getstatic}}$). If $v_1 = C_1 \{\overrightarrow{fld : v_f, fld_l : v_l}\}$, then either

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

ASSUMPTION B.03 ($\text{JVM}_{\text{invokestatic}}$). If $v_1 = C_1 \{\overrightarrow{fld_l : v_l}\}$, $\forall i. v_i = C_i \{\overrightarrow{fld_j : v_j}\}$ or $v_i = \text{nil}$ then either

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

LEMMA B.01. If ρ and ρ' agree on $\text{fv}(\psi)$ and $\rho \models \psi$ then $\rho' \models \psi$.

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

LEMMA B.02. If

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

then $\rho_1 \models \psi_1$.

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

LEMMA B.03. If $\rho \models \Gamma$ and $\Gamma \vdash \psi$ then $\rho \models \psi$.

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

LEMMA B.04. If $\Gamma \vdash \tau_{\pi(x)}$, $\rho \models \Gamma$ and $\rho(\pi(x)) = v$ then $\vdash v : \tau$; $\psi'_+ | \psi'_-$; o' for some ψ'_+ , ψ'_- and o' .

PROOF. Corollary of lemma B.03. □

LEMMA B.05 (Paths are independent). *If $\rho(o) = \rho_1(o')$ then $\rho(\pi(o)) = \rho_1(\pi(o'))$*

PROOF. By induction on π . □

LEMMA B.06 (*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 B.01 (Consistent with). *v is consistent with ρ iff*

- $\forall [\rho_1, \lambda x^\sigma.e]_c$ in v , if $\vdash [\rho_1, \lambda x^\sigma.e]_c : \tau$; $\text{tt} \Vdash \text{ff}$; \emptyset , and $\forall o'$ in τ , either $o' = \emptyset$, or $o' = \pi'(x)$, or $\rho(o') = \rho_1(o')$.

DEFINITION B.02. *ρ is consistent iff*

$\forall v \in \text{rng}(\rho)$, *v is consistent with ρ .*

DEFINITION B.03 (TrueVal). *TrueVal(v) iff $v \neq \text{false}$ and $v \neq \text{nil}$.*

DEFINITION B.04 (FalseVal). *FalseVal(v) iff $v = \text{false}$ or $v = \text{nil}$.*

LEMMA B.07 (*isa?* has correct propositions). *If*

- $\Gamma \vdash v_1 : \tau_1$; $\psi_{1+} | \psi_{1-}$; o_1 ,
- $\Gamma \vdash v_2 : \tau_2$; $\psi_{2+} | \psi_{2-}$; o_2 ,
- $\text{IsA}(v_1, v_2) = v$,
- $\rho \models \Gamma$,
- $\text{IsAProps}(o_1, \tau_2) = \psi'_+ | \psi'_-$,
- $\psi'_+ \vdash \psi_+$, and
- $\psi'_- \vdash \psi_-$,

then either

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

PROOF. By cases on the definition of *IsA* and subcases on *IsA*.

SUBCASE ($\text{IsA}(v_1, v_1) = \text{true}$, if $v_1 \neq C$).

$v_1 = v_2$, $v_1 \neq C$, $v_2 \neq C$, TrueVal(v)

Since TrueVal(v) we prove $\rho \models \psi_+$ by cases on the definition of *IsAProps*:

SUBCASE ($\text{IsAProps}(\text{class}(\pi(x)), (\mathbf{Val} C)) = C_{\pi(x)} | \overline{C}_{\pi(x)}$).

$o_1 = \text{class}(\pi(x))$, $\tau_2 = (\mathbf{Val} C)$, $C_{\pi(x)} \vdash \psi_+$

Unreachable by inversion on the typing relation, since $\tau_2 = (\mathbf{Val} C)$, yet $v_2 \neq C$.

SUBCASE ($\text{IsAProps}(o, (\mathbf{Val} l)) = ((\mathbf{Val} l)_x | \overline{(\mathbf{Val} l)}_x)[o/x]$ 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 $\psi_+ = \text{tt}$ and we derive $\rho \models \text{tt}$ 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 ($\text{IsAProps}(o, \tau) = \text{tt}|\text{tt}$).

$\psi_+ = \text{tt}$

$\rho \models \text{tt}$ holds by M-Top.

SUBCASE ($\text{IsA}(C_1, C_2) = \text{true}$, if $\vdash C_1 <: C_2$).

$v_1 = C_1, v_2 = C_2, \vdash C_1 <: C_2, \text{TrueVal}(v)$

Since $\text{TrueVal}(v)$ we prove $\rho \models \psi_+$ by cases on the definition of IsAProps :

SUBCASE ($\text{IsAProps}(\text{class}(\pi(x)), (\mathbf{Val} C)) = C_{\pi(x)}|\overline{C}_{\pi(x)}$).

$o_1 = \text{class}(\pi(x)), \tau_2 = (\mathbf{Val} C_2), C_{2\pi(x)} \vdash \psi_+$

By inversion on the typing relation, since **class** is the last path element of o_1 then $\rho \vdash (\text{class } \rho(\pi(x))) \Downarrow v_1$.

Since $\rho \vdash (\text{class } \rho(\pi(x))) \Downarrow C_1$, as $v_1 = C_1$, we can derive from lemma B.06 $\rho \models C_{1\pi(x)}$.

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 B.03 we derive $\rho \models C_{2\pi(x)}$.

SUBCASE ($\text{IsAProps}(o, (\mathbf{Val} l)) = ((\mathbf{Val} l)_x|\overline{(\mathbf{Val} l)}_x)[o/x]$ 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 ($\text{IsAProps}(o, \tau) = \text{tt}|\text{tt}$).

