Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Ochre: A Dependently Typed Systems Programming Language

Author:
Charlie Lidbury

Supervisor(s):
Steffen van Bakel
Nicolas Wu

June 15, 2024

Submitted in partial fulfillment of the requirements for the MEng Computing of Imperial College London

Abstract

This research presents Ochre, a dependently typed, low-level systems language. In Ochre, programmers can use the type system to prove stronger properties about their programs than they can in non-dependently typed languages such as Rust or Haskell. Ochre also gives programmers low-level enough control over their programs to be able to express efficient in-place algorithms and control the memory layout of user-defined data structures, which makes it a systems language, akin to Rust, C, or C++.

This paper presents the formal semantics of Ochre via λ_{Ochre}, an abstract interpretation over λ_{Ochre}, a concrete interpretation, a proof that the abstract interpretation and the concrete interpretation are consistent, and an implementation of Ochre in the form of an embedding into the Rust programming language.

Acknowledgments

I would like to thank my supervisor Steffen van Bakel for his type system wisdom, relentless skepticism, and for giving me the freedom to explore such a high-risk project with very little bearing on his research. Steffen even involved his son Isaac van Bakel to help us understand RustBelt and Aeneas, prior work which Ochre takes heavy inspiration from.

I would also like to extend as much gratitude as is physically possible to do via Latex to David Davies, a previous master's student of Steffen who has proven invaluable throughout this project. David has taught me crucial things about dependent types, spent days getting into the nitty gritty of my ideas to make sure I'm on track, and, most importantly, given me the confidence in myself I needed to commit to this project.

Last, but in no means least, I would like to thank my mother Kate Darracott. As well as giving birth to me, which has arguably enabled this project even more than the aforementioned, Mum came up with the brilliant name "Ochre", after being told no more than "the syntax is going to look a little bit like Rust's". Despite not knowing what syntax is, or the significance of dependently typed low-level systems programming languages, she may well have had the most visible contribution to this project of anyone.

Ethical Considerations

Much like Wittgenstein [Wittgenstein, 1922, proposition 6.421], I believe there is an equivalence between ethics and aesthetics; if you do not, here are a few parallels between the two you might find thought-provoking: We do not choose what we deem ethically permissive, much like we do not choose what we find beautiful. Pursuing one's ethical convictions is not a means to an end, it is an end in and of itself, much like aesthetic experiences.

I and many others including cite cite cite, find aesthetic value in problems & concepts turning out to be reduceable to each other and equivalences being drawn between distant domains. Some particularly high-profile instances of this happening include Euler's formula, the Curry-Howard correspondence and the Church-Turing thesis. To a smaller degree, I also think it happened with Rust's borrow checker, in solving memory management they also solved concurrency, iterator invalidation, and a few other problems that plagued imperative languages.

Despite being sufficiently arrogant and pretentious [Lidbury, 2024, Ethical Considerations], I know Ochre isn't as singificant or as beautiful as the previously mentioned identities and isomorphisms. But, in the walled garden of my special interests and obsessions, I have found great aesthetic value in the interplay between ownership semantics and dependent types.

From this aesthetic value, and its equivalence to moral value, I conclude that this research is ethically permissible; I hope Imperial's ethical approval process will too.

Contents

Chapter 1

Introduction

(TAKEN FROM AENEAS FOR NOW)

In 2006, exasperated by yet another crash of his building's elevator's firmware, and exhausted after walking up 21 flights of stairs, Graydon Hoare set out to design a new programming language [Hoare, 2022]. The language, soon to be known as Rust, had two goals. First, to be system-oriented, meaning the programmer would deal with references, pointers, and manually manage memory. Second, to be safe, meaning the compiler's static discipline would rule out memory errors such as use-after-free, or arbitrary memory access. Even though the language evolved a great deal since its inception, these two core premises remain today.

Eighteen years later, Rust enjoys a substantial amount of success and has ranked as the most loved programming language for 7 consecutive years on StackOverflow's developer survey [sta, 2021], until they changed the phrasing of the question in 2023 in which it was the most *admired* language. But as the systems community can attest [**?**Lorch et al., 2020; Ferraiuolo et al., 2017; Bhargavan et al., 2017], memory safety is too weak of a property, no matter how remarkable of an achievement Rust is.

We have attempted to prove further properties

1.1 The Problem

This research hopes to develop a type-checker that is capable of type-checking languages that support both mutation and a kind of type called dependent types. It will do this by removing mutation from the code before type checking, so the type checker only has to reason about immutable code.

Dependent types are covered properly in the background section, but for now, it's enough to know they're a feature that allows you to check even more properties than just type safety at compile time. For instance, instead of just being able to say a variable x is an integer, you can say it's an *even* integer, and reject programs like $x := 5$ at compile time, instead of waiting

for them to go wrong at runtime.

This type-checker will support mutation, which is when a variable's value is changed. For instance, when a variable is declared with a value like $x = 2$, then later given a new value like $x := 5$. The most popular languages all support mutation [cite], it's somewhat the (industry) default. Some languages choose to be *immutable* however, which means they do not support mutation. These include Haskell, and almost all languages with dependent types like Agda, Idris, and Coq.

This type-checker is being built to hopefully be used for a larger, more useful language in the future, called Ochre. Ochre which will have both the speed of *systems languages* like C and Rust and the ability to reason about runtime behaviour at compile time of *theorem provers* like Agda and Coq. Exactly what systems languages and theorem provers are is discussed in Chapter 2.4.

For now, I plan on presenting this type-checker in the form of an implementation; however, there is a good argument for focusing more on the theory behind this type-checker, for instance by presenting a set of typing rules or an abstract algorithm. Whether an implementation-heavy or theory-heavy approach is better is an open, and very important question.

1.1.1 Why Is It Hard?

The problem with having these features together in the same language is that a value that another variable's type depends on can be mutated, which changes the *type* of the other variable. Concretely: if we have a variable $x : T$, and another variable $y : F(x)$ whose type depends on x, we can assign a new value to x which in turn changes the type $F(x)$; now y is ill-typed because its type has changed, but not it's value. The programmer could fix this by reassigning y with a new value of type $F(x)$, if this happens before y is ever used, the compiler should be able to identify this interaction as type-safe.

1.2 The Solution

The technique this research presents goes as follows: convert the source code from the programmer, which will contain mutation, into a functionally equivalent (but maybe inefficient) immutable version, which can be dependently type-checked. Once this immutable version has been type-checked, the original mutable version can be executed, with full efficiency granted to it by mutability.

Because this translation has been shown to be behaviour preserving[**?**] we know properties we prove about the immutable version of the programmer's code also hold for the mutable version which will be executed.

1.3 Motivation

The main contribution of this research will be progress towards making a language that supports both mutability and dependent types, so the motivation behind this research will be the motivation behind these two features, as well as their combination.

This section refers to technical concepts that haven't been explained yet, such as dependent types. The reader is advised to refer to Chapter 2 if they find concepts being referenced that they do not understand.

1.3.1 Mutation

This section argues why one would want mutation in a programming language.

Performance

Some data structures and operations, such as hash maps and their $O(1)$ access/modification, need to modify data in place to be efficiently implemented. Immutable languages like Haskell get around this by performing these mutable operations via unsafe escape hatches and then wrapping those in monads to sequence the immutable operations. However, this often makes mutable code harder to maintain and harder for beginners to understand. For instance, to operate on two hash maps at the same time, you would have to be operating within multiple monads simultaneously, which involves monad transformers or effect types, a much more advanced skillset than what would be required to do the same in Python.

This has widespread effects on the data structures programmers use, and how they structure their programs. Often programmers in immutable languages will simply switch to data structures that don't perform as well but are easier to use in a pure-functional context, like tree-based maps and cons lists instead of hash-maps and vectors.

The performance of explicit mutation can also be easier to reason about. For instance, the Rust code which increments every value in a list of integers doesn't perform any allocations: `for x in xs.iter_mut() { x += 1 }`; whereas the Haskell equivalent looks like it allocates a whole new list, and relies on compiler optimizations to be efficient: `map (+1) xs`. In fact, in this example, Haskell does not do the update in-place and instead allocates a new list in case the old one is being referred to somewhere else. Languages like Koka

Usability

Some algorithms are best thought of in terms of mutable operations, and new programmers especially tend to write stuff mutably. By embracing this in the language design, we can come to the user instead of making the user come to us.

Since the CPU is natively works on mutable operations, if you want control over what the CPU does, which you do if you want to extract all the performance you can from it, you want the language to have graceful support for mutation.

The Immutability Argument

Proponents of immutability argue immutability helps you reason about your program; since there are no side effects of function calls, you cannot be tripped up by side effects you didn't see coming.

I think this correctly identifies that aliased mutation is bad, but goes too far by removing all mutation. In languages like Rust, only one *mutable* reference can exist to any given memory location, which is needed to write to that memory. This gives you most of the benefits of mutation while avoiding the uncontrolled side effects.

Popularity

The majority is often wrong, but it's a good sign if significant proportions of the industry agree on something. In the last quarter of 2023, at least 97.24% of all committed code was written in a language with mutation [cite: GitHut]. At the very least this shows that people like languages with mutability, even if they are wrong to do so.

1.3.2 Dependent Types

This section argues why one would want dependent types in a programming language.

Formal Verification

Dependent types are one of the ways to mechanize logical reasoning, which allows you to reason about the correctness of your programs. For instance, a program that sorts lists should have (amongst other things) the property that it always outputs a list with ascending items. In a language with dependent types, you can make the type of a function express the fact that not only will it return a list of integers, but that it will be a sorted list of integers.

The goal of Ochre, the language this research is done in the name of, is to enable formal verification of low-level systems code. There are other ways to do formal verification, but this is a popular and natural one.

Usability

Dependent types are a notoriously difficult feature to learn and reason about, and their ergonomics are underexplored due to them only being used in very niche, academic languages. However, I think if you're not using them for their extra power, they can be just as ergonomic as typical type systems. In this sense, if the language is designed correctly, you only pay for what you use.

1.3.3 Mutation + Dependent Types

This section explains why mutability and dependent types combine to form more than the sum of their parts.

If you use the mutability to make the language high performance, you can use mutability and dependent types to do formal verification of high performance code. This is a common combination of requirements because they both occur when software is extremely widespread and has very high budget.

1.3.4 This Particular Method

This section explains what advantages this particular method has over other combinations of mutability and dependent types, such as ATS, Magmide, and Low*.

This type checker allows the types and mutable values to be unusually close. In ATS for instance there are basically two separate languages: a dependently typed compile time language and a mutable run-time language. This creates lots of overhead manually linking the two together. For instance, $x : int(y)$ means an integer x with value y. In compile time contexts, you use y to refer to the value, in runtime contexts you use x. I hope to remove the need for this distinction.

Chapter 2

Background

2.1 Dependent Types

2.2 Rust

Rust is a modern programming language that offers a unique combination of strong (memory) safety guarantees and bare-metal performance. Rust innovates in other areas relevant to software engineering, but for this research performance and safety are the two key features which will be built upon.

Performance

Rust is a fast language. Its performance is roughly equivalent to that of C and C++ Rus [b], which are generally accepted as the benchmark of language performance. Rust has enduring performance problems Are [2022], but it is fair to say that on the whole there aren't major performance differences between the fastest languages. The fastest programming languages have more or less hit a ceiling of performance, with no major improvements in speed even since Fortran Gcc which dates back to 1957 [Wilson and Clark, 2001, p. 16].

Making a fast programming language is more about removing slow features than it is about introducing ones that explicitly help performance. Languages like Haskell and Java automatically handle memory allocation and deallocation at the cost of having to have a garbage collector that periodically scans the heap and deallocates inaccessible objects; this is an example of a feature that reduces performance.

Rust is a fast language because it doesn't have a runtime or garbage collector, and has an efficient memory layout. In languages like Haskell or Java, almost all data is heap-allocated and deallocated automatically via a

To generate optimal code, systems languages let the programmer manage their memory, and choose memory layouts. In doing so, they typically sacrifice the memory safety guarantees higher-level languages make due to not being able to check the programmer has managed their memory correctly, this is the case in C and C++. Rust uses a concept called *ownership* to recover these memory safety guarantees while still giving the programmer sufficient control to match C and C++'s performance.

Ownership

Ownership is a set of rules that govern how a Rust program manages memory. All programs have to manage the way they use a computer's memory while running. Some languages have garbage collection that regularly looks for no-longer-used memory as the program runs; in other languages, the programmer must explicitly allocate and free the memory. Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks. If any of the rules are violated, the program won't compile. None of the features of ownership will slow down your program while it's running. [1]

There are three rules associated with ownership in Rust:

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Borrowing And The Borrow Checker

A consequence of only being able to have one owner of any given value at a time is that passing a value to a function invalidates the variable that used to hold that value. This is referred to as the ownership *moving*. For instance:

```
let x = Box::new(5);
f(x); // Ownership of x passed to f
g(x); // Invalid, we no longer have ownership of x
```

To get around this we could get the functions to give ownership back to us when they return, but this is very syntax-heavy. Rust uses a concept called borrowing in this scenario, which allows you to temporarily give a function access to a value, without giving it ownership. The above example would be done like so:

```
let x = Box::new(5);
f(&x);
g(&x); // Now works
```

[1] Paragraph taken from the Rust Book https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html which I highly recommend for a deeper explanation of ownership.

Here, `&x` denotes a *reference* to `x`. At runtime, this is represented as a pointer. There are two different types of references in Rust: immutable references, denoted by `&T`, and mutable references denoted by `&mut T`. For any given value, you can either hold a single mutable reference or n immutable references, but never both at the same time. This is called the aliasing xor (exclusive or) constraint, or AXM for short.

The borrow checker keeps track of when these references exist to ensure AXM is being upheld. To do this the programmer must annotate references with lifetime annotations, so the compiler has the information of how long the programmer intends each reference to last. Checking these lifetimes overlap in compatible ways is the job of the borrow checker.

2.3 Aeneas & The LLBC

2.4 Prerequisite Concepts

This section explains the concepts required to understand this research.

2.4.1 Mutability

Mutability is when the value of a variable can change at runtime. For instance in Rust, `let mut x = 5; x = 6;` first assigns the value 5 to the variable x, then updates it to 6, which means the value of x depends on the point within the programs execution. This becomes more relevant when you have large objects that get passed around your program, like `let mut v = Vec::new(); v.push(1); v.push(2);` which makes a resizable array on the heap, then pushes 1 and 2 to it.

In Rust to make a variable mutable you must annotate its definition with `mut`, but in most languages, it is just always enabled, like in C `int x = 5; x = 6;` works.