$\psi_+ = \text{tt}$

$\rho \models \text{tt}$ holds by M-Top.

SUBCASE ($\text{IsA}(v_1, v_2) = \text{false}$, otherwise).

$v_1 \neq v_2, \text{FalseVal}(v)$

Since $\text{FalseVal}(v)$ we prove $\rho \models \psi_-$ by cases on the definition of IsAProps :

SUBCASE ($\text{IsAProps}(\text{class}(\pi(x)), (\mathbf{Val} C)) = C_{\pi(x)}|\overline{C}_{\pi(x)}$).

$o_1 = \text{class}(\pi(x)), \tau_2 = (\mathbf{Val} C), \overline{C}_{\pi(x)} \vdash \psi_-$

By inversion on the typing relation, since **class** is the last path element of o_1 then $\rho \vdash (\text{class } \rho(\pi(x))) \Downarrow v_1$.

By the definition of *class* either $v_1 = C$ or $v_1 = \text{nil}$.

If $v_1 = \text{nil}$, then we know from the definition of **IsA** that $\rho(\pi(x)) = \text{nil}$.

Since $\vdash \rho(\pi(x)) : \text{nil}$, and there is no v_1 such that both $\vdash \rho(\pi(x)) : C$ and $\vdash \rho(\pi(x)) : \text{nil}$, we use M-NotType to derive $\rho \models \overline{C}_{\pi(x)}$.

Similarly if $v_1 = C_1$, by the definition of 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 ($\text{IsAProps}(o, (\mathbf{Val} l)) = ((\mathbf{Val} l)_x|\overline{(\mathbf{Val} l)}_x)[o/x]$ 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.

If $o_1 = \emptyset$ then $\psi_- = \text{tt}$ and we derive $\rho \models \text{tt}$ 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 (IsAProps(o, τ) = $\mathbb{t}\mathbb{t}|\mathbb{t}\mathbb{t}$).

$\psi_- = \mathbb{t}\mathbb{t}$

$\rho \models \mathbb{t}\mathbb{t}$ holds by M-Top.

□

LEMMA B.08. *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 $\mathbf{TrueVal}(v)$ and $\rho \models \psi_+$ or $\mathbf{FalseVal}(v)$ and $\rho \models \psi_-$,
 - (3) $\vdash v : \tau$; $\psi'_+|\psi'_-$; o' for some ψ'_+, ψ'_- and o' , and
 - (4) v is consistent with ρ , or
- $\rho \vdash e \Downarrow \mathbf{err}$.

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

CASE (B-Val).

SUBCASE (T-True). $v = \mathbf{true}$, $e' = \mathbf{true}$, $e = \mathbf{true}$, $\vdash \mathbf{true} <: \tau$, $\mathbb{t}\mathbb{t} \vdash \psi_+$, $\mathbf{ff} \vdash \psi_-$, $\vdash \emptyset <: o$

Proving part 1 is trivial: o is a superobject of \emptyset , which can only be \emptyset .

To prove part 2, we note that $v = \mathbf{true}$ and $\mathbb{t}\mathbb{t} \vdash \psi_+$, 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). $v = \overrightarrow{\{v_k \mapsto v_v\}}$, $e' = \overrightarrow{\{v_k \mapsto v_v\}}$, $e = \overrightarrow{\{v_k \mapsto v_v\}}$, $\vdash (\mathbf{HMap}^C \mathcal{M}) <: \tau$, $\mathbb{t}\mathbb{t} \vdash \psi_+$, $\mathbf{ff} \vdash \psi_-$, $\vdash \emptyset <: o$, $\vdash v_k : (\mathbf{Val} k)$, $\vdash v_v : \tau_v$, $\mathcal{M} = \{k \mapsto \tau_v\}$

Similar to T-True.

Part 4 holds by the induction hypothesis on $\overrightarrow{v_k}$ and $\overrightarrow{v_v}$.

SUBCASE (T-Kw). $v = k$, $e' = k$, $e = k$, $\vdash (\mathbf{Val} k) <: \tau$, $\mathbb{t}\mathbb{t} \vdash \psi_+$, $\mathbf{ff} \vdash \psi_-$, $\vdash \emptyset <: o$

Similar to T-True.

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

SUBCASE (T-False). $v = \mathbf{false}$, $e' = \mathbf{false}$, $e = \mathbf{false}$, $\vdash \mathbf{false} <: \tau$, $\mathbf{ff} \vdash \psi_+$, $\mathbb{t}\mathbb{t} \vdash \psi_-$, $\vdash \emptyset <: o$

Proving part 1 is trivial: o is a superobject of \emptyset , which must be \emptyset .

To prove part 2, we note that $v = \mathbf{false}$ and $\mathbb{t}\mathbb{t} \vdash \psi_-$, 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). $v = C$, $e' = C$, $e = C$, $\vdash (\mathbf{Val} C) <: \tau$, $\mathbb{t}\mathbb{t} \vdash \psi_+$, $\mathbf{ff} \vdash \psi_-$, $\vdash \emptyset <: o$

Similar to T-True.

SUBCASE (T-Instance). $v = C \{\overrightarrow{fld_i : v_i}\}, e' = C \{\overrightarrow{fld : v}\}, e = C \{\overrightarrow{fld : v}\}, \vdash C <: \tau, \mathbb{tt} \vdash \psi_+, \mathbb{ff} \vdash \psi_-, \vdash \emptyset <: o$

Similar to T-True.

Part 4 holds by the induction hypotheses on $\overrightarrow{v'_i}$.

SUBCASE (T-Nil). $v = \text{nil}, e' = \text{nil}, e = \text{nil}, \vdash \mathbf{nil} <: \tau, \mathbb{ff} \vdash \psi_+, \mathbb{tt} \vdash \psi_-, \vdash \emptyset <: o$

Similar to T-False.

SUBCASE (T-Multi). $v = [v_1, \{\overrightarrow{v_k \mapsto v'_v}\}]_m, e' = [v_1, \{\overrightarrow{v_k \mapsto v'_v}\}]_m, \vdash v_1 \Rightarrow v_1 : \tau_1, \vdash \overrightarrow{v_k \Rightarrow v_k : \tau}, \vdash \overrightarrow{v_v \Rightarrow v_v : \sigma}, e = [v_1, \{\overrightarrow{v_k \mapsto v'_v}\}]_m, \vdash (\mathbf{Multi} \sigma \tau_1) <: \tau, \mathbb{tt} \vdash \psi_+, \mathbb{ff} \vdash \psi_-, \vdash \emptyset <: o$

Similar to T-True.

SUBCASE (T-Const). $e = c, \vdash \delta_\tau(c) <: \tau, \mathbb{tt} \vdash \psi_+, \mathbb{ff} \vdash \psi_-, \vdash \emptyset <: o$

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). $e' = x, e = x, \overline{(\cup \mathbf{nil} \mathbf{false})}_x \vdash \psi_+, (\cup \mathbf{nil} \mathbf{false})_x \vdash \psi_-, \vdash x <: o, \Gamma \vdash \tau_x$

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 $\mathbf{TrueVal}(v)$, then $\rho \models \overline{(\cup \mathbf{nil} \mathbf{false})}_x$ holds by M-NotType; if $\mathbf{FalseVal}(v)$, then $\rho \models (\cup \mathbf{nil} \mathbf{false})_x$ 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). $e' = (\mathbf{do} e'_1 e'_2), \Gamma \vdash e'_1 : \tau_1; \psi_{1+} | \psi_{1-}; o_1, \Gamma, \psi_{1+} \vee \psi_{1-} \vdash e' : \tau; \psi_+ | \psi_-; o, e = (\mathbf{do} e_1 e_2)$

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

Parts 3 and 4 follow from the induction hypothesis on e_2 .

CASE (BE-Do1). $\rho \vdash e_1 \Downarrow \text{err}, \rho \vdash e \Downarrow \text{err}$

Trivially reduces to an error.

CASE (BE-Do2). $\rho \vdash e_1 \Downarrow v_1, \rho \vdash e_2 \Downarrow \text{err}, \rho \vdash e \Downarrow \text{err}$

As above.

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

SUBCASE (T-New). $e' = (\text{new } C \vec{e}_i), [\vec{C}_i] \in \mathcal{CT}[C][c], \overline{\text{JT}_{\text{nil}}(C_i)} = \tau_i, \Gamma \vdash e'_i \Rightarrow e_i : \tau_i, e = (\text{new}_{[\vec{C}_i]} C \vec{e}_i), \vdash \text{JT}(C) <: \tau, \text{tt} \vdash \psi_+, \text{ff} \vdash \psi_-, \vdash \emptyset <: o$

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 $v \neq \text{false}$ (or `nil`) where $\psi_+ = \text{tt}$ so $\rho \models \psi_+$. `Void` also lacks a constructor.

Part 3 holds as B-New reduces to a *non-nilable* instance of C via JVM_{new} (by assumption B.01), and τ is a supertype of $\text{JT}(C)$.

SUBCASE (T-NewStatic). $e' = (\text{new}_{[\vec{C}_i]} C \vec{e}_i)$

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). $\overline{\rho \vdash e_{i-1} \Downarrow v_{i-1}, \rho \vdash e_i \Downarrow \text{err}, \rho \vdash e \Downarrow \text{err}}$

Trivially reduces to an error.

CASE (BE-New2). $\overline{\rho \vdash e_i \Downarrow v_i, \text{JVM}_{\text{new}}[C_1, [\vec{C}_i], [\vec{v}_i]] = \text{err}, \rho \vdash e \Downarrow \text{err}}$

As above.

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

SUBCASE (T-Field). $e' = (. e'_1 fld), \Gamma \vdash e' \Rightarrow e : \sigma, \vdash \sigma <: \mathbf{Object}, \text{TJ}(\sigma) = C_1, fld \mapsto C_2 \in \mathcal{CT}[C_1][f], e = (. e_1 fld_{C_2}^{C_1}) \vdash \text{JT}_{\text{nil}}(C_2) <: \tau, \text{tt} \vdash \psi_+, \text{tt} \vdash \psi_-, \vdash \emptyset <: o$

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 `tt`.

Part 3 relies on the semantics of $\text{JVM}_{\text{getstatic}}$ (assumption B.02) in B-Field, which returns a *nilable* instance of C_2 , and τ is a supertype of $\text{JT}_{\text{nil}}(C_2)$. Notice $\vdash \sigma <: \mathbf{Object}$ is required to guard from dereferencing `nil`, as C_1 erases occurrences of `nil` in σ via $\text{TJ}(\sigma) = C_1$.

SUBCASE (T-FieldStatic). $e' = (. e_1 fld_{C_2}^{C_1})$

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). $\rho \vdash e_1 \Downarrow \text{err}, \rho \vdash e \Downarrow \text{err}$

Trivially reduces to an error.