2.4.2 Dependent Types

A dependent type is a type that can change based on the value of another variable in the program. For instance, you might have a variable y which is sometimes an integer, and sometimes a boolean, depending on the value of another variable, x.

When discussing dependent types, there are two important dependent type constructors: Σ and Π. They're usually referenced together because they're roughly equivalent; the dual of Σ types are Π types and visa versa, which apparently means something to category theorists. In the following, I use $Vec(\mathbb{Z}, n)$ to denote the type of an n-tuple of integers, i.e. $(1, 2, 3) : Vec(\mathbb{Z}, 3)$.

- **Dependent Functions** (Π Types) - A dependent function is one whose return type depends on the input value. For instance, you could define a function f which takes a natural n, and returns n copies of 42 in a tuple i.e. $f(3) = (42, 42, 42)$. f's type would be denoted as $f : (\mathbf{n} : \mathbb{N}) \to Vec(\mathbb{Z}, \mathbf{n})$ in Agda/Ochre syntax, or $f : \Pi_{\mathbf{n}:\mathbb{N}} Vec(\mathbb{Z}, \mathbf{n})$ in a more formal mathematical context.
- **Dependent Pairs** (Σ Types) - A dependent pair is a pair where the type of the right element depends on the value of the left element. For instance, you could define a pair p which holds a natural n and a n-tuple of integers i.e. $p = (3, (42, 42, 42))$. p's type would be denoted as $p : (\mathbf{n} : \mathbb{N}, Vec(\mathbb{Z}, \mathbf{n}))$ in Agda/Ochre syntax, or $p : \Sigma_{\mathbf{n}:\mathbb{N}} Vec(\mathbb{Z}, \mathbf{n})$ in a more formal mathematical context.

A language supports dependent types if it can type-check objects like the aforementioned f and s. Just allowing them to exist is not enough. For instance, Python is not dependently typed just because a function's return type can depend on its input, because its type checker doesn't reject programs when you do this wrong. f can be typed in Agda, a dependently typed language with $f : (n : \mathbb{N}) \to Vec(\mathbb{Z}, n)$ but has no valid type in Haskell, which doesn't support dependent types.

2.4.3 Formal Verification with Dependent Types

While dependent types can be nice to have by themselves, a large part of their motivation is using them to perform formal verification.

If you are willing to accept that dependent types can be used to perform formal verification, you do not need to understand how dependent types can be used for logical reasoning: none of this information will be used since the goal of this research is not to perform formal verification, it's just to do dependent type checking.

Readers who are nonetheless interested are invited to read Appendix A.1.

2.4.4 Rust

The mutable $\to$ immutable translation this research relies on requires lifetime annotations to work. While ownership and lifetimes are standalone concepts, their only real-world use case so far has been memory management in the Rust programming language. This section explains these concepts in the context of Rust.

2.4.5 Mutable $\to$ Immutable Translation

To reason about and type-check the mutable code from the programmer, the type checker this research presents translates the source code into an immutable version, as outlined in

Section 1.2.

The crux of this translation is the observation that **a function that mutates a value can be replaced by one that instead returns the new value**. I.e. if the programmer writes a function with type `&mut i32 -> ()`, it can be replaced by `i32 -> i32`. Which would then be used like this:

Listing 2.1: Original

```
let mut x = 5;
f(&mut x); // Mutates x
```

Listing 2.2: Translated

```
let x = 5;
let x = f(x); // Re-defines x
```

The complexity of this translation comes in handling all language constructs in the general case, for instance, if statements need to return the values they edit. Like so:

Listing 2.3: Original

```
let mut x = 5;
if x > 3 {
  x = x + 1; // Mutation
}
```

Listing 2.4: Translated

```
let x = 5;
let x = if x > 3 {
  x + 1
} else {
  x
};
```

This quickly gets complicated when you start to use more advanced features like for loops and functions which return mutable references [2]. So much so safe Rust isn't even entirely covered by the two main attempts at this translation Electrolysis [**?**] and Aeneas [Ho and Protzenko, 2022] [3]. In this research I don't intend to support any constructs not already supported by either of these prior works, so I can use the translation algorithms they have already developed.

2.5 Related Work

Related work comes under two main categories: research which works towards combining mutability with dependent types, and more general work which works towards formal verification of low level code.

[2] See [Ho and Protzenko, 2022] Chapter 2 *Aeneas and its Functional Translation, by Example* for explanation of returning mutable references. (Search for "Returning a Mutable Borrow, and a Backward Function") for the exact paragraph.

[3] See Figure 14 of [Ho and Protzenko, 2022] for a table showing roughly which features are covered by Aeneas/Electrolysis, and see https://kha.github.io/electrolysis/ for exact Rust coverage for Electrolysis.

2.5.1 Languages with Mutability and Dependent Types

ATS

ATS [**?**] is the most mature systems programming language to date, with work dating back to 2002 [ATS, b]. As its website states, it is a *statically typed programming language that unifies implementation with formal specification* [ATS, a].

It's more or less an eagerly evaluated functional language like OCaml, but with functions in the standard library that manipulate pointers, like `ptr_get0` and `ptr_set0` which read and write from the heap respectively. To read or write to a location in memory, you must have a token that represents your ownership of the memory, called a *view*.

For instance, the `ptr_get0` function has the type $\{l : addr\}(T@l|ptr(l)) \rightarrow (T@l|T)$ where

- $\{l : addr\}$ means for all memory addresses, l
- $|$ is the pair type constructor
- $T@l$ means ownership of a value of type T, at location l. Since it is both an input and an output, this function is only *borrowing* ownership.
- $ptr(l)$ means a pointer pointing to location l. Since it can only point at location l, it is a singleton type. This is used to convert the static compile-time variable l into an assertion about the runtime argument.

So overall, this type reads "for all memory addresses l, the function borrows ownership of location l, and turns a pointer to location l into a value of type T".

This necessity to manually pass ownership around introduces a lot of administrative overhead to ATS, which is one of the reasons it is a notoriously hard language to learn/use. ATS introduces syntactic shorthand for these things which you can use in simple cases to clean things up, but still requires this proof passing in many cases which would be dealt with automatically by Rust's borrow checker.

Over the years several versions of ATS have been built, with interesting differences in approach. The current version, ATS2 has only a dependent type-checker, whereas the in-progress ATS3 uses both a conventional ML-like type-checker, as well as a dependent type-checker, and approach that the author of ATS himself developed in separate research, from which ATS3 gets its full name, ATS/Xanadu.

Magmide

The goal of Magmide [**?**] is to "create a programming language capable of making formal verification and provably correct software practical and mainstream". Currently, Magmide is

unimplemented, and there are barely even code snippets of it. However, there is extensive design documentation in which the author Blaine Hansen lays out the compiler architecture he intends to use, which involves two internal representations: *logical* Magmide and *host* Magmide.

- Logical Magmide is a dependently typed lambda calculus of constructions, where to-be-erased types and proofs are constructed.
- Host Magmide is the imperative language that runs on real machines. (Hansen intends on using Rust for this)

I believe this will mean there are two separate languages co-existing on the front end, much like the separation between type-level objects and value-level objects in a language like Haskell.

I suspect this will cause a similar situation to what you see in ATS where for each variable you care about you have two versions, a compile-time one and a runtime one, but it's hard to tell because of the lack of code examples.

Low*

Low*[**?**] is a subset of another language, F*, which can be extracted into C via a transpiler called KreMLin. It has achieved impressive results, mostly at Microsoft Research, where they have used it to implement a formally verified library of modern cryptographic algorithms[**?**] and EverParse

Its set of allowed features is carefully chosen to make this translation possible in the general case, which restricts the ergonomics of the language, it does not support closures, and therefore higher-order programming for example.

It is very much not a pay-for-what-you-use language, to compile anything you must manually manage things like pushing and popping frames on and off the stack, so even if it can achieve impressive results, it's only useful for teams willing to pay the high price which comes with verifying the entire program. This research aims to be better by not requiring any effort from the programmer in the case that they do not wish to use dependent types for their reasoning power.

2.5.2 Embedding Mutability in Languages With Dependent Types

Ynot: Dependent Types for Imperative Programs

Ynot[**?**] is an extension of the Coq proof assistant which allows writing, reasoning about, and extracting higher-order, dependently-typed programs with side-effects including muta-

tion. It does so by defining a monad `ST p A q` which performs an effectful operation, with precondition `p`, postcondition `q` and producing a value of type `A`. They also define another monad, `STSep p A q` which is the same as `ST` except it satisfies the frame rule from separation logic: any part of the heap that isn't referenced by the precondition won't be affected by the computation. This means if you prove properties about a `STSep` computation locally, those proofs still apply even when the computation is put into a different context: this is called compositional reasoning. The Ynot paper presents a formally verified mutable hash table.

Ynot is important foundational work in this area which seems to have inspired many of the other related work here, but is itself not up to the task of verifying low-level code for two reasons:

1. It cannot be used to create performant imperative programs because all mutation occurs through a Coq monad which limits the performance to what you can do in Coq, which is a relatively slow language. This is in contrast to Low*[**?**] for example which is extracted to C, and therefore unrestricted when it comes to performance.

2. To do any verification at all, you must use heap assertions, instead of reasoning about the values directly. This is sometimes needed, like when you're doing aliased mutation (verifying unsafe Rust), but usually not; Aeneas[Ho and Protzenko, 2022] claims to be hugely more productive than its competitors by not requiring heap assertions for safe Rust code.

2.5.3 Formal Verification of Low-Level Code

Low-level code, such as C code can be directly reasoned about by theorem provers like Isabelle, as was done to verify an entire operating system kernel SeL4[**?**]. However, going via C like this has major drawbacks: since the source language is very unsafe, you have a lot of proof obligations. For instance, when reasoning about C you must often prove that a set of pointers do not point to the same location, otherwise mutating the value of one might mutate the others. With Rust references you do not need to do this because the type system prevents you from creating aliased pointers.

Rust Belt

RustBelt[**?**] is a formal model of Rust, including unsafe Rust. Its primary implementation is a Coq framework, Iris[**?**] which allows you to model unsafe Rust code in Coq, and prove it upholds Rust's correctness properties.

I see RustBelt as a great complement to this work in the future: real programs require unsafe code, but you want to avoid having to model your code in a separate proof assistant as little as possible. In Ochre, I imagine the few people who write unsafe code will verify

it with something like RustBelt, while the majority won't have to, but will benefit from the guarantees provided by the verified libraries they use which do.

Chapter 3

Ochre, by Example

This chapter introduces the various language constructs of Ochre, at first via intuitive examples, then formally. Each language construct's runtime behavior is discussed, then how it is reasoned about statically, which splits this chapter into 4 sections with the following distinctions:

DEPRECATED

	Runtime Semantics	Static Analysis
Intuition Building	Section 3.1 Ochre, by Example	Section 3.2 Type & Borrow Checking, by Example
Formal	Section **??** Concrete Interpretation	Section 4 Abstract Interpretation

The motivations behind Ochre, alternative design decisions, evaluation, implementation, or reasoning about any properties are all explicit non-goals of this Chapter.

Attribution

With the exception of references, the primitive types in Ochre are from the $\Pi\Sigma$ language presented in Altenkirch et al. [2010], including the representation of algebraic data types.

The mutation & memory management techniques presented are from Rust, including move semantics, references, and the restrictions placed on references.

The novel work presented is the combination of these two features, which requires introducing a new kind of subtyping in which every term is its own type. TypeScript partly does this with literal types, producing results like `5 : 5`, but this research takes this to its logical conclusion where even functions are their own type, and there is almost no distinction between

types and terms apart from the requirement that all types are resolved at compile time.

explain double page + hyper links thing

do hyperlinks thing

3.1 The Language

This section covers Ochre in a gradual, example-heavy manner, much like programming language tutorials like The Rust Book [Rus, a]. The goal of this section is to build an intuition behind the behavior which the type-checking will later reason about.

Ochre is an impure functional language, composed of expressions that can have side effects.

Basic Language Constructs

The simplest Ochre value is an *atom*. Atoms are constructed with `'`, for example: `'hello` or `'world`[1]. Atoms are an unopinionated primitive type upon which more complex structures can be built.

```
'hello
```

M=N writes the result of evaluating N to M, for example: `x='one` sets `x` to 'one. Declarations are implicit in Ochre (for now); if `x` was in scope previously, `x='one` will bring it into scope, and if it was already in scope, it will mutate it. M;N sequences M, then N. Line comments are opened with `//`.

```
x = 'hello;
x // 'hello
```

References & Mutation

Variables are either modified directly or via a mutable reference. The latter is constructed with `&mut` and eliminated (dereferenced) with `*`.

```
x = 'one;
x = 'two; // mutates x directly
rx = &mut x;
*rx = 'three; // mutates x via a mutable reference
x // 'three
```

Listing 1: Mutation

[1]The runtime representation of an atom is assumed to be the hash of the string after the tick, which makes them constant length. This allows them to be stack-allocated instead of heap-allocated

Whilst a mutable reference to a value exists, that value cannot be read or modified directly, it can only be read or modified via the mutable reference. In Listing 1, the use of `x` on line 5 is not an error despite `rx` existing because is implicitly *dropped* just before the usage of `x`. Because of this implicit drop, `rx` cannot be used after line 5.

In practice, this is intolerably restrictive because it means only one pointer can exist to any value at a time. Like Rust, Ochre solves this by supporting *immutable* references, constructed with `&` and dereferenced with `*`. These allow the programmer to have multiple references to the same value, called *aliasing*. There is a tradeoff that you cannot mutate the referenced value, known as *aliasing xor mutability* (AXM), and it's crucial to how Rust can be converted to pure functional code, or dependently type-checked [Ho and Protzenko, 2022; Ullrich, 2024].

```
1    x = 'one;
2    rx1 = &x;
3    rx2 = &x;
4    x; // 'one
5    *rx1; // 'one
6    *rx2; // 'one
```

Listing 2: The value `'one` can be accessed via `x`, `rx1`, and `rx2` simultaneously

Pairs

M , N constructs the pair of M and N. Pairs are typically surrounded in brackets to make the precedence explicit. M `.0` and M `.1` access the right and left elements of the pair M.

```
1    x = ('one, 'two);
2    x.0; // 'one
3    x.1; // 'two
```

Move Semantics

Ochre uses Rust's ownership semantics to handle manual memory management. Using a value *moves* it, which means it is no longer accessible in the original location. This means you have exclusive access to any value not accessed via an immutable reference. This enables the "whenever a variable goes out of scope, free its associated memory" rule, which is how Rust and Ochre avoid the need for a garbage collector.