CASE (B-Method). $\overline{\rho \vdash e_m \Downarrow v_m, \rho \vdash e_a \Downarrow v_a, \text{JVM}_{\text{invokestatic}}[C_1, v_m, mth, [\vec{C}_a], [\vec{v}_a], C_2] = v}$

SUBCASE (T-Method). $\Gamma \vdash e' \Rightarrow e : \sigma, \vdash \sigma <: \mathbf{Object}, \text{TJ}(\sigma) = C_1, mth \mapsto [[\vec{C}_i], C_2] \in \mathcal{CT}[C_1][m], \overline{\text{JT}_{\text{nil}}(C_i)} = \tau_i, \overline{\Gamma \vdash e'_i \Rightarrow e_i : \tau_i}, e = (. e_m (mth_{[[\vec{C}_i], C_2]}^{C_1} \vec{e}_a)), \vdash \text{JT}_{\text{nil}}(C_2) <: \tau, \text{tt} \vdash \psi_+, \text{tt} \vdash \psi_-, \vdash \emptyset <: o$

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 $\text{JVM}_{\text{invokestatic}}$ (assumption B.03) in B-Method, which returns a *nilable* instance of C_2 , and τ is a supertype of $\text{JT}_{\text{nil}}(C_2) = \cdot$. Notice $\vdash \sigma <: \mathbf{Object}$ is required to guard from dereferencing nil, as C_1 erases occurrences of **nil** in σ via $\text{TJ}(\sigma) = C_1$.

SUBCASE (T-MethodStatic). $e' = (\cdot e_1 (\text{mth}_{[[\vec{C}_1], C_2]}^{C_1} \vec{e}_i))$

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). $\rho \vdash e_m \Downarrow \text{err}, \rho \vdash e \Downarrow \text{err}$

Trivially reduces to an error.

CASE (BE-Method2). $\rho \vdash e_m \Downarrow v_m, \rho \vdash e_{n-1} \Downarrow v_{n-1}, \rho \vdash e_n \Downarrow \text{err}, \rho \vdash e \Downarrow \text{err}$

As above.

CASE (BE-Method3). $\rho \vdash e_m \Downarrow v_m, \rho \vdash e_a \Downarrow v_a, \text{JVM}_{\text{invokestatic}}[C_1, v_m, \text{mth}, [\vec{C}_a], [\vec{v}_a], C_2] = \text{err}, \rho \vdash e \Downarrow \text{err}$

As above.

CASE (B-DefMulti). $v = [v_d, \{\}]_m, \rho \vdash e_d \Downarrow v_d$

SUBCASE (T-DefMulti). $e' = (\text{defmulti } \sigma e'_d), \sigma = x:\tau_1 \xrightarrow{o_1^{\psi_{1+}|\psi_{1-}}} \tau_2, \tau_d = x:\tau_1 \xrightarrow{o_2^{\psi_{2+}|\psi_{2-}}} \tau_3,$
 $\Gamma \vdash e' \Rightarrow e : \sigma', e = (\text{defmulti } \sigma e_d), \vdash (\mathbf{Multi} \sigma \tau_d) <: \tau, \mathbb{tt} \vdash \psi_+, \text{ff} \vdash \psi_-, \vdash \emptyset <: o$

Part 1 and 2 hold for the same reasons as T-True. For part 3 we show $\vdash [v_d, \{\}]_m : (\mathbf{Multi} \sigma \tau_d)$ 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). $\rho \vdash e_d \Downarrow \text{err}, \rho \vdash e \Downarrow \text{err}$

Trivially reduces to an error.

CASE (B-DefMethod).

- (1) $v = [v_d, t']_m,$
- (2) $\rho \vdash e_m \Downarrow [v_d, t]_m,$
- (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) $\text{IsAProps}(o_d, \tau_v) = \psi_{i+}|\psi_{i-},$

- (11) $\Gamma \vdash e_v \Rightarrow e_v : \tau_v$
- (12) $\Gamma, \tau_{1x}, \psi_{i_+} \vdash e'_f : \sigma ; \psi_{m_+} | \psi_{m_-} ; o_m$
- (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) $\text{tt} \vdash \psi_+,$
- (17) $\text{ff} \vdash \psi_-,$
- (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 $\vdash [v_d, t[v_v \mapsto v_f]]_m : (\mathbf{Multi} \ \tau_m \ \tau_d)$ 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). $\rho \vdash e_m \Downarrow \text{err}, \rho \vdash e \Downarrow \text{err}$

Trivially reduces to an error.

CASE (BE-DefMethod2). $\rho \vdash e_m \Downarrow [v_d, t]_m, \rho \vdash e_v \Downarrow \text{err}, \rho \vdash e \Downarrow \text{err}$

Trivially reduces to an error.

CASE (BE-DefMethod3). $\rho \vdash e_m \Downarrow [v_d, t]_m, \rho \vdash e_v \Downarrow v_v, \rho \vdash e_f \Downarrow \text{err}, \rho \vdash e \Downarrow \text{err}$

Trivially reduces to an error.

CASE (B-BetaClosure).

- $\rho \vdash e \Downarrow v,$
- $\rho \vdash e_1 \Downarrow [\rho_c, \lambda x^\sigma. e_b]_c,$
- $\rho \vdash e_2 \Downarrow v_2,$
- $\rho_c[x \mapsto v_2] \vdash e_b \Downarrow v$

SUBCASE (T-App).

- $e' = (e'_1 \ e'_2),$
- $\Gamma \vdash e'_1 : x : \sigma \xrightarrow[o_f]{\psi_{f_+} | \psi_{f_-}} \tau_f ; \psi_{1_+} | \psi_{1_-} ; o_1,$
- $\Gamma \vdash e'_2 : \sigma ; \psi_{2_+} | \psi_{2_-} ; o_2,$
- $e = (e_1 \ e_2),$
- $\vdash \tau_f[o_2/x] <: \tau,$
- $\psi_{f_+}[o_2/x] \vdash \psi_+,$
- $\psi_{f_-}[o_2/x] \vdash \psi_-,$
- $\vdash o_f[o_2/x] <: o$

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

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

From

- $\rho_c \models \Gamma_c,$
- $\Gamma \vdash e'_2 : \sigma ; \psi_{2_+} | \psi_{2_-} ; o_2$ 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 $\Gamma_c \vdash e'_b[v_2/x] : \tau_f[o_2/x] ; \psi_{f_+} | \psi_{f_-}[o_2/x] ; o_f[o_2/x]$, which can be justified by noting

- $\Gamma_c, \sigma_x \vdash e'_b \Rightarrow e_b : \tau_f$,
- $\Gamma \vdash e'_2 : \sigma ; \psi_{2_+} | \psi_{2_-} ; o_2$ 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

- $\Gamma_c \vdash e'_b[v_2/x] : \tau_f[o_2/x] ; \psi_{f_+} | \psi_{f_-}[o_2/x] ; o_f[o_2/x]$,
- $\rho_c \models \Gamma_c$,
- 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).

- $e' = (e'_1 e'_2)$,
- $\Gamma \vdash e'_1 : x : \sigma \xrightarrow[o_f]{\psi_{f_+} | \psi_{f_-}} \tau_f ; \psi_{1_+} | \psi_{1_-} ; o_1$,
- $\Gamma \vdash e'_2 : \sigma ; \psi_{2_+} | \psi_{2_-} ; o_2$,
- $e = (e_1 e_2)$,
- $\vdash \tau_f[o_2/x] <: \tau$,
- $\psi_{f_+}[o_2/x] \vdash \psi_+$,
- $\psi_{f_-}[o_2/x] \vdash \psi_-$,
- $\vdash o_f[o_2/x] <: o$

Prove by cases on c .

SUBCASE ($c = \mathbf{class}$). $\vdash x : \top \xrightarrow[\mathbf{class}(x)]{\mathbf{tt} | \mathbf{tt}} (\bigcup \mathbf{nil} \mathbf{Class}) <: x : \sigma \xrightarrow[o_f]{\psi_{f_+} | \psi_{f_-}} \tau_f$

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 $\rho(\mathbf{class}(o_2)) = (\mathbf{class} \rho(o_2))$, which evaluates to v .

Part 2 is trivial since both propositions can only be \mathbf{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 ($v_2 = \mathbf{true}$). $v = \mathbf{B}$

As above.

SUBCASE ($v_2 = \text{false}$). $v = \mathbf{B}$
As above.

SUBCASE ($v_2 = [\rho, \lambda x^\tau.e]_c$). $v = \mathbf{Fn}$
As above.

SUBCASE ($v_2 = [v_d, t]_m$). $v = \mathbf{Map}$
As above.

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

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

CASE (B-BetaMulti).

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

SUBCASE (T-App).

- $e' = (e'_1 e'_2)$,
- $\Gamma \vdash e'_1 : x:\sigma \xrightarrow[o_f]{\psi_{f+}|\psi_{f-}} \tau_f ; \psi_{1+}|\psi_{1-} ; o_1$,
- $\Gamma \vdash e'_2 : \sigma ; \psi_{2+}|\psi_{2-} ; o_2$,
- $e = (e_1 e_2)$,
- $\vdash \tau_f[o_2/x] <: \tau$,
- $\psi_{f+}[o_2/x] \vdash \psi_+$,
- $\psi_{f-}[o_2/x] \vdash \psi_-$,
- $\vdash o_f[o_2/x] <: o$,

By inversion on e_1 via T-Multi we know

- $\Gamma \vdash e'_1 : (\mathbf{Multi} \sigma_t \sigma_d) ; \psi_{1+}|\psi_{1-} ; o_1$,
- $\sigma_t = x:\sigma \xrightarrow[o_f]{\psi_{f+}|\psi_{f-}} \tau_f$,
- $\sigma_d = x:\sigma \xrightarrow[o_d]{\psi_{d+}|\psi_{d-}} \tau_d$,
- $\vdash v_d : \sigma_d$
- $\vdash v_k : \overline{\top}$, and
- $\vdash v_v : \sigma_t$.

By the inductive hypothesis on $\rho \vdash e_2 \Downarrow v_2$ we know $\Gamma \vdash v_2 : \sigma ; \psi_{2+}|\psi_{2-} ; o_2$.

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

– $\Gamma \vdash v_2 : \sigma ; \psi_{2+} | \psi_{2-} ; o_2$,

we can apply the inductive hypothesis to derive $\Gamma \vdash v_e : \tau_d[o_2/x] ; \psi_{d+} | \psi_{d-}[o_2/x] ; o_d[o_2/x]$.
Now we consider how we choose which method to dispatch to.

As $\text{GM}(t, v_e) = v_g$, by inversion on GM we know there exists exactly one v_k such that $v_k \mapsto v_g \in t$ and $\text{lsA}(v_e, v_k) = \text{true}$.