Move semantics can lead to some strange results, such as the following program being invalid:

```
1  x = 'one;
```

```
y = x;
x; // error! use of moved value
```

`y = x` moved the value 'one from `x` into `y`, which uninitializes `x`. Moving is granular; you can move components of a pair out of the pair without invalidating the whole pair:

```
x = ('unmoved, 'moved);
y = x.1; // move right component into y
x.0; // 'unmoved
x.1; // error! use of moved value
```

Structural Typing and Type Union

Ochre uses a structural type system. This means a type is entirely defined by the (potentially infinite) set of its inhabitants. This is in contrast to *nominal* typing, where type equivalence depends on the type's name or place of declaration. Take the following type definitions in Rust:

```
struct Foo(i32, i32);
struct Bar(i32, i32);
```

Both `Foo` and `Bar` are types that can be constructed with a pair of integers[2]. In Rust, it would be a type error to pass a `Foo` to a function that expects a `Bar`, because despite holding the same data, they are different types. The equivalent Ochre code would be:

```
Foo = (Int, Int);
Bar = (Int, Int);
```

Unlike in nominally typed languages, an Ochre function which expects a value of type `Foo` as input, can be given a value of type `Bar`. Every identifier you use to refer to a type in Ochre is roughly equivalent to a type *alias* in nominally typed languages like Rust and Haskell.

In Ochre, every value is its own type. So `'one` is of type `'one`, which is expressed in Ochre via colon. Non-singleton types are made up by taking the union of other types, using the `|` operator, like `'a | 'b | 'c`, which can be any of `'a`, `'b`, or `'c`.

```
'a: 'a; // valid
'a: 'a | 'b; // also valid
'c: 'a | 'b; // type error
```

[2]In Rust, `i32` is the type of 32-bit signed integers.

The same goes for references, pairs, and functions (which will be introduced later): the type of a reference is itself a reference, the type of a pair is itself a pair, and the type of a function is itself a function. The only consistent difference between types and terms is types must be statically known, which means they can be erased by runtime.

```
('a, 'b): ('a, 'b); // valid
('a, 'b): ('a | 'b, 'a | 'b); // also valid
```

The $*$ syntax denotes the infinite type/top, the type that contains all values. This is used to represent the concept of no typing information being available. There are three main places where this comes up:

1. Taking the union of two types which don't have a meaningful union, like pairs and atoms. `'a | ('a, 'a): *`.
2. Using it to represent the type of types, which is how you do generic functions. Polymorphic functions are defined by making a function which takes a type as input, and returns a function which uses that type.
3. The type of uninitialised/moved data.

Comptime vs Runtime

Types, just like values, can be assigned to variables for future re-use. However, they must all be statically known, which is enforced by only allowing them to be assigned to *comptime* variables, which start with capital letters. This is similar to how in Haskell types must start with a capital letter, but here the line between types and values is blurred significantly.

```
abPair = ('a | 'b, 'a | 'b); // error! type union can only occur at compile time
ABPair = ('a | 'b, 'a | 'b); // valid
('a, 'b): ABPair;
```

Functions

Functions are defined with an arrow `->` and an optional runtime body surrounded in curly braces. For instance, the identity function over `'true | 'false` is defined as such:

```
Bool = 'true | 'false;
id = (x: Bool) -> Bool { x };
```

If the runtime body is omitted, the function can only be called at compile time, which means it must be written to a comp time variable:

```
Bool = 'true | 'false;
Id = (x: Bool) -> Bool; // valid
id = (x: Bool) -> Bool; // invalid: attempt to assign comptime func to runtime var
```

The only difference between a function body and its return type is that its return type is run at compile time, there is no syntactic difference. For functions you want to run at compile time, syntax after the arrow *is* the function body.

```
Id = x -> x; // Definition of identity which can only be run at comp time
id = x -> x { x }; // Definition of identity which also exists at runtime
```

Case Statements

In Ochre, atoms can be branched on via a case statement. The discriminant of the case statement must be an atom, and there must be exactly one branch for each possible atom. In the future, I plan on adding if and match statements, which will be syntactic sugar for case statements.

```
Bool = 'true | 'false;
not = (b: Bool) -> Bool {
  case b {
    'true => 'false,
    'false => 'true,
  }
};
not('true); // 'false
```

Dependent Pairs

If a pair is being evaluated in a comptime context, the right of a pair can depend on the left. This is done by making the right a function that maps from left to right.

```
Same = (Bool, L -> L); // binds LHS to L, so can be used by right
('true, 'true): Same; // valid
('true, 'false): Same; // error! 'false is not of type 'true

Different = (Bool, L -> case L { 'true => 'false, 'false => 'true});
```

```
    ('true, 'false): Different; // valid
    ('true, 'true): Different; // error!
```

When you union together pairs, it doesn't just union together their left and right and make a new pair, it uses any information it can get from the left pair to more precisely type the right pair.

```
    Same = ('true, 'true) | ('false, 'false);
    // Expanded internally to:
    Same = ('true | 'false, L -> case L { 'true => 'true, 'false => 'false })
```

Listing 3

This makes the union operator precise, taking the union of two types should never produce a type with inhabitants that weren't in either of the types which were unioned together.

If you want to record dependence between the left and right of a pair in a runtime context, you must construct the pair without the dependence, and then use a type constraint to add it back in.

```
    Same = ('true, 'true) | ('false, 'false);
    x = ('true, 'true); // x is a non-dependent pair
    x: Same; // type constraint has made x a dependent pair
```

Type Narrowing

If the right of a pair depends on the left, and then you find something out about the left, you should in turn find something out about the right. This is done in Ochre via type *narrowing*. In the below example, we define a function `f`, and within `f` we know that the left and right of our pair `p` are the same (using the definition in Listing 3). When we match on its left with `p.0`, each branch is type-checked with the additional knowledge that we are in that particular branch. This allows the compiler to correctly identify that when matching on the other side of the pair, you only need to have one branch.

```
    Same = ('true, 'true) | ('false, 'false);
    f = (p: Same) -> Bool {
      case p.0 {
        'true => case p.1 { 'true => 'unit }, // p.1: 'true
        'false => case p.1 { 'false => 'unit }, // p.1: 'false
      }
    }
```

Listing 4: Case statements narrow down the type of their discriminant in each branch

Algebraic Data Types

Take the following definition of Peano naturals in Haskell syntax:

```
data Nat = Zero | Succ Nat
```

In Ochre this is represented by a dependent pair. The left of the pair indicates which variant the ADT is in (either zero or successor), and the right contains the payload of that variant. In the zero case, nothing is stored, so the payload is `'unit`, in the successor case, we store the natural that we are the successor of, so our payload is `Nat`.

```
// "manual" ADT encoding
Nat = (T: 'zero | 'succ, case T { 'zero => 'unit, 'succ => Nat });
// idiomatic encoding using type union
Nat = ('zero, 'unit) | ('succ, Nat);
```

By matching on the left, you can determine which variant the ADT is in, then you can access the payload through the right. For instance, this is how would define addition over Peano naturals:

```
Nat = ('zero, 'unit) | ('succ, Nat);
add = (x: Nat, y: Nat) -> Nat {
  case x.0 {
    'zero => y, // 0 + y = y
    'succ => ('succ, add(x.1, y)), // (1 + x) + y = 1 + (x + y)
  }
}
```

Recursion

The definition of `add` above won't compile because of how it does recursion. When type-checking assignments, Ochre looks at the left first to figure out what type the identifiers have. In the case of `Nat` and `add` above there are no type annotations, so it evaluates the assigned value with no extra type information.

If the programmer puts type annotations on the left of an assignment, the compiler knows at least something about the type, so it can evaluate the expression with that knowledge. This isn't required in the definition of `Nat` because you can put anything in a pair, regardless of its type, so the usage of `Nat` on the right was permissible.

In the `add` case, we need to know that `add` has type (Nat, Nat) -> Nat while evaluating the function body, so we can check that `add(x.1, y)` has type `Nat`. To introduce this, add type annotations to the left of the assignment:

```
add: (Nat, Nat) -> Nat = (x: Nat, y: Nat) -> Nat {
  // ...
}
```

This introduces repetition in the types, which we remove by adding the following syntactic sugar for the above:

```
add(x: Nat, y: Nat): Nat = {
  // ...
}
```

3.2 Type & Borrow Checking

This section aims to give the reader an intuition behind the abstract interpretation used to type-check Ochre. Specifically, it answers two questions: what is the abstract environment? And how do the various syntactic constructs modify it?

The abstract environment is a mapping from identifiers to types, although it can often look like a mapping from identifiers to values because the type of a value like `'true` is $'true$. It stores the types of both runtime and comptime variables, which are distinguished by comptime variables starting with a capital letter.

Throughout this thesis, syntax will be in monospace font `like this`, and abstract values will be in mathematical text $like\,this$.

Basic Language Features

Type-checking an Ochre program always starts with an empty environment, and every time information is gained, it is added to the abstract environment. Like so. The type of every atom `'a` is the singleton set $\{'a\}$, but it is also every superset of that singleton set like $\{'a, 'b\}$.

```
x = 'true; // {x ↦ {'true}}
y = 'hello; // {x ↦ {'true}, y ↦ {'hello}}
x = 'false; // {x ↦ {'false}, y ↦ {'hello}}
```

Listing 5: A series of assignments, and their corresponding effects on the abstract environment.

In the above example, it would be sound for the abstract environment to map x onto $\{'true, 'false\}$, or even $\{'true, 'unrelated\}$, but that would be losing information. The concept of losing typing information will be made explicit later with environment *rearrangements*, but for now, we'll focus on the environment being as precise as possible.

For brevity, we use $'a$ as syntactic sugar for the singleton set $\{'a\}$. This never causes ambiguity because the abstract environment only ever uses atoms in sets, never by themselves.

```
x = 'true; // {x ↦ 'true}
y = 'hello; // {x ↦ 'true, y ↦ 'hello}
x = 'false; // {x ↦ 'false, y ↦ 'hello}
```

Listing 6: Listing 5 but using syntactic sugar for singlton sets of atoms.

When you move a value, it is mapped to $\bot$ in the abstract environment:

```
x = 'hello; // {x ↦ 'hello}
y = x; // {x ↦ ⊥, y ↦ 'hello}
```

References & Mutation

When you construct a reference, the value is *borrowed*. In the case of mutable borrows, this means the value isn't available in the original location, which is represented in the abstract environment as $\mathsf{loan}^m\, l$ where l is the *loan identifier* for this particular loan. We set it to this instead of $\bot$ so we can find it again in the future when we want to terminate the loan. The reference will map to $\mathsf{borrow}^{\mathsf{m}}\, l\, v$ where v is the type of the value being borrowed.

```
x = 'one; // {x ↦ 'one}
rx = &mut x; // {x ↦ loan^m l, rx ↦ borrow^m l 'one}
*rx = 'two; // {x ↦ loan^m l, rx ↦ borrow^m l 'two}
// rx dropped
x; // {x ↦ 'two, rx ↦ ⊥}
```

Listing 7: A reference to a variable being constructed and used for a mutation. When the reference `rx` is dropped, the updated value from the mutable reference is written back to the original variable `x`.

```
x = 'one; // {x ↦ 'one}
rx = &mut x; // {x ↦ loan^m l, rx ↦ borrow^m l 'one}
*rx = 'two; // {x ↦ loan^m l, rx ↦ borrow^m l 'two}
// rx dropped
x; // {x ↦ 'two, rx ↦ ⊥}
```

Listing 8: A reference to a variable being constructed. When the reference is dropped, the updated value from the mutable reference is written back to the original variable.

Mutable references are similar, except the value is also stored on the loan, reflecting the fact that while an immutable loan exists, the value is still available in its original location. Having loan in an environment like this is also used to prevent mutations to a borrowed value.

```
x = 'one; // {x ↦ 'one}
rx = &x; // {x ↦ loan^s l 'one, rx ↦ borrow^s l 'one}
```

Loans can be nested, which is useful when you want to temporarily give a value you have borrowed to something else.

```
x = 'one; // {x ↦ one}
rx1 = &mut x; // {x ↦ loan^m l, rx ↦ borrow^m l 'one}
rx2 = &mut *rx1; // {x ↦ loan^m l, rx ↦ borrow^m l (loan^m l'), rx2 ↦ borrow^m l 'one}
```

Listing 9: A reborrow

When immutable references are re-borrowed, the syntactic representation of the environment grows exponentially.

```
x = 'one; // {x ↦ one}
rx1 = &x; // {x ↦ loan^s l 'one, rx ↦ borrow^s l 'one}
rx2 = &*rx1; // {x ↦ loan^s l (loan^s l' 'one), rx ↦ borrow^s l (loan^s l' 'one), rx2 ↦ borrow^s l' 'one}
rx3 = &*rx2; // {x ↦ loan^s l (loan^s l' (loan^s l'' 'one)), rx ↦ borrow^s l (loan^s l' (loan^s l'' 'one)),
             // rx2 ↦ borrow^s l' (loan^s l'' 'one), rx3 ↦ borrow^s l'' 'one}
```

Listing 10: An immutable re-borrow

This is not a problem for the implementation because the value stored in the loan and the value stored in the borrow are two pointers to the same underlying memory, it can just make working examples out by hand longer.

(Dependent) Pairs

In the abstract environment pairs store the type of the left side, and how to turn the type of the left side into the right, like so: $(\{'true, 'false\}, \texttt{L} \to \texttt{L})$. This reads "The left of the pair is of type $\{'true, 'false\}$, and the right is whatever the left is". This means in the future if the left is narrowed down to be $'true$, the right will be read as $'true$.

Non-dependent pairs are a special case of dependent pairs where the right happens to evaluate to the same type for any given left. A non-dependent pair of booleans would be constructed with `(Bool, Bool)`, which is syntactic sugar for `(Bool, _ -> Bool)`.

```
Bool = 'true | 'false;                // {Bool ↦ {'true, 'false}}
BoolPair = (Bool, Bool);              // {..., BoolPair ↦ ({'true, 'false}, _ → Bool)}
Same = (Bool, L -> L);                // {..., Same ↦ ({'true, 'false}, L → L}
specificPair = ('true, 'true);        // {..., specificPair ↦ ('true, _ → 'true)}
widenedPair = ('true, 'true): Same; // {..., widenedPair ↦ ({'true, 'false}, L → L}
```

Listing 11: Various pair constructions and their respective entries in the abstract environment

Mutation breaks type dependencies across pairs. Once the left of a pair is mutated, the right must be generalized because the data is lost, meaning the programmer will never be able to recover which specific type the right had in the future.

```
1    Same = ('true,  'true )
2         | ('false, 'false);  // {Same ↦ ({'true, 'false}, L → L)}
3    p = ('true, 'true): Same; // {Same ↦ ..., p ↦ ({'true, 'false}, L → L)}
4    p.0 = 'false;             // {Same ↦ ..., p ↦ ('false, _ → ('true | 'false))}
5    p.1 = 'false;             // {Same ↦ ..., p ↦ ('false, _ → 'false)}
6    p: Same;                  // {Same ↦ ..., p ↦ ({'true, 'false}, L → L)}
```

Listing 12: Demonstration of how mutation interacts with dependent pairs. On line 4 when the left of the pair is mutated, the dependence is broken. When the right is mutated to 'false, the pair's type is narrowed down, but it doesn't regain the dependence until the programmer explicitly widens the type on line 6.

Listing 12 depicts `('true, 'true) | ('false, 'false)` being evaluated to ({'true, 'false}, `L` → `L`), which isn't strictly true. Type union between pairs will make the right depend on the left by producing a case statement for each of the possible left atoms, so `('true, 'true) | ('false, 'false)` would instead evaluate to ({'true, 'false}, `L` → `case L { 'true => 'true, 'false => 'false }`). In code examples it often evaluates to the former, to aid readability.

Type Annotations

Sometimes you want to manually manipulate what type the abstract interpretation reads from a piece of syntax. You do this with type annotations like `M:T`. Evaluating a piece of syntax like `M: T` both asserts that type of M is a subtype of T and makes the expression be of type T instead of M.

```
1    x = 'true; // {x ↦ 'true}
2    y = 'true: 'true | 'false; // {..., y ↦ {'true, 'false}}
```

Listing 13: The type annotation has caused type information to be lost: both `x` and `y` are set to 'true in the above code, but the type annotation on `y` has caused the abstract interpretation to only be able to assign the wider type of {'true, 'false}

Comptime vs Runtime

As you will see in Section 4, there are large differences in how the abstract interpretation is performed on runtime and comptime terms; however, for the most part, they map very similarly to the abstract environment. Following from the syntax level distinction, an entry in the abstract environment is marked as runtime or comptime by the variable identifier being capitalized or not.

```
1    x = 'one; // {x ↦ 'one}
2    X = 'one; // {..., X ↦ 'one}
```

One place differences do show is that runtime variables can mutate and be moved, whereas comptime values are immutable and can be freely used like values in typical pure functional languages.

This is so the programmer doesn't have to deal with manual memory management of comptime values, which they wouldn't benefit from anyway because all comptime variables are erased by the time the code is executed.

```
1  x = 'runtime; // {x ↦ 'runtime}
2  y = x; // {x ↦ ⊥, y ↦ 'runtime}
3
4  X = 'comptime; // {..., X ↦ 'comptime}
5  Y = X; // {..., X ↦ 'comptime, Y ↦ 'comptime}
```

Listing 14: Unlike the runtime variable `x`, which becomes uninitialized after being moved to `y`, the comptime variable `X` remains accessible while the value is simultaneously used by `Y`, as you would expect from languages move semantics like Haskell

Functions

When a function is called, two things need to be calculated at the call site: whether or not the argument the programmer supplied is a subtype of the required argument; and what the return type is given this argument type. To achieve this we store two pieces of syntax, the input syntax and the return type syntax. We store syntax instead of types so the return type can depend on the input type.

```
1  Bool = 'true | 'false; // {Bool ↦ {'true, 'false}}
2  id = (b: Bool) -> Bool {
3    b // {Bool ↦ ..., id ↦ ⊤, b ↦ {'true, 'false}}
4  } // {Bool ↦ {'true, 'false}, id ↦ (b: Bool) → Bool}
```

Listing 15: While type checking the body, argument `b` is in the abstract environment. Abstractly the function is two pieces of *syntax*: `(b: Bool)` and `Bool` instead of their respective types which are both {'true, 'false}

Case Statements

In each of the branches of a case statement, the type of the scrutinee is narrowed down to a specific atom. This is useful when the case is branching over the left of a dependent pair, because when the left of the pair gets narrowed down, so does the right[3].

Each branch of the case statement will modify the environment in some possibly different

[3]The right only gets narrowed once it is accessed, not immediately when the left is narrowed

way. These are combined into one environment via an environment-wide union operation, which is the output environment.

```
1   f = (b: 'true | 'false) -> 'unit {
2                         // {b ↦ {'true, 'false}}
3     case b {
4       'true => (        // {b ↦ 'true}
5         x = 'hello;     // {b ↦ 'true, x ↦ 'hello}
6       ),
7       'false => (       // {b ↦ 'false}
8         x = 'world;     // {b ↦ 'false, x ↦ 'world}
9         b = 'true;      // {b ↦ 'true, x ↦ 'world}
10      )
11    };                  // {b ↦ 'true, x ↦ {'hello, 'world}}
12  }
```

Listing 16: Each branch of the case statement is abstractly interpreted with `b` narrowed down to a single atom (lines 4 and 7). Both branches modify `x`, but to different values which make their environments different (lines 5 and 8); these different values are unioned together in the final environment to {'hello, 'world}. The false branch happens to mutate `b` back to 'true, which means by the end of *both* branches, b : 'true, which is reflected in the final environment which maps `b` to 'true instead of {'true, 'false}.

Complex Example Programs

And last but not least, here are a few example programs which use several of the previous features together:

Chapter 4

Formal Definition

Type-checking is done via an abstract interpretation which takes a term and outputs a type for that term while modifying an abstract environment. All interpretation rules take the form $\Omega \vdash M \diamond t \dashv \Omega'$ where $\diamond$ is one of $\{ \dot{\rightsquigarrow}, \overset{(\cdot)}{\rightarrow}, \overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \dot{\leftsquigarrow} \}$, M is a term, Ω and Ω' are the abstract environments before and after, and t is the type of the value which has been read or written.

Section 4.1 introduces key concepts that are required to understand the definitions; subsequent subsections define the abstract interpretation. The reader is encouraged to skip around definitions a lot, they are laid out in tables to make finding a particular definition easier, as one might in a repository of code.

4.1 Modalities

It is best to conceptualize the arrows as a single interpretation with many modalities. The following arrows are used to denote the different modalities:

	Read	Write
Runtime destructive	$\overset{(\cdot)}{\Rightarrow}$ // *move*	$\overset{(\cdot)}{\Leftarrow}$ // *write*
Runtime non-destructive	$\overset{(\cdot)}{\rightarrow}$ // *read*	$\overset{(\cdot)}{\leftarrow}$ // *type narrow*
Comptime	$\dot{\rightsquigarrow}$ // *erased read*	$\dot{\leftsquigarrow}$ // *erased write*

Figure 4.1: Exhaustive table of the interpretations which make up Ochre

To summarize the modalities: a dot above the arrow denotes *abstract* interpretation which means it is not executing the code, only type-checking it; squiggly arrows denote the interpretation of terms that are erased at compile time; arrows which point rightwards (away from

the term) denote reads, arrows which point leftwards (towards the term) denote writes. These modalities are elaborated on below.

Comptime vs Runtime Modality

There are two distinct types of terms in Ochre: runtime terms, and comptime terms.

Runtime terms are constrained such that they can be executed efficiently on hardware. This involves manual memory management with move semantics, so no garbage collector is required at runtime, and allows the in-place mutation of data structures.

Comptime terms are erased at compile time and are only used to compute types. Because comptime terms only exist during compilation, and not during runtime, inefficient but automatic memory management strategies can be used, such as reference counting. This removes the need for move semantics, which allows types to be used multiple times without explicit copying.

Because comptime terms do not have move semantics, we cannot have mutation[1], and we do not need immutable references. Not supporting mutation within comptime terms has the added benefit of the programmer not having to reason about the side effects of evaluating types, which happens implicitly in situations including type checking a function call site (see ⟨*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* MN⟩).

There is no syntactic distinction between comptime and runtime terms because they are so similar, although there could have been because one can always determine whether a term is comptime or runtime given its position.

Abstract vs Concrete Modality

Comptime interpretations ($\dot{\rightsquigarrow}$, $\dot{\leftsquigarrow}$) are only ever abstract, whereas runtime interpretations ($\overset{(\cdot)}{\Rightarrow}$, $\overset{(\cdot)}{\Leftarrow}$, $\overset{(\cdot)}{\leftarrow}$, $\overset{(\cdot)}{\rightarrow}$) can have an optional dot above them; this dot means "abstract interpret".

Execution and type checking are very closely related in Ochre because execution is just the totally precise version of type checking. They differ only in how they treat type annotations, and how they treat functions. Abstract interpretation will evaluate $M : T$ to the result of evaluating T, and concrete interpretation will evaluate the same syntax to the result of evaluating M. Abstract interpretation will evaluate MN by interpreting the return term of the function M evaluates to, whereas concrete interpretation will execute the function's body.

This guarantees concrete interpretation will output a precise result (singleton type) because the only way to form non-singleton types is via type union, which can only occur on the right-hand side of a type annotation, or function return type.

[1]The method of combining mutability with dependent types this research uses relies on move semantics, therefore we cannot have mutability on non-move semantics code.

Def. 4.1.1: Concrete and Abstract Arrow Designations

$$\forall \diamond \in \{ \dot{\rightsquigarrow}, \dot{\rightarrow}, \dot{\Rightarrow}, \dot{\Leftarrow}, \dot{\leftarrow}, \dot{\rightsquigarrow} \} \left[\frac{}{\mathsf{abstract}\,\diamond} \right] \qquad \forall \diamond \in \{ \rightarrow, \Rightarrow, \Leftarrow, \leftarrow \} \left[\frac{}{\mathsf{concrete}\,\diamond} \right]$$

Read vs Write Modality

In typical languages like the λ-calculus, terms are evaluated to a value, which is equivilant to the notion of the read modality in Ochre. The write modality allows you to write a value to a term, which is how variables are brought into scope. You might find it useful to compare reading a variable x with writing to a variable x:

$$\frac{\langle \textit{def. } \overset{(\cdot)}{\Rightarrow} \textit{ for } x \rangle \quad \Omega' = \Omega \left[\frac{x \mapsto \top}{x \mapsto v} \right]}{\Omega \vdash x \overset{(\cdot)}{\Rightarrow} m \dashv \Omega'} \qquad \frac{\langle \textit{def. } \overset{(\cdot)}{\Leftarrow} \textit{ for } x \rangle \quad \Omega' = \Omega \left[\frac{x \mapsto v}{x \mapsto \top} \right]}{\Omega \vdash x \overset{(\cdot)}{\Leftarrow} v \dashv \Omega'}$$

Figure 4.2: Reading removes a value from the environment, whereas writing adds a value.

Defining the write operation for more complex pieces of syntax is how several language features are defined, including but not limited to: pattern matching, destructuring, and specifying function arguments.

Def. 4.1.2: Read/Write Arrow Designations

$$\forall \diamond \in \{ \overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\rightarrow}, \dot{\rightsquigarrow} \} \left[\frac{}{\mathsf{read}\,\diamond} \right] \qquad \forall \diamond \in \{ \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \dot{\rightsquigarrow} \} \left[\frac{}{\mathsf{write}\,\diamond} \right]$$

4.2 Syntax

Ochre grammar:

$$
\begin{array}{lll}
S & ::= & \text{// statement} \\
 & \quad M & \text{// expression} \\
 & \quad M = N; S & \text{// assignment} \\
 & \quad \texttt{match}\ M\ \{\ \overrightarrow{M'\ \texttt{=>}\ S}\ \} & \text{// match statement} \\
 & & \\
T, U, M, N & ::= & \text{// expression} \\
 & \quad x \mid y \mid z & \text{// runtime variable identifier} \\
 & \quad X \mid Y \mid Z & \text{// comptime variable identifier} \\
 & \quad \texttt{'a} & \text{// atom construction} \\
 & \quad M\texttt{,}(T\texttt{->}\ )N & \text{// pair construction} \\
 & \quad M\texttt{.0} & \text{// pair left access} \\
 & \quad M\texttt{.1} & \text{// pair right access} \\
 & \quad *M & \text{// dereference} \\
 & \quad \&M \mid \texttt{\&mut}\ M & \text{// borrow constructor} \\
 & \quad MN & \text{// application} \\
 & \quad M\ \texttt{->}\ N\ (\{\ N\ \}) & \text{// abstraction (optional runtime body)} \\
 & \quad \_ & \text{// uninitialised} \\
 & \quad T\ \texttt{|}\ U & \text{// type union} \\
 & \quad M\texttt{:}\ T & \text{// type constraint} \\
 & \quad v & \text{// type/value}
\end{array}
$$

Figure 4.3: Ochre syntax

Assignments never occur in a terminal position. This avoids the question of what is the return value of an assignment.

Match statements always occur in a terminal position. RustBelt's λ_{Rust} Jung et al. [2018] has the same restriction, but Aeneas' LLBC Ho and Protzenko [2022] does not. If it was permitted for operations to occur after a match statement, the environment *after* the match statement would have to be calculated, which would be the type union of the environments produced by the branches. For this type union operation to be precise over environments, like it is on pairs, we would have to support dependencies between variables. We do not support such dependencies for the sake of simplicity, and thus cannot precisely define environment union.

This restriction does not limit what programs can be represented because a program with match statements in non-terminal positions can be re-written to one that only has matches in terminal positions. This could be done by moving everything after the match statement into each of the match statement branches:

$$
\forall \diamond \left\{ \overset{(\cdot)}{\Rightarrow}, \dot{\rightsquigarrow} \right\}. \left[\frac{\Omega \vdash \texttt{match}\ M\ \texttt{\{}\ \overrightarrow{M'\ \texttt{=>}\ S; S'}\ \texttt{\}}\ \diamond t}{\Omega \vdash \texttt{match}\ M\ \texttt{\{}\ \overrightarrow{M'\ \texttt{=>}\ S}\ \texttt{\}}\ ; S' \diamond t} \right]
$$

Figure 4.4: A re-write rule which could enable matches in non-terminal positions.

This rewrite rule has not been included in Ochre because I intend to support dependence between variables in the future, and therefore precise environment union, which removes the need for the rewrite. This rewrite rule has the disadvantage of causing an exponential blowup in code size/interpretation derivation size.

Assignment and match statements are the only constructs that can narrow types in the environment, so by having them both in statement (S) instead of expression (M), we guarantee that expression evaluation only ever widens types. This simplifies type-checking expressions.

Types/values can be treated as syntax. This syntactic construction cannot be constructed by the programmer. It is constructed internally within the interpretations to make syntax that always returns the same type/value, such as in $\langle$*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* M `.0`$\rangle$ where it is used to break the dependence of the pair so the left value can be moved out.