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

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

we can apply Lemma B.07 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:

- $\Gamma, \psi'_{+} \vdash e'_b[v_2/x] : \tau_f[o_2/x] ; \psi_{f+} | \psi_{f-}[o_2/x] ; o_f[o_2/x]$,
 - * From $\Gamma, \sigma_x \vdash e_b : \tau_f ; \psi_{f+} | \psi_{f-} ; o_f$ via the inductive hypothesis on $\rho \vdash (\lambda x^\sigma . e_b v_2) \Downarrow v$,
 - * then we can derive $\Gamma \vdash e'_b[v_2/x] : \tau_f[o_2/x] ; \psi_{f+} | \psi_{f-}[o_2/x] ; o_f[o_2/x]$ via substitution and the fact that x is fresh therefore $x \notin \text{fv}(\Gamma)$ so we do not need to substitute 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). $\rho \vdash e_1 \Downarrow v_1, \rho \vdash e_2 \Downarrow v_2, \text{IsA}(v_1, v_2) = v$

SUBCASE (T-IsA). $e' = (\text{isa? } e'_1 e'_2), \Gamma \vdash e'_1 : \tau_1 ; \psi_{1+} | \psi_{1-} ; o_1, \Gamma \vdash e'_2 : \tau_2 ; \psi_{2+} | \psi_{2-} ; o_2, e = (\text{isa? } e_1 e_2), \vdash \mathbf{B} <: \tau, \text{IsAProps}(o_1, \tau_2) = \psi'_+ | \psi'_-, \psi'_+ \vdash \psi_+, \psi'_- \vdash \psi_-, \vdash \emptyset <: o$

Part 1 holds trivially with $o = \emptyset$.

For part 2, by the induction hypothesis on e_1 and e_2 we know $\Gamma \vdash v_1 : \tau_1 ; \psi_{1+} | \psi_{1-} ; o_1$ and $\Gamma \vdash v_2 : \tau_2 ; \psi_{2+} | \psi_{2-} ; o_2$, so we can then apply Lemma B.07 to reach our goal.

Part 3 holds because by the definition of $\text{IsA } v$ can only be true or false, and since $\Gamma \vdash \text{true} : \tau$ and $\Gamma \vdash \text{false} : \tau$ we are done.

CASE (BE-IsA1). $\rho \vdash e_1 \Downarrow \text{err}$

Trivially reduces to an error.

CASE (BE-IsA2). $\rho \vdash e_1 \Downarrow v_1, \rho \vdash e_2 \Downarrow \text{err}$

Trivially reduces to an error.

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

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

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 $\rho(\mathbf{key}_k(o_m)) = (\text{get } \rho(o_m) k)$ and by the induction hypothesis on $e_m \rho(o_m) = \{\overrightarrow{(v_a v_b)}\}$, which together imply $\rho(o) = (\text{get } \{\overrightarrow{(v_a v_b)}\} k)$. 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 $\psi_+ = \text{tt}$ and $\psi_- = \text{tt}$.

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 $k \in \text{dom}(\{\overrightarrow{(v_a v_b)}\})$ and $\{\overrightarrow{(v_a v_b)}\}[k] = v$ imply $\vdash v : (\bigcup \overrightarrow{\tau_i})$.

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

Unreachable subcase because $k \in \text{dom}(\{\overrightarrow{(v_a v_b)}\})$, contradicts $k \in \mathcal{A}$.

SUBCASE (T-GetHMapPartialDefault). $e' = (\text{get } e'_m e'_k), \Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val } k), \Gamma \vdash e'_m : (\mathbf{HMap}^P \mathcal{M} \mathcal{A}) ; \psi_{m+} | \psi_{m-} ; o_m, k \notin \text{dom}(\mathcal{M}), k \notin \mathcal{A}, e = (\text{get } e_m e_k), \tau = \top, \psi_+ = \text{tt}, \psi_- = \text{tt}, \vdash \mathbf{key}_k(x)[o_m/x] <: o$

Parts 1 and 2 are the same as the B-Get subcase. Part 3 is trivial as $\tau = \top$.

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

SUBCASE (T-GetHMap). $e' = (\text{get } e'_m e'_k), \Gamma \vdash e'_m : (\bigcup \overrightarrow{(\mathbf{HMap}^\mathcal{E} \mathcal{M} \mathcal{A})}) ; \psi_{m_+} | \psi_{m_-} ; o_m,$
 $\Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val} k), \mathcal{M}[k] = \tau_i, e = (\text{get } e_m e_k), \vdash (\bigcup \overrightarrow{\tau_i}) <: \tau, \psi_+ = \text{tt}, \psi_- = \text{tt},$
 $\vdash \mathbf{key}_k(x)[o_m/x] <: o$

Unreachable subcase because $k \notin \text{dom}(\overrightarrow{\{(v_a v_b)\}})$ contradicts $\mathcal{M}[k] = \tau$.

SUBCASE (T-GetHMapAbsent). $e' = (\text{get } e'_m e'_k), \Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val} k),$
 $\Gamma \vdash e'_m : (\mathbf{HMap}^\mathcal{E} \mathcal{M} \mathcal{A}) ; \psi_{m_+} | \psi_{m_-} ; o_m, k \in \mathcal{A}, e = (\text{get } e_m e_k), \vdash \mathbf{nil} <: \tau, \psi_+ = \text{tt}, \psi_-$
 $= \text{tt}, \vdash \mathbf{key}_k(x)[o_m/x] <: o$

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 $\rho(\mathbf{key}_k(o_m)) = (\text{get } \rho(o_m) k)$ and by the induction hypothesis on $e_m \rho(o_m) = \overrightarrow{\{(v_a v_b)\}}$, which together imply $\rho(o) = (\text{get } \overrightarrow{\{(v_a v_b)\}} k)$. Since this is the same form as B-GetMissing, we can apply the premise $v = \mathbf{nil}$ to derive $\rho(o) = v$.

Part 2 holds trivially as $\psi_+ = \text{tt}$ and $\psi_- = \text{tt}$.

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 $k \notin \text{dom}(\overrightarrow{\{(v_a v_b)\}})$ and $v = \mathbf{nil}$ tell us v must be of type τ .