4.3 Environment, Values, and Types

The abstract environment, and abstract values:

$$
\begin{array}{lll}
\Omega & ::= & \text{// abstract environment (stack)} \\
 & \quad \emptyset & \text{// empty stack} \\
 & \quad \Omega, x \mapsto v & \text{// runtime variable} \\
 & \quad \Omega, X \mapsto v & \text{// comptime variable} \\
 & \quad \Omega, l \mapsto v & \text{// loan restriction} \\
 & & \\
m, n, v, w, t, u & ::= & \text{// type/value} \\
 & \quad \{\overrightarrow{\text{'a}}\} & \text{// atom} \\
 & \quad (v, T \to U) & \text{// pair} \\
 & \quad (T \to U) & \text{// function} \\
 & \quad \mathsf{borrow}^{\mathsf{s}}\, l\, v \mid \mathsf{borrow}^{\mathsf{m}}\, l\, v & \text{// reference} \\
 & \quad \mathsf{loan}^{\mathsf{s}}\, l\, v \mid \mathsf{loan}^{m}\, l & \text{// referenced value} \\
 & \quad \top & \text{// top}
\end{array}
$$

Figure 4.5: Abstract/Concrete Environment and Types

The Top Type

The $\top$ type is used to denote a lack of typing information. Every type/value is of type $\top$. When you move a value, the previous location is set to $\top$, to denote uninitialized data. When a value has never been written to before, its value is $\top$, again, to denote uninitialized data. When a type t depends on another type u, but u has not been narrowed down enough to deduce the type of t, t evaluates to $\top$ to denote the lack of typing information.

Concrete Values

If a type is a singleton type (a type with one inhabitant), it is referred to as a value. For example {'a} is a concrete value/type, but {'a, 'b} is not. Figure 4.3 shows the formal definition of concrete values. Concrete values and non-singleton types share a grammar because rules are typically generic over both and preserve a values concreteness, so combining them avoids the syntatic overhead of introducing an additional modality.

Drop Operation

When an operation is no longer used, it must be dropped. At runtime, dropping a value will free it's associated memory, allowing it to be used for other operations, which is why Ochre doesn't need a garbage collector. Dropping a reference to a value removes the restrictions created by that reference. Drop is defined in Figure 4.3.

Def. 4.3.1: Drop and Concrete Operations

v	$\Omega \vdash \mathsf{drop}\, v \dashv \Omega'$	$\mathsf{concrete}\, v$
$\{\overrightarrow{\text{'a}}\}$	$\dfrac{}{\Omega \vdash \mathsf{drop}\, \{\overrightarrow{\text{'a}}\}}$	$\dfrac{}{\mathsf{concrete}\, \{\text{'a}\}}$
$(v, T \to U)$	$\dfrac{\Omega \vdash T \dot{\leftsquigarrow} v \dashv \Omega' \quad \Omega' \vdash U \dot{\rightsquigarrow} w \quad \Omega \vdash \mathsf{drop}\, v \dashv \Omega'' \quad \Omega'' \vdash \mathsf{drop}\, w \dashv \Omega'''}{\Omega \vdash \mathsf{drop}\, (v, T \to U) \dashv \Omega'}$	$\dfrac{\mathsf{concrete}\, v \quad \mathsf{concrete}\, (T \to U)}{\mathsf{concrete}\, (v, T \to U)}$
$(T \to U)$	$\dfrac{}{\Omega \vdash \mathsf{drop}\, (T \to U)}$	$\dfrac{\mathsf{runtime}\, T \quad \mathsf{runtime}\, U}{\mathsf{concrete}\, (T \to U)}$
$\mathsf{borrow}^{\mathsf{s}\|\mathsf{m}}\, l\, v$	$\dfrac{\Omega' = \Omega \left[\dfrac{v}{\mathsf{loan}^{\mathsf{s}\|\mathsf{m}}\, l\, (v)}\right]}{\Omega \vdash \mathsf{drop}\, (\mathsf{borrow}^{\mathsf{s}\|\mathsf{m}}\, l\, v) \dashv \Omega'}$	$\dfrac{\mathsf{concrete}\, v}{\mathsf{concrete}\, (\mathsf{borrow}^{\mathsf{s}}\, l\, v)}$
	$\dfrac{\Omega' = \Omega \backslash \{l \mapsto v'\} \quad \Omega' \vdash v \sqsubseteq v'}{\Omega \vdash \mathsf{drop}\, (\mathsf{borrow}^{\mathsf{s}\|\mathsf{m}}\, l\, v) \dashv \Omega'}$	
$\mathsf{loan}^{\mathsf{s}\|\mathsf{m}}\, l\, (v)$	$\dfrac{\Omega' = \Omega \left[\dfrac{\top}{\mathsf{borrow}^{\mathsf{s}\|\mathsf{m}}\, l\, v}\right] \quad \Omega' \vdash \mathsf{drop}\, v \dashv \Omega''}{\Omega \vdash \mathsf{drop}\, (\mathsf{loan}^{\mathsf{s}\|\mathsf{m}}\, l\, (v)) \dashv \Omega''}$	$\dfrac{\mathsf{concrete}\, v}{\mathsf{concrete}\, (\mathsf{loan}^{\mathsf{s}}\, l\, v)}$
$\mathsf{loan}^{m}\, l$		$\dfrac{}{\mathsf{concrete}\, (\mathsf{loan}^{m}\, l)}$

Environment Rearrangement

At any point during a program interpretation, whether it be abstract or concrete interpretation, the environment can be *rearranged*, a technique introduced by Aeneas Ho and Protzenko [2022]. Environment rearranges can be inserted before or after any of the interpretation judgements.

Def. 4.3.2: Environment Rearrangement

$$\textsc{Allocation}\ \frac{\Omega' = \Omega, xX \mapsto \top}{\Omega \hookrightarrow \Omega'} \qquad \textsc{Deallocation}\ \frac{\Omega', x \mapsto \top = \Omega}{\Omega \hookrightarrow \Omega'} \qquad \textsc{Type-Widen}\ \frac{\Omega' = \Omega\left[\frac{x \mapsto v'}{x \mapsto v}\right] \quad \Omega \vdash v \sqsubseteq v'}{\Omega \hookrightarrow \Omega'} \qquad \textsc{Drop}\ \frac{\Omega' = \Omega\left[\frac{x \mapsto \top}{x \mapsto v}\right] \quad \Omega' \vdash \mathsf{drop}\, v \dashv \Omega''}{\Omega \hookrightarrow \Omega''}$$

$$\forall \diamond \in \{ \dashrightarrow, \overset{(\cdot)}{\rightarrow}, \overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \dashleftarrow \} \left[\begin{array}{c} \textsc{Rearrange-Before} \\ \dfrac{\Omega \hookrightarrow \Omega' \quad \Omega' \vdash M \diamond v \dashv \Omega''}{\Omega \vdash M \diamond v \dashv \Omega''} \end{array}\right]$$

$$\forall \diamond \in \{ \dashrightarrow, \overset{(\cdot)}{\rightarrow}, \overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \dashleftarrow \} \left[\begin{array}{c} \textsc{Rearrange-After} \\ \dfrac{\Omega \vdash M \diamond v \dashv \Omega' \quad \Omega' \hookrightarrow \Omega''}{\Omega \vdash M \diamond v \dashv \Omega''} \end{array}\right]$$

Allocation - Before a variable is used, including before it is first written to, it must be mapped to $\top$ in the environment. Allocation takes a variable previously not in the environment, and maps it to $\top$.

Deallocation - Occasionally typing judgements will assert that a series of operations leave the environment back in its original state (see ⟨*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* M -> T { N }⟩). In order to achieve this variables allocated in that series of operations must be deallocated.

Type Widening - At any point during the interpretation it is valid to forget typing information. For example: if a value is known to be one of {'a, 'b}, it is valid to now consider it to be one of {'a, 'b, 'c}.

Dropping - Before deallocation, values must be dropped. This is achieved in derivations by inserting rearrangements which drop values.

Def. 4.3.3: Environment Helpers

Ω	$\Gamma \vdash \mathsf{comptime}$	$\Delta \vdash \mathsf{concrete}$	$\Omega \vdash \mathsf{drop}$	$\Omega \sqsubseteq \Omega'$
$\emptyset$	$\dfrac{}{\emptyset \vdash \mathsf{comptime}}$	$\dfrac{}{\emptyset \vdash \mathsf{concrete}}$	$\dfrac{}{\emptyset \vdash \mathsf{drop}}$	$\dfrac{}{\Omega \sqsubseteq \emptyset}$
$\Omega, x \mapsto t$		$\dfrac{\mathsf{concrete}\, t \quad \Omega \vdash \mathsf{concrete}}{\Omega, x \mapsto t \vdash \mathsf{concrete}}$	$\dfrac{\Omega \vdash \mathsf{drop}\, t \dashv \Omega' \quad \Omega' \vdash \mathsf{drop}}{\Omega, x \mapsto t \vdash \mathsf{drop}}$	$\dfrac{x \mapsto t \in \Omega \quad \Omega \vdash t \sqsubseteq t' \quad \Omega \sqsubseteq \Omega'}{\Omega \sqsubseteq \Omega', x \mapsto t'}$
$\Omega, X \mapsto t$	$\dfrac{\Omega \vdash \mathsf{comptime}}{\Omega, X \mapsto t \vdash \mathsf{comptime}}$	$\dfrac{\Omega \vdash \mathsf{concrete}}{\Omega, X \mapsto t \vdash \mathsf{concrete}}$	$\dfrac{\Omega \vdash \mathsf{drop}}{\Omega, X \mapsto t \vdash \mathsf{drop}}$	$\dfrac{X \mapsto t \in \Omega \quad \Omega \vdash t \sqsubseteq t' \quad \Omega \sqsubseteq \Omega'}{\Omega \sqsubseteq \Omega', X \mapsto t'}$
$\Omega, l \mapsto t$	*// all environment operations ignore loan restrictions*			

Comptime - An environment is comptime iff it only contains comptime variables.

Concrete - An environment is concrete iff every runtime value within it is concrete. This does not cause problems for the soundness proof because concrete interpretation never reads comptime variables.

Drop - Drops every runtime variable. Does not drop comptime ones.

Subtype - If a variable is not mapped to something in the environment, it is implicitly mapped to $\top$, so $\emptyset$ is the supertype of every environment. This is the base case. It then takes variables off the super environment one by one, making sure each one is a supertype of its equivalent in the sub environment.

4.4 Type Operations

This section defines subtyping, type intersection, and type union. These operations are used throughout the interpretations and discussions of properties.

Def. 4.4.1: Subtype

v	$t \sqsubseteq u$
$\{\overrightarrow{'a}\}$	$\dfrac{\{\overrightarrow{'a}\} \subseteq \{\overrightarrow{'b}\}}{\Omega \vdash \{\overrightarrow{'a}\} \sqsubseteq \{\overrightarrow{'b}\}}$
$(v, T \to U)$	$\dfrac{\begin{array}{c}\Omega \vdash t \sqsubseteq t' \\ \Omega \vdash \mathsf{comptime} \dashv \Gamma \\ \Gamma \vdash T' \dot{\leftsquigarrow} t \dashv \Gamma' \\ \Gamma' \vdash T \dot{\leftsquigarrow} t \dashv \Gamma'' \\ \Gamma'' \vdash S \sqsubseteq S' \dot{\rightsquigarrow} v\end{array}}{\Omega \vdash (t, T \to S) \sqsubseteq (t', T' \to S')}$
$(T \to S)$	*// note: function domains must be equal* $\dfrac{\begin{array}{c}\Omega \vdash \mathsf{comptime} \dashv \Gamma \\ \Gamma \vdash M \underset{\mathsf{max}}{\Leftarrow} m_{max} \dashv \Gamma' \\ \Gamma' \vdash S_t \sqsubseteq S'_t \dot{\rightsquigarrow} t\end{array}}{\Omega \vdash M \to S_t \sqsubseteq t \to S'_t}$
$\mathsf{borrow}^{\mathsf{s}}\, l\, v$	$\dfrac{\Omega \vdash v \sqsubseteq v'}{\Omega \vdash \mathsf{borrow}^{\mathsf{s}}\, l\, v \sqsubseteq \mathsf{borrow}^{\mathsf{s}}\, l\, v'}$
$\mathsf{borrow}^{\mathsf{m}}\, l\, v$	$\dfrac{\Omega \vdash v \sqsubseteq v'}{\Omega \vdash \mathsf{borrow}^{\mathsf{m}}\, l\, v \sqsubseteq \mathsf{borrow}^{\mathsf{m}}\, l\, v'}$
$\mathsf{loan}^{\mathsf{s}}\, l\, v$	$\dfrac{\Omega \vdash v \sqsubseteq v'}{\Omega \vdash \mathsf{loan}^{\mathsf{s}}\, l\, v \sqsubseteq \mathsf{loan}^{\mathsf{s}}\, l\, v'}$
$\mathsf{loan}^{m}\, l$	$\dfrac{}{\Omega \vdash \mathsf{loan}^{m}\, l \sqsubseteq \mathsf{loan}^{m}\, l}$

Def. 4.4.2: Type Union and Intersection

v	$v = t \sqcup u$	$v = t \sqcap u$
$\{\overrightarrow{\text{'a}}\}$	$\dfrac{v = \{\overrightarrow{\text{'a}}\} \uplus \{\overrightarrow{\text{'b}}\}}{\Omega \vdash v = \{\overrightarrow{\text{'a}}\} \sqcup \{\overrightarrow{\text{'b}}\}}$	$\dfrac{v = \{\overrightarrow{\text{'a}}\} \cap \{\overrightarrow{\text{'b}}\}}{\Omega \vdash v = \{\overrightarrow{\text{'a}}\} \sqcap \{\overrightarrow{\text{'b}}\}}$
$(v, T \to U)$	$\dfrac{\begin{array}{c}\Omega \vdash v'' = v \sqcup v' \\ \Omega \vdash (T'' \to U'') = (T \to U) \sqcup (T' \to U') \\ w = (v'', T'' \to U'')\end{array}}{\Omega \vdash w = (v, T \to U) \sqcup (v', T' \to U')}$	
$(T \to S)$	$\dfrac{\begin{array}{c}v = (\texttt{(L: } T \texttt{ | } T'\texttt{)} \to \texttt{match L \{} \\ T \texttt{ => } S\texttt{, } T' \texttt{ => } S' \texttt{ \}})\end{array}}{\Omega \vdash v = (T \to S) \sqcup (T' \to S')}$	
$\mathsf{borrow}^{\mathsf{s}}\, l\, v$	$\dfrac{\Omega \vdash t = v \sqcup v'}{\Omega \vdash \mathsf{borrow}^{\mathsf{s}}\, l\, t = \mathsf{borrow}^{\mathsf{s}}\, l\, v \sqcup \mathsf{borrow}^{\mathsf{s}}\, l\, v'}$	
$\mathsf{borrow}^{\mathsf{m}}\, l\, v$	$\dfrac{\Omega \vdash t = v \sqcup v'}{\Omega \vdash \mathsf{borrow}^{\mathsf{m}}\, l\, t = \mathsf{borrow}^{\mathsf{m}}\, l\, v \sqcup \mathsf{borrow}^{\mathsf{m}}\, l\, v'}$	
$\mathsf{loan}^{\mathsf{s}}\, l\, v$	$\dfrac{\Omega \vdash t = v \sqcup v'}{\Omega \vdash \mathsf{loan}^{\mathsf{s}}\, l\, t = \mathsf{loan}^{\mathsf{s}}\, l\, v \sqcup \mathsf{loan}^{\mathsf{s}}\, l\, v'}$	
$\mathsf{loan}^{m}\, l$	$\dfrac{}{\Omega \vdash \mathsf{loan}^{m}\, l = \mathsf{loan}^{m}\, l \sqcup \mathsf{loan}^{m}\, l}$	

Def. 4.4.3: Type Operator

$$\frac{\begin{array}{c}\mathsf{concrete}\, v \\ \Omega \vdash v \sqsubseteq t\end{array}}{\Omega \vdash v : t}$$

4.5 Interpretations

This section defines the abstract and concrete interpretations which define Ochre's type checking and runtime semantics respectively.

Definitions are identified by $\langle \textit{def.}M \textit{for } \to \rangle$, where $\to$ is the arrow being defined, and M, is the piece of syntax it is being defined for. For example $\langle \textit{def.}MN \textit{for } \overset{(\cdot)}{\Rightarrow} \rangle$ refers to the rule for destructively reading a function application (potentially abstractly). The reader is encouraged to refer to Table 4.1 while reading the interpretation definitions to look up the meaning of arrows.

The definitions of the interpretations are laid out in tables. Each cell defines the interpretation of a kind of term for a single combination of modalities. For example, a single cell might define $\overset{(\cdot)}{\Rightarrow}$ for function application.

The position of a definition within a definition table does not encode any information. The table layout serves entirely to aid definition lookup.

For some syntactic constructs, the definition for one modality is identical to it's definition for another; to avoid repetition multiple interpretations can be defined for a single construct at once by quantifying over the arrow being defined. An extreme example of this is the atom constructor, which is defined identically for all 6 arrows:

$$\forall \diamond \in \{\, \dot{\rightsquigarrow}\,,\ \overset{(\cdot)}{\rightarrow}\,,\ \overset{(\cdot)}{\Rightarrow}\,,\ \overset{(\cdot)}{\Leftarrow}\,,\ \overset{(\cdot)}{\leftarrow}\,,\ \dot{\leftsquigarrow}\,\}.\ \left[\frac{\langle \mathit{def}.everything\mathit{for}\,\text{'A}\rangle}{\Omega \vdash \texttt{'a} \diamond {}'a}\right]$$

Figure 4.6: An example of quantification over all interpretation arrows

Not every syntactic construct is defined for every interpretation, which determines which constructions are permitted in which places in a program. For example, comptime variable identifiers cannot be used in runtime terms, so the runtime arrows are not defined for comptime variable identifiers: $\langle \mathit{def}.X\mathit{for}\ \dot{\rightsquigarrow}\,\rangle$ exists but $\langle \mathit{def}.X\mathit{for}\ \ \dot{\rightarrow}\,\rangle$ does not.

The following table shows which interpretations are defined for which constructs, and where to find its definition:

M	$\dot{\rightsquigarrow}$	$\overset{(\cdot)}{\rightarrow}$	$\overset{(\cdot)}{\Rightarrow}$	$\overset{(\cdot)}{\Leftarrow}$	$\overset{(\cdot)}{\leftarrow}$	$\dot{\leftsquigarrow}$
// base expressions			Section 4.5.1			
`'`a	✓	✓	✓	✓	✓	✓
`_`	✓	✓	✓	✓	✓	✓
x	✓	✓	✓	✓	✓	✓
X	✓					✓
// references			Section 4.5.2			
`*`M		✓	✓	✓	✓	
`&`M	✓		✓			
`&mut` M	✓		✓			
// functions			Section 4.5.3			
$M\,N$	✓		✓			
M `->` T `{` N `}`			✓			
M `->` N	✓					
// pairs			Section 4.5.4			
M`,`N	✓	✓	✓	✓	✓	✓
M`.0`	✓	✓	✓	✓	✓	✓
M`.1`	✓	✓	✓	✓	✓	✓
// types			Section 4.5.5			
M`:`T	✓	✓	✓	✓	✓	✓
T `|` U	✓					
t	✓					

S	$\dot{\rightsquigarrow}$	$\overset{(\cdot)}{\Rightarrow}$
// statements	Section 4.5.6	
M	✓	✓
$M = N; S$	✓	✓
`match` M `{` $\overrightarrow{M'\ \texttt{=>}\ S}$ `}`	✓	✓

M	$\underset{\max}{\Leftarrow}$	$\underset{\max}{\dot{\leftsquigarrow}}$
// expressions	Section **??**	
`'`a	✓	✓
`_`	✓	✓
x	✓	✓
X		✓
`&`M	✓	✓
`&mut` M	✓	✓
M`,`N	✓	✓
M`:`T	✓	✓

Figure 4.7: Interpretation Definition Lookup Table

Each of the sections in the above tables will define the typing judgments formally and explain their definition.

4.5.1 Base Expressions

Base expressions are the simplest form of expression in Ochre. They do not themselves contain expressions, so the definitions of their interpretations do not rely on any other interpretations.

Def. 4.5.1: Variable Read Interpretations

M	$M \dot{\rightsquigarrow} t$	$M \overset{(\cdot)}{\rightarrow} t$	$M \overset{(\cdot)}{\Rightarrow} t$
xX	$\dfrac{xX \mapsto t \in \Omega}{\Omega \vdash xX \dot{\rightsquigarrow} t}$	$\dfrac{x \mapsto t \in \Omega}{\Omega \vdash x \overset{(\cdot)}{\rightarrow} t}$	$\dfrac{\Omega' = \Omega\left[\dfrac{x \mapsto \top}{x \mapsto v}\right]}{\Omega \vdash x \overset{(\cdot)}{\Rightarrow} m \dashv \Omega'}$

Def. 4.5.2: Variable Write Interpretations

M	$M \overset{(\cdot)}{\Leftarrow} t$	$M \overset{(\cdot)}{\leftarrow} t$	$M \dot{\leftsquigarrow} t$
xX	$\dfrac{\Omega' = \Omega\left[\dfrac{x \mapsto v}{x \mapsto \top}\right]}{\Omega \vdash x \overset{(\cdot)}{\Leftarrow} v \dashv \Omega'}$	$\dfrac{\Omega' = \Omega\left[\dfrac{x \mapsto v'}{x \mapsto v}\right] \quad m' : m}{\Omega \vdash x \overset{(\cdot)}{\leftarrow} v' \dashv \Omega'}$	$\dfrac{\Omega' = \Omega\left[\dfrac{xX \mapsto v'}{xX \mapsto v}\right] \quad m' : m}{\Omega \vdash xX \dot{\leftsquigarrow} v' \dashv \Omega'}$

The definition of the various interpretations of variables are a great demonstration of the differences between the different modalities. Compare ⟨*def.* $\Rightarrow$ *for* x⟩ with ⟨*def.* $\dot{\rightarrow}$ *for* x⟩: when a variable is moved, its value is replaced with $\top$ in the environment to reflect the fact it has been moved, but when it is only read, the environment remains unchanged.

Comptime interpretations (squiggly arrows) operate on comptime variables (upper case) as well as runtime variables (lower case), whereas runtime interpretations (straight arrows) can only operate on runtime variables.

Note: you can only write a value to a variable if that variable is currently mapped to $\top$. This forces the value you are over-writing to be dropped via an environment rearrangement before writing to it.

Def. 4.5.3: Constant Expression Interpretations

M	$M \dot{\rightsquigarrow} t$ $M \overset{(\cdot)}{\rightarrow} t$ $M \overset{(\cdot)}{\Rightarrow} t$ $M \overset{(\cdot)}{\Leftarrow} t$ $M \overset{(\cdot)}{\leftarrow} t$ $M \dot{\leftsquigarrow} t$
`'a`	$\forall \diamond \in \{\dot{\rightsquigarrow}, \overset{(\cdot)}{\rightarrow}, \overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \dot{\leftsquigarrow}\}. \left[\dfrac{}{\Omega \vdash \texttt{'a} \diamond {}'a}\right]$
`_`	$\forall \diamond \in \{\dot{\rightsquigarrow}, \overset{(\cdot)}{\rightarrow}, \overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \dot{\leftsquigarrow}\}. \left[\dfrac{}{\Omega \vdash \_ \diamond \top}\right]$

Atoms - Reading an atom gives you the singleton type (value) of that atom. Writing to an atom does not modify the environment, but it only works if the atom being written matches, so it is useful when you want to restrict the circumstances under which a write works. Match statements use this to determine under what circumstances each branch is executed: see ⟨*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* `match`⟩.

Underscore/Top - Reading from an underscore always gives you $\top$, which means no

typing information. This is how the programmer constructs uninitialized data. $\top$ is also the type of all types, so it is how you explicitly declare that a variable is a type, like when making generic functions. Writing to an underscore drops the value, which is useful for ignoring the result of a function call.

4.5.2 References

Borrow checking in Ochre occurs in the rules for interpretting references. `&` constructs references, and $*$ eliminates references (*dereferencing*).

Def. 4.5.4: Dereference Interpretations

M	$M \overset{(\cdot)}{\to} t$	$M \overset{(\cdot)}{\Rightarrow} t$
$*M$	$\dfrac{\Omega \vdash M \overset{(\cdot)}{\to} \mathsf{borrow^s}\, l\, v}{\Omega \vdash *M \overset{(\cdot)}{\to} v}$	
	$\dfrac{\Omega \vdash M \overset{(\cdot)}{\to} \mathsf{borrow^m}\, l\, v}{\Omega \vdash *M \overset{(\cdot)}{\to} v}$	$\dfrac{\Omega \vdash M \overset{(\cdot)}{\Rightarrow} \mathsf{borrow^m}\, l\, v \dashv \Omega' \quad \Omega' \vdash M \overset{(\cdot)}{\Leftarrow} \mathsf{borrow^m}\, l\, \bot \dashv \Omega''}{\Omega \vdash *M \overset{(\cdot)}{\Rightarrow} v \dashv \Omega''}$

M	$M \overset{(\cdot)}{\Leftarrow} t$	$M \overset{(\cdot)}{\leftarrow} t$
$*M$		$\dfrac{\Omega \vdash M \overset{(\cdot)}{\to} \mathsf{borrow^s}\, l\, v' \dashv \Omega' \quad \Omega'' = \Omega'[\mathsf{loan^s}\, l\, v'/\mathsf{loan^s}\, l\, v] \quad v' : v}{\Omega \vdash *M \overset{(\cdot)}{\leftarrow} v' \dashv \Omega'}$
	$\dfrac{\Omega \vdash M \overset{(\cdot)}{\Rightarrow} \mathsf{borrow^m}\, l\, \bot \dashv \Omega' \quad \Omega' \vdash M \overset{(\cdot)}{\Leftarrow} \mathsf{borrow^m}\, l\, v \dashv \Omega''}{\Omega \vdash *M \overset{(\cdot)}{\Leftarrow} v \dashv \Omega''}$	$\dfrac{\Omega \vdash M \overset{(\cdot)}{\to} \mathsf{borrow^m}\, l\, v' \dashv \Omega' \quad \Omega'' = \Omega'[\mathsf{loan^s}\, l\, v'/\mathsf{loan^s}\, l\, v] \quad v' : v}{\Omega \vdash *M \overset{(\cdot)}{\leftarrow} v' \dashv \Omega'}$

There is no syntactic distinction between a mutable and an immutable dereference because that can be determined by the type of the expression being dereferenced. Destructive operations are not defined for immutable operations, because that would break AXM[2].

Many types contain other types, in this case, the mutable reference type contains the type of the value being referenced. In these cases, move is typically defined by moving the entire type out of the context, and then writing it back with an inner value. $\langle$*def.* $\overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\Leftarrow}$ *for* $*M\rangle$ both do this in the above definition and $\langle$*def.* $\overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\Leftarrow}$ *for* $M\texttt{.0}, M\texttt{.1}\rangle$ are good examples of this pattern in other language constructs.

[2] aliasing xor mutability

Type narrowing is permitted even via *immutable* references ($\langle$*def.* $\overset{(\cdot)}{\leftarrow}$ *for* $*M\rangle$), despite causing a mutation to the environment. This is because type narrows don't occur when the user writes to a variable, they are only used by match statements to narrow the type of the scrutinee down to the appropriate branch.

Comptime terms do not use move semantics, so there is no need for borrowing, so dereference is not defined for comptime interpretations.

Def. 4.5.5: Reference Construction Interpretations

M	$M \dot{\rightsquigarrow} t$	$M \overset{(\cdot)}{\Rightarrow} t$
$\&M$	$\dfrac{\Omega \vdash M \dot{\rightsquigarrow} t \dashv \Omega' \quad \Omega'' = \Omega', l \mapsto t}{\Omega \vdash \&M \dot{\rightsquigarrow} \mathsf{borrow}^{\mathsf{s}}\, l\, t \dashv \Omega'}$	$\dfrac{\Omega \vdash M \overset{(\cdot)}{\rightarrow} t \quad \Omega \vdash M \overset{(\cdot)}{\Leftarrow} \mathsf{loan}^{\mathsf{s}}\, l\, t \dashv \Omega'}{\Omega \vdash \&M \overset{(\cdot)}{\Rightarrow} \mathsf{borrow}^{\mathsf{s}}\, l\, t \dashv \Omega'}$
$\&\mathsf{mut}\ M$	$\dfrac{\Omega \vdash M \dot{\rightsquigarrow} t \dashv \Omega' \quad \Omega'' = \Omega', l \mapsto t}{\Omega \vdash \&\mathsf{mut}\ M \dot{\rightsquigarrow} \mathsf{borrow}^{\mathsf{m}}\, l\, t \dashv \Omega'}$	$\dfrac{\Omega \vdash M \overset{(\cdot)}{\Rightarrow} t \dashv \Omega' \quad \Omega' \vdash M \overset{(\cdot)}{\Leftarrow} \mathsf{loan}^{m}\, l \dashv \Omega''}{\Omega \vdash \&\mathsf{mut}\ M \overset{(\cdot)}{\Rightarrow} \mathsf{borrow}^{\mathsf{m}}\, l\, t \dashv \Omega''}$

To construct a mutable reference ($\langle$*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $\&\mathsf{mut}\ M\rangle$), you first move the value from M, and replace it via a write with a loan identifier. When the reference is dropped, it uses this loan identifier to find where to write the updated value back to. Because $\langle$*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $\&\mathsf{mut}\ M\rangle$ uses both move and write, it also enforces the location being referenced isn't behind an immutable reference.