SUBCASE (T-GetHMapPartialDefault). $e' = (\text{get } e'_m e'_k), \Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val} k),$
 $\Gamma \vdash e'_m : (\mathbf{HMap}^P \mathcal{M} \mathcal{A}) ; \psi_{m_+} | \psi_{m_-} ; o_m, k \notin \text{dom}(\mathcal{M}), k \notin \mathcal{A}, e = (\text{get } e_m e_k), \tau = \top, \psi_+$
 $= \text{tt}, \psi_- = \text{tt}, \vdash \mathbf{key}_k(x)[o_m/x] <: o$

Parts 1 and 2 are the same as the B-GetMissing subcase of T-GetHMapAbsent. Part 3 is trivial, since $\tau = \top$.

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). $\Gamma \vdash e'_m \Rightarrow e_m : (\mathbf{HMap}^\mathcal{E} \mathcal{M} \mathcal{A}), \Gamma \vdash e'_k \Rightarrow e_k : (\mathbf{Val} k),$
 $\Gamma \vdash e'_v \Rightarrow e_v : \tau, k \notin \mathcal{A}, e' = (\mathbf{assoc } e'_m e'_k e'_v),$
 $e = (\mathbf{assoc } e_m e_k e_v), \vdash (\mathbf{HMap}^\mathcal{E} \mathcal{M}[k \mapsto \tau] \mathcal{A}) <: \tau, \psi_+ = \text{tt}, \psi_- = \text{ff}, o = \emptyset$
 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). $\rho \vdash e_1 \Downarrow \mathbf{false}$ or $\rho \vdash e_1 \Downarrow \mathbf{nil}, \rho \vdash e_3 \Downarrow v$

SUBCASE (T-If). $e' = (\text{if } e'_1 \ e'_2 \ e'_3)$, $\Gamma \vdash e'_1 : \tau_1$; $\psi_{1+} | \psi_{1-}$; o_1 , $\Gamma, \psi_{1+} \vdash e'_2 : \tau$; $\psi_{2+} | \psi_{2-}$; o , $\Gamma, \psi_{1-} \vdash e'_3 : \tau$; $\psi_{3+} | \psi_{3-}$; o , $e = (\text{if } e_1 \ e_2 \ e_3)$, $\psi_{2+} \vee \psi_{3+} \vdash \psi_+$, $\psi_{2-} \vee \psi_{3-} \vdash \psi_-$

For part 1, either $o = \emptyset$, or e evaluates to the result of e_3 .

To prove part 2, we consider two cases:

- if $\text{FalseVal}(v)$ 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 $\text{TrueVal}(v)$, 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). $\rho \vdash e_1 \Downarrow v_1$, $v_1 \neq \text{false}$, $v_1 \neq \text{nil}$, $\rho \vdash e_2 \Downarrow v$

SUBCASE (T-If). $e' = (\text{if } e'_1 \ e'_2 \ e'_3)$, $\Gamma \vdash e'_1 : \tau_1$; $\psi_{1+} | \psi_{1-}$; o_1 , $\Gamma, \psi_{1+} \vdash e'_2 : \tau$; $\psi_{2+} | \psi_{2-}$; o , $\Gamma, \psi_{1-} \vdash e'_3 : \tau$; $\psi_{3+} | \psi_{3-}$; o , $e = (\text{if } e_1 \ e_2 \ e_3)$, $\psi_{2+} \vee \psi_{3+} \vdash \psi_+$, $\psi_{2-} \vee \psi_{3-} \vdash \psi_-$

For part 1, either $o = \emptyset$, or e evaluates to the result of e_2 .

To prove part 2, we consider two cases:

- if $\text{FalseVal}(v)$ 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 $\text{TrueVal}(v)$, 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). $e' = (\text{let } [x \ e'_1] \ e'_2)$, $\Gamma \vdash e'_1 : \sigma$; $\psi_{1+} | \psi_{1-}$; o_1 , $\psi' = \overline{(\cup \ \text{nil} \ \text{false})}_x \supset \psi_{1+}$, $\psi'' = (\cup \ \text{nil} \ \text{false})_x \supset \psi_{1-}$, $\Gamma, \sigma_x, \psi', \psi'' \vdash e'_2 : \tau$; $\psi_+ | \psi_-$; 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

- $\Gamma, \sigma_x, \psi', \psi'' \vdash e'_2 : \tau$; $\psi_+ | \psi_-$; 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). $v = [\rho, \lambda x^\sigma . e_1]_c$

SUBCASE (T-Clos). $e' = [\rho, \lambda x^\sigma.e_1]_c$, $\exists \Gamma'. \rho \models \Gamma'$ and $\Gamma' \vdash \lambda x^\sigma.e_1 : \tau$; $\psi_{f_+} | \psi_{f_-}$; o_f , $e = [\rho, \lambda x^\sigma.e_1]_c$, $\psi_+ = \text{tt}$, $\psi_- = \text{ff}$, $o = \emptyset$

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 $\psi_+ = \text{tt}$, $\psi_- = \text{ff}$ 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). $v = [\rho, \lambda x^\sigma.e_1]_c$, $\rho \vdash \lambda x^\tau.e_1 \Downarrow [\rho, \lambda x^\sigma.e_1]_c$

SUBCASE (T-Abs). $e' = \lambda x^\sigma.e'_1$, $\Gamma, \sigma_x \vdash e'_1 : \tau$; $\psi_{1_+} | \psi_{1_-}$; $o_1, \vdash x : \sigma \xrightarrow{o_1} \tau_1 <: \tau$, $\text{tt} \vdash \psi_+$, $\text{ff} \vdash \psi_-$, $o = \emptyset$

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). $\rho \vdash e \Downarrow \text{err}$

SUBCASE (T-Error). $e' = \text{err}$, $e = \text{err}$, $\tau = \perp$, $\psi_+ = \text{ff}$, $\psi_- = \text{ff}$, $o = \emptyset$
Trivially reduces to an error.