Immutable reference construction ($\langle$*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $\&M\rangle$) does the same, but with non-destructive operations which allows it to happen through immutable references.

Loan Restrictions - Comptime reference construction puts a loan restriction into the environment, which forces the reference to have the same type when it is dropped as when it is created. This is used by function body checking:

```
f = (x: &mut 'a) -> 'unit {
  *x = 'b;
  'unit
}; // × : x cannot be dropped because it has the wrong type
```

Function bodies drop everything in their environments (by nature of being statements, not expressions), and **to drop a reference it must be within its loan restriction**, so by type-checking a statement with a loan restriction in the environment, you know it writes the correct type back eventually, even if it temporarily changes the type of the reference locally.

4.5.3 Functions

This section defines abstraction and application.

Def. 4.5.6: Function Interpretations

M	$\Omega \vdash M \overset{(\cdot)}{\Rightarrow} t \dashv \Omega'$	$\Omega \vdash M \dot{\rightsquigarrow} t \dashv \Omega'$
MN	$\dfrac{\begin{array}{ll}\Omega \vdash F \dot{\rightarrow} (M \to S_t) & /\!/ \textit{ function def.}\\ \Omega \vdash A \Rightarrow v \dashv \Omega' & /\!/ \textit{ argument type}\\ \Omega' \vdash T \Leftarrow v \dashv \Omega'' & /\!/ \textit{ inner env.}\\ \Omega'' \vdash S_t \sqsubseteq * \dot{\rightsquigarrow} w & /\!/ \textit{ return type}\\ \Omega' \vdash \mathsf{drop}\, v \dashv \Omega''' & /\!/ \textit{ propogate effects}\end{array}}{\Omega \vdash FA \Rightarrow w \dashv \Omega'''}$	$\dfrac{\begin{array}{ll}\Omega \vdash F \dot{\rightsquigarrow} (T \to S_t) & /\!/ \textit{ function def.}\\ \Omega \vdash A \dot{\rightsquigarrow} v & /\!/ \textit{ argument type}\\ \Omega \vdash T \dot{\leftsquigarrow} v \dashv \Omega' & /\!/ \textit{ inner env.}\\ \Omega' \vdash S_t \sqsubseteq * \dot{\rightsquigarrow} w' & /\!/ \textit{ return type}\end{array}}{\Omega \vdash FA \dot{\rightsquigarrow} w}$
	$\dfrac{\begin{array}{ll}\Omega \vdash F \rightarrow (M \to S) & /\!/ \textit{ function def.}\\ \Omega \vdash A \Rightarrow v \dashv \Omega' & /\!/ \textit{ argument value}\\ \Omega' \vdash M \Leftarrow v \dashv \Omega'' & /\!/ \textit{ inner env.}\\ \Omega'' \vdash S \Rightarrow w \dashv \Omega''' & /\!/ \textit{ return value}\end{array}}{\Omega \vdash FA \Rightarrow w \dashv \Omega'''}$	
M->S_t{ S }	$\dfrac{\begin{array}{ll}\Omega \vdash \mathsf{comptime} \dashv \Gamma & /\!/ \textit{ no scope capture}\\ \Gamma \vdash M \underset{\mathsf{max}}{\Leftarrow} m \dashv \Gamma' & /\!/ \textit{ input type}\\ \Gamma' \vdash S \sqsubseteq S_t \Rightarrow u & /\!/ \textit{ check body}\end{array}}{\Omega \vdash M\text{->}S_t\{S\} \Rightarrow M \to S_t}$	$\dfrac{\begin{array}{ll}\Omega \vdash T \underset{\mathsf{max}}{\dot{\leftsquigarrow}} t \dashv \Omega' & /\!/ \textit{ argument type}\\ \Omega' \vdash S_t \sqsubseteq S'_t \dot{\rightsquigarrow} u & /\!/ \textit{ check body}\end{array}}{\Omega \vdash T\text{->}S_t \dot{\rightsquigarrow} T \to S_t}$
	$\dfrac{/\!/ \textit{ no checking}}{\Omega \vdash M\text{->}S_t\{S\} \Rightarrow M \to S}$	

Concrete vs abstract differences

Apart from ⟨*def.* $\overset{(\cdot)}{\Rightarrow}$, $\overset{(\cdot)}{\Leftarrow}$ *for* $M : T$⟩, abstraction and application are the only constructions that have a different behavior during abstract interpretation and concrete interpretation. **At a function call site: abstract interpretation executes the function's *return type* whereas concrete interpretation executes the function's *body*.** This means the cost of abstract interpretation is linear w.r.t. the number of runtime function bodies, whereas concrete interpretation directly executes your program, and gets its computational complexity from there.

While abstract interpretation will not diverge due to executing runtime terms, it may diverge from executing comptime terms due to comptime terms themselves likely being Turing complete.

For a function M `->` S_t `{` S `}`, abstract interpretation will run to the function return type ($M \to S_t$) whereas abstraction will run to the body ($M \to S$). This is why concrete interpretation runs the body whenever the function is called and abstract interpretation only runs the return type.

Function Checking

The return type must be a sound approximation of the body if the aforementioned difference in behavior between concrete and abstract interpretation is to be sound. This is done via bounded statement interpretation, $\Omega \vdash S \sqsubseteq S_t \Rightarrow u$. If bounded statement interpretation succeeds, then S is a subtype S_t, and it will continue to be in all narrowings of Ω. This property is discussed further in Property 5.2.1 and 5.2.3.

4.5.4 Pairs

Due to being dependent, the interpretations of pairs in Ochre are similar to the interpretations for functions. Accessing the left-hand side of a pair directly gives you the stored value/type. Accessing the right-hand side of a pair uses the type of the left, along with the syntax stored in the abstract environment, to calculate the type of the right-hand side.

Def. 4.5.7: Environment Pair Elimination

t	left t	right t
$(t, T \to S)$	$\dfrac{}{\Omega \vdash \mathsf{left}\,(t, T \to S) = t}$	$\dfrac{\Omega \vdash \mathsf{comptime} \dashv \Gamma \quad \Gamma \vdash T \dot{\rightsquigarrow} t \dashv \Gamma' \quad \Gamma' \vdash S \sqsubseteq \_ \dot{\rightsquigarrow} u}{\Omega \vdash \mathsf{right}\,(t, T \to S) = u}$

Accessing the right element of a pair is so involved because it involves interpreting the term stored in the abstract environment. This must be done with care to avoid soundness issues: the environment being used to execute this term during a pair access is different to the one used to construct the pair; how can we expect it to give the same result? Because we filter the environment for only the comptime variables, and because interpretation only ever narrows comptime variables, we know that the result of executing the syntax at elimination is a subtype of what it was at construction. See Property **??** for more.

Def. 4.5.8: Pair Construction Interpretations

M	$M \overset{(\cdot)}{\Rightarrow} v$	$M \overset{(\cdot)}{\to} v$	$T \dot{\rightsquigarrow} t$
M, N	$\forall \diamond \in \{\overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\to}\}. \left[\dfrac{\Omega \vdash M \diamond m \dashv \Omega' \quad \Omega' \vdash N \diamond n \dashv \Omega''}{\Omega \vdash (M,\ N) \diamond (m, \_ \to n) \dashv \Omega''}\right]$		$\dfrac{\Omega \vdash (T,\ \_\ \texttt{->}\ S) \dot{\rightsquigarrow} v}{\Omega \vdash (T,\ S) \dot{\rightsquigarrow} v}$ $\dfrac{\Omega \vdash T \dot{\rightsquigarrow} t \quad \Omega \vdash \mathsf{right}\,(t, T' \to S) = u}{\Omega \vdash (T,\ T'\ \texttt{->}\ S) \dot{\rightsquigarrow} (t, T' \to S)}$

Pairs are a combination of a left type, and a way to turn a left type into the right type. Pairs constructors interpreted with a runtime modality *cannot* be dependent immediately; but

```
x = ('a, 'a); // {x ↦ ('a,'a)}
x = x: ('a | 'b, L -> L); // {x ↦ ({'a,'b}, L → L)}
```

Listing 17: When x is constructed, it is non-dependent. When the type annotation is applied, it becomes dependent.

you can make them dependent after constructing them by widening their type with a type annotation, like so:

This shows itself in the difference between the comptime and runtime typing rules: In ⟨*def.* $\overset{(\cdot)}{\Rightarrow}$, $\overset{(\cdot)}{\to}$ *for* M, N⟩, the constructed pair $((m, \_ \to n))$ is always non-dependent. Whereas in ⟨*def.* $\dot{\rightsquigarrow}$ *for* M, N⟩ the constructed pair can be dependent $((t, T' \to S))$. This is done because when calculating the right value of a pair, the term used to construct the right is executed, and since that execution is happening at compile time, it must be a comptime term.

Within the comptime modality the left value of a pair can be bound in the definition of the right with an optional T `->` prefix, as used in Figure 17. As shown in ⟨*def.* $\dot{\rightsquigarrow}$ *for* M, N⟩ this is put in the environment for later re-use.

Def. 4.5.9: Pair Elimination Interpretations

M	$M \overset{(\cdot)}{\Rightarrow} v$	$M \overset{(\cdot)}{\to} v$	$T \dot{\rightsquigarrow} t$
$M.0$	$\dfrac{\begin{array}{c}\Omega \vdash M \overset{(\cdot)}{\Rightarrow} t \dashv \Omega' \\ \Omega \vdash \mathsf{left}\, t = t_0 \\ \Omega \vdash \mathsf{right}\, t = t_1 \\ \Omega' \vdash M \overset{(\cdot)}{\Leftarrow} (\top, \_ \to t_1) \dashv \Omega''\end{array}}{\Omega \vdash M.0 \overset{(\cdot)}{\Rightarrow} t_0 \dashv \Omega''}$	$\forall \diamond \in \{\overset{(\cdot)}{\to}, \dot{\rightsquigarrow}\}. \left[\dfrac{\Omega \vdash M \diamond t \quad \Omega \vdash \mathsf{left}\, t = t_0}{\Omega \vdash M.0 \diamond t_0}\right]$	
$M.1$	$\dfrac{\begin{array}{c}\Omega \vdash M \overset{(\cdot)}{\Rightarrow} t \dashv \Omega' \\ \Omega' \vdash \mathsf{left}\, t = t_0 \\ \Omega' \vdash \mathsf{right}\, t = t_1 \\ \Omega' \vdash M \overset{(\cdot)}{\Leftarrow} (t_0, \_ \to \top) \dashv \Omega''\end{array}}{\Omega \vdash M.1 \overset{(\cdot)}{\Rightarrow} t_1 \dashv \Omega''}$	$\forall \diamond \in \{\overset{(\cdot)}{\to}, \dot{\rightsquigarrow}\}. \left[\dfrac{\Omega \vdash M \diamond t \quad \Omega' \vdash \mathsf{right}\, t = t_1}{\Omega \vdash M.1 \diamond t_1}\right]$	

Pair elimination rules heavily leverage the left and right helper functions. Moving either the left or right element of a pair away from the pair breaks the dependence, as represented by both ⟨*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $M.0$⟩ and ⟨*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $M.1$⟩ replacing the pair with a non-dependent pair $((\top, \_ \to t_1)$ and $(t_0, \_ \to \top)$ respectively).

Def. 4.5.10: Pair Write Interpretations

M	$M \overset{(\cdot)}{\Leftarrow} v$	$M \overset{(\cdot)}{\leftarrow} v$	$M \dot{\leftsquigarrow} v$
M, N		$\forall \diamond \in \{ \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \dot{\leftsquigarrow} \}.$ $\left[\dfrac{\begin{array}{c}\Omega \vdash \mathsf{left}\, t = t_0 \\ \Omega \vdash \mathsf{right}\, t = t_1 \\ \Omega \vdash M \diamond t_0 \dashv \Omega'' \\ \Omega'' \vdash N \diamond t_1 \dashv \Omega'''\end{array}}{\Omega \vdash M, N \diamond t \dashv \Omega'''}\right]$	
$M.0$	$\dfrac{\begin{array}{c}\Omega \vdash M \overset{(\cdot)}{\Rightarrow} t \dashv \Omega' \\ \Omega' \vdash \mathsf{left}\, t = \top \\ \Omega' \vdash \mathsf{right}\, t = t_1 \\ \Omega' \vdash M \overset{(\cdot)}{\Leftarrow} (t_0', \_ \to t_1) \dashv \Omega''\end{array}}{\Omega \vdash M.0 \overset{(\cdot)}{\Leftarrow} t_0' \dashv \Omega''}$	$\forall \diamond \in \{ \overset{(\cdot)}{\leftarrow}, \dot{\leftsquigarrow} \}.$ $\left[\dfrac{\begin{array}{c}\Omega \vdash M(\mathsf{flip} \diamond)(t_0, T \to S) \\ \Omega \vdash t_0' \sqsubseteq t_0 \\ \Omega \vdash M \diamond (t_0', T \to S) \dashv \Omega'\end{array}}{\Omega \vdash M.0 \diamond m \dashv \Omega''}\right]$	
$M.1$	$\dfrac{\begin{array}{c}\Omega \vdash M \overset{(\cdot)}{\Rightarrow} t \dashv \Omega' \\ \Omega' \vdash \mathsf{left}\, t = t_0 \\ \Omega' \vdash \mathsf{right}\, t = t_1 \\ \Omega' \vdash M \overset{(\cdot)}{\Leftarrow} (t_0, \_ \to t_1') \dashv \Omega''\end{array}}{\Omega \vdash M.1 \overset{(\cdot)}{\Leftarrow} t_1' \dashv \Omega''}$		

Destructuring is done via $\langle$*def.* $\overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \dot{\leftsquigarrow}$ *for* $M, N\rangle$. Writing a pair to a pair constructor breaks the pair into its left and right element, then writes each element to the respective terms in the pair constructor.