□

THEOREM B.01 (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 B.08, since by lemma B.08 when $\vdash e' \Rightarrow e : \tau$; $\psi_+ | \psi_-$; o , either $\vdash e \Downarrow v$ or $\vdash e \Downarrow \text{err}$, therefore $\not\vdash e \Downarrow$ wrong. □

THEOREM B.02 (Type soundness for λ_{TC}). *If $\Gamma \vdash e' \Rightarrow e : \tau$; $\psi_+ | \psi_-$; o and $\rho \vdash e \Downarrow v$ then $\vdash v : \tau$; $\psi'_+ | \psi'_-$; o' for some ψ'_+ , ψ'_- and o' .*

PROOF. Corollary of lemma B.08. □

CV: Ambrose Bonnaire-Sergeant *Last update on May 17, 2019*

abonnairesergeant[at]gmail.com • ambrosebs.com • github.com/frenchy64

Education

Indiana University Bloomington	BLOOMINGTON, INDIANA
Ph.D. in Computer Science	2014 — 2019
Master of Science in Computer Science	2014 — 2017
University of Western Australia	WESTERN AUSTRALIA, AUSTRALIA
BSc in Computer Science with Honours	2008 — 2013

Employment

Software Developer (Remote Intern)	SPARKFUND, <i>May 2017 — February 2018</i>
Studied verification techniques used in practice (Clojure).	
<ul style="list-style-type: none">• enhanced interactive pricing estimation tools using ClojureScript and om.next• produced internal pricing reports in collaboration with sales team using Clojure and Google Sheets• parallelized the CI builds of several projects with a migration to CircleCI 2.0 Workflows• completed company-wide upgrade to a version of Clojure with breaking changes	
Research Assistant / Assistant Instructor	INDIANA UNIVERSITY BLOOMINGTON, <i>Fall 2014—Spring 2019</i>
<ul style="list-style-type: none">• published peer reviewed papers about Typed Clojure and clojure.spec• taught undergraduates introductory programming based on How To Design Programs (C211) and introductory data structures (C343)	
Analyst Programmer	UNIVERSITY OF WESTERN AUSTRALIA, <i>2010 — 2011</i>
<ul style="list-style-type: none">• conducted interviews of library staff to identify the format and location of research data• set up VIVO semantic web software, utilizing Java, XSLT, bash, and Linux	

Open Source Work

Crowdfunded Open Source work	PERTH, WESTERN AUSTRALIA
Automatic Annotations for Typed Clojure and clojure.spec (\$8,621 USD raised by 69 backers)	2016
This campaign concentrated on automating the manual labor of type and spec annotations. It helped me attend multiple industry conferences to meet real users of Typed Clojure and clojure.spec to discuss the needs of the community.	
Gradual Typing for Clojure (\$11,695 USD raised by 199 backers)	2015
These funds helped me design a <i>gradual typing</i> framework for Typed Clojure. It supports automatic contracts for global variables based on the typed-untyped boundary, as well as allowing arbitrary exporting of macros from typed namespaces—a shortcoming of previous systems.	

Typed Clojure (\$35,254 USD raised by 545 backers)

2013

These funds helped support me as I worked on improving and extending Typed Clojure's type system. \$5,000 of these funds were used to commission further open source work on tools.analyzer, now an important Clojure library.

Google Summer of Code

CLOJURE ORGANISATION

Student

2012, 2013

I developed Typed Clojure, improving the support of Clojure idioms, supporting more expressive types, and adding documentation.

Mentor

2014 (3 students), 2015 (2 students)

I have mentored 5 projects, including several adding advanced type system features to Typed Clojure, and work towards a ClojureScript compiler using tools.analyzer as a back-end.

Administrator

2014, 2015

My role as an administrator included proposing projects, preparing and reviewing application documents, and advertising for interested students.

Selected Open Source Contributions

core.typed

LEAD DEVELOPER

Designed, implemented, and maintain the main library of Typed Clojure.

core.match

CONTRIBUTOR

Studied optimising pattern matching literature and collaborated on a pattern matcher for Clojure.

Clojure

CONTRIBUTOR

Contributed several minor patches improving error messages, fixing bugs, and adding features to the core Clojure language.

ClojureScript

CONTRIBUTOR

Converts the internal AST representation of ClojureScript's compiler to tools.analyzer format. Various bug fixes and performance enhancements.

Research Groups

PL Wonks

INDIANA UNIVERSITY BLOOMINGTON

Gradual Typing Group

INDIANA UNIVERSITY BLOOMINGTON

Conference Publications

Practical Optional Types for Clojure (ESOP'16) WITH ROWAN DAVIES, SAM TOBIN-HOCHSTADT

Theses

A Practical Optional Type System for Clojure
Undergraduate Honours dissertation.

SUPERVISED BY ROWAN DAVIES

Selected Talks

Tool-assisted spec Development	CLOJURE/CONJ 2017
Practical Optional Types for Clojure	ESOP 2016
Typed Clojure: From Optional to Gradual Typing	STRANGELOOP 2015
Typed Clojure in Practice	STRANGELOOP 2014
Typed Clojure	CLOJURE/CONJ 2012
Introduction to Logic Programming	CLOJURE/CONJ 2011

Service

- Organizer of PL Wonks, Indiana University's weekly Programming Languages seminar (2017-2019)
- President of Indiana University Graduate Computer Science Association (2017-2019)