Writing to a pair breaks the pair into left and right, then writes it back with a new left or right element. In the process, the dependence of the pair is broken.

Narrowing the left of a pair does *not* break the dependence, because the new value is a subtype of the old, so you know the term stored for the right-hand side will still work.

Narrowing the right of a pair has not been defined because it is never used. There is no technical reason why it could not be interpreted; although asserting that a new term is a subtype could be difficult if you wanted to keep the dependence, and would probably involve bounded statement interpretation.

4.5.5 Type Constructs

Def. 4.5.11: Type Annotation Interpretations

M	$M \dot{\rightsquigarrow} t$ $M \overset{(\cdot)}{\rightarrow} t$	$M \overset{(\cdot)}{\Rightarrow} t$	$M \overset{(\cdot)}{\Leftarrow} t$	$M \overset{(\cdot)}{\leftarrow} t$	$M \dot{\leftsquigarrow} t$
M:T	$\forall \diamond \in \{\dot{\rightsquigarrow}, \rightarrow, \Rightarrow\}. \left[\dfrac{\Omega \vdash M \diamond m \dashv \Omega' \quad \Omega' \vdash T \dot{\rightsquigarrow} t \quad \Omega' \vdash m \sqsubseteq t}{\Omega \vdash M : T \diamond t \dashv \Omega'}\right]$			$\forall \diamond \in \{\dot{\leftsquigarrow}, \leftarrow, \Leftarrow\}. \left[\dfrac{\Omega \vdash M \diamond m \dashv \Omega' \quad \Omega' \vdash T \dot{\rightsquigarrow} t \quad \Omega' \vdash m \sqsubseteq t}{\Omega \vdash M : T \diamond m \dashv \Omega'}\right]$	
	$\forall \diamond \in \{\rightarrow, \Rightarrow, \Leftarrow, \leftarrow\}. \left[\dfrac{\Omega \vdash M \diamond m \dashv \Omega' \quad /\!/ \textit{ ignores type annotations}}{\Omega \vdash M : T \diamond m \dashv \Omega'}\right]$				

Apart from ⟨*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $M\,N, M$ ->S⟩, type annotations are the only constructions that have a different behavior during abstract interpretation and concrete interpretation. **When interpreting a type annotation: abstract interpretation interprets both the left and right of the** :**, then asserts the left is a subtype of the right whereas concrete interpretation ignores the right and acts on the left**. When read-interpreting a type annotation, information is lost: you only get the type of the annotation, not the term being typed.

Def. 4.5.12: Type Constructor Interpretations

T	$T \dot{\rightsquigarrow} t$
$T \mid U$	$\dfrac{\Omega \vdash T \dot{\rightsquigarrow} t \dashv \Omega' \quad \Omega' \vdash U \dot{\rightsquigarrow} u \dashv \Omega'' \quad \Omega'' \vdash t \sqcup u = t'}{\Omega \vdash T \mid U \dot{\rightsquigarrow} t' \dashv \Omega''}$
v	$\dfrac{}{\Omega \vdash v \dot{\rightsquigarrow} v}$

The type union operator | interprets its arguments, then returns the union of their types. The type union operator is the only place in the expression interpretations where type union occurs (⟨*def.* $\overset{(\cdot)}{\Rightarrow}$, $\dot{\rightsquigarrow}$ *for* match⟩ also does type union), and it is only defined for the comptime modality. This is how we guarantee that non-singleton types are only ever introduced via comptime interpretations, and therefore that runtime interpretations if they choose to read the left of type annotations instead of the right.

⟨*def.* $\dot{\rightsquigarrow}$ *for* v⟩ is a trick so we can write already interpreted values to the right of a pair, which is useful for breaking dependencies as is done in ⟨*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $M\,.\,0$⟩. Having to define this interpretation is a sign of a mistake in the design, I think this mistake is storing syntax in the abstract environment instead of isolating it more behind some other construct. I have chosen to prioritize other work over this.

4.5.6 Statements

Statements are unique because in abstract interpretation they do not return a modified environment, they consume the environment. This is useful because in abstract interpretation you cannot soundly reason about the state of the environment after a match, at least not without supporting dependence between variables.

In concrete interpretation, only a single match statement is executed (as guaranteed by the match arms being disjoint), so it is sound to return an environment afterward. Concrete interpretation uses this to allow the side effects of a statement to depend on the input environment.

Def. 4.5.13: Expression Statement Interpretations

S	$\Omega \vdash S \sqsubseteq S_t \overset{(\cdot)}{\Rrightarrow} v$	$\Omega \vdash S \dot{\rightsquigarrow} t$
M	$\dfrac{\begin{array}{c}\Omega \vdash M \overset{(\cdot)}{\Rrightarrow} v \dashv \Omega' \\ \Omega' \vdash \mathsf{drop} \\ \Omega \vdash S_t \sqsubseteq * \dot{\rightsquigarrow} t \\ \Omega \vdash v \sqsubseteq t\end{array}}{\Omega \vdash M \sqsubseteq S_t \overset{(\cdot)}{\Rrightarrow} w}$	$\dfrac{\begin{array}{c}\Omega \vdash T \dot{\rightsquigarrow} t \\ \Omega \vdash S_t \sqsubseteq * \dot{\rightsquigarrow} t_s \\ \Omega \vdash t \sqsubseteq t_s\end{array}}{\Omega \vdash T \sqsubseteq S_t \dot{\rightsquigarrow} t}$

All expressions are themselves statements, when this is the case, the expression is executed and checked against the *bound*, which is usually represented by S_t. After the expression is executed, the leftover environment is dropped, which will drop all leftover references and assert they conform to their loan restriction, see ⟨*def.* drop*for* borrow⟩ for discussion around loan restrictions.

Def. 4.5.14: Assingment Interpretations

S	$\Omega \vdash S \sqsubseteq S_t \overset{(\cdot)}{\Rrightarrow} v$	$\Omega \vdash S \dot{\rightsquigarrow} t$
$M{=}N\,;S$	$\forall \diamond \in \{ \dot{\rightsquigarrow}, \overset{(\cdot)}{\Rrightarrow} \}.$ $\left[\begin{array}{c} /\!/ \textit{ peform assignment} \\ \Omega \vdash N \diamond v \dashv \Omega' \\ \Omega' \vdash M\ (\mathsf{flip}\diamond)\ v \dashv \Omega'' \\ /\!/ \textit{ assert: compatible LHS syntax} \\ \mathsf{runtime}\ M \quad \text{if concrete}\ \diamond \\ \mathsf{comptime}\ M \quad \text{if squiggily}\ \diamond \\ /\!/ \textit{ assert: bound can only tighten} \\ \Omega \vdash S_t \dot{\rightsquigarrow} t \quad \text{if abstract}\ \diamond \\ \Omega'' \vdash S_t \dot{\rightsquigarrow} t' \quad \text{if abstract}\ \diamond \\ \Omega'' \vdash t' \sqsubseteq t \quad \text{if abstract}\ \diamond \\ /\!/ \textit{ execute remaining computation} \\ \Omega'' \vdash S \sqsubseteq S_t \overset{(\cdot)}{\Rrightarrow} w \\ \hline \Omega \vdash M{=}N\,;S \sqsubseteq S_t \overset{(\cdot)}{\Rrightarrow} w \end{array}\right]$	$\dfrac{\begin{array}{c}\Omega \vdash N \dot{\rightsquigarrow} v \dashv \Omega' \\ \Omega' \vdash M \dot{\leftsquigarrow} v \dashv \Omega'' \\ \Omega'' \vdash S_t \dot{\rightsquigarrow} w\end{array}}{\Omega \vdash M{=}N\,;S_t \dot{\rightsquigarrow} w}$

Assignment read-interprets the right-hand side of the =, then write-interprets the result to the left.

Assignment has to be a statement-level construct because it can narrow the environment. Expressions cannot narrow the environment because that would break Property 5.2.5. This is why we must assert the assignment has only tightened our upper bound, because that would defeat the point of an upper bound and break Property 5.2.1.

Def. 4.5.15: Match Statement Interpretations

S	$\Omega \vdash S \sqsubseteq S_t \overset{(\cdot)}{\Rightarrow} v$	$\Omega \vdash S \dot{\rightsquigarrow} t$

`match` M `{`
M'_0 `=>` S_0 `,`
$\vdots$
M'_k `=>` S_k `,`
`}`
	$\forall \diamond \in \{ \Rightarrow, \dot{\rightsquigarrow} \}.$ $\left[\begin{array}{rll} & \Omega \vdash M \diamond m \dashv \Omega' & /\!/ \textit{ eval scruitinee} \\ \forall i.[& \Omega' \vdash M_i \overset{(\cdot)}{\underset{\max}{\Leftarrow}} m_i \dashv \Omega'_i & /\!/ \textit{ branch input} \quad] \\ & m = \biguplus_i [m_i] & /\!/ \textit{ assert branches disjoint} \\ \forall i.[& \Omega'_i \vdash S_i : S_t \diamond n_i & /\!/ \textit{ branch output} \quad] \\ & n = \bigsqcup_i [n_i] & /\!/ \textit{ combine branch outputs} \\ \hline & \Omega \vdash \texttt{match}\, M\, \{ \overrightarrow{M' \texttt{ => } S} \} \sqsubseteq S_t \diamond n & \end{array} \right]$	

$$\frac{\begin{array}{rl} \Omega \vdash M \diamond m_i \dashv \Omega' & /\!/ \textit{ eval scruitinee} \\ \Omega' \vdash M'_i \overset{(\cdot)}{\Leftarrow} m_i \dashv \Omega'' & /\!/ \textit{ branch input} \\ \Omega'' \vdash S_i \diamond v \dashv \Omega''' & /\!/ \textit{ branch output} \end{array}}{\Omega \vdash \texttt{match}\, M\, \{ \overrightarrow{M' \texttt{ => } S} \} \diamond v \dashv \Omega'''}$$

Concrete match statement interpretation interprets the scrutinee and then calculates the return value from one of the outputs. By itself, this would make concrete interpretation non-deterministic because it can match any branch, but in our abstract interpretation of match statements, we assert that the branch domains are disjoint, so if we also have an abstract interpretation derivation, we know the concrete interpretation is deterministic.

Abstract match statement interpretation first evaluates the scrutinee, then it evaluates all of the branches. The result of interpreting a match statement is the union of the types of each of the branches, which is precise (Property 5.2.5).

4.5.7 Max Interpretation

Max interpretation returns the maximum value that can be written to a given term. It is used in function definition checking to check the type of the argument (see ⟨*def.* $\Rightarrow$, $\dot{\rightsquigarrow}$ *for* M->S_t⟩).

Def. 4.5.16: Max Interpretation

M	$M \underset{\max}{\Leftarrow} t$	$T \underset{\max}{\dot{\rightsquigarrow}}$
$'a$	$\forall \diamond \in \{ \underset{\max}{\Leftarrow}, \underset{\max}{\dot{\rightsquigarrow}} \}. \left[\frac{}{\Omega \vdash {'a} \diamond {'a}} \right]$	
$\_$	$\forall \diamond \in \{ \underset{\max}{\Leftarrow}, \underset{\max}{\dot{\rightsquigarrow}} \}. \left[\frac{}{\Omega \vdash \_ \diamond \top} \right]$	
x	$\forall \diamond \in \{ \underset{\max}{\Leftarrow}, \underset{\max}{\dot{\rightsquigarrow}} \}. \left[\frac{}{\Omega \vdash x \diamond \top} \right]$	
X		$\frac{}{\Omega \vdash X \diamond \top}$
M, N	$\forall \diamond \in \{ \underset{\max}{\Leftarrow}, \underset{\max}{\dot{\rightsquigarrow}} \}. \left[\frac{\Omega \vdash M \diamond m \dashv \Omega' \quad \Omega' \vdash N \diamond n \dashv \Omega''}{\Omega \vdash M, N \diamond (m, \_ \to n) \vdash \Omega''} \right]$	
$M : T$	$\forall \diamond \in \{ \Leftarrow, \dot{\rightsquigarrow} \}. \left[\frac{\Omega \vdash T \dot{\rightsquigarrow} t \dashv \Omega' \quad \Omega' \vdash M \diamond t \dashv \Omega''}{\Omega \vdash M : T \underset{\max}{\diamond} t \vdash \Omega''} \right]$	

The maximum value one can write to a type annotation is the annotated type, because of the subtype restriction on $\langle$*def.* $\Leftarrow, \dot{\rightsquigarrow}$ *for* $M : T\rangle$.

4.6 Design Decisions

4.6.1 Justification of Feature Inclusion

List of features, along with why their inclusion was important.

Type Erasure

A crucial goal of this project is to generate efficient machine code, so I don't want any aspect of the type system to influence runtime. It also ensures all reasoning about the program's correctness is done at compile time.

Implementability

The type system presented as a means to the end of making a production-ready language with sound foundations. If it relies too heavily on non-syntax-driven typing rules or extra information provided during the derivation, implementation could be rendered infeasible. An example of this is the $:$ operator, which asserts that the LHS has the type of the RHS; if this was just theory work I wouldn't need this because I could make these type assertions in the derivations.

Manual memory management

Manual memory management is important both toward the end of making efficient machine code, and dependent types. The real core of why dependent types are possible in this context is because safe Rust behaves very similarly to pure functional code behind the scenes, as demonstrated by the existence of multiple projects that can translate safe Rust into pure functional code [Ho and Protzenko, 2022][Ullrich, 2024]. The abstract interpretation introduced by Aeneas to track the state of ownership has proven crucial to detecting when typing judgments are invalidated by mutations.

4.6.2 Justification of Feature Omission

List of omitted features, along with why their omission is inconsequential for the conclusions of this research:

Returning mutable references

In Ochre you can put references in variables and pass them to functions, but you can never return them from a function. This doesn't restrict which programs you can express, because you can inline any function that would return a mutable reference and it will work, however, it does make using custom data structures like containers extremely cumbersome because you cannot define generic getters that return references to elements within the container. Supporting returning mutable references would involve introducing the concept of regions from Aeneas into Ochre, which I'm almost certain is possible, but would have complicated the already complicated type system.

Reasoning about function side effects/strong updates

In Ochre, if a function takes a mutable reference to a value of type T, the value is guaranteed to still be of type T after the function return. You may want this not to be the case if the type encodes some property of your data structure, for instance, if you have a type for lists and

another for sorted lists you may want an in-place sorting algorithm to change the type of the referenced list into a sorted list. I choose to not support this for a few reasons:

1. I predict that it will be idiomatic in Ochre to separate data structures from proofs about their structure. If this is the case, you could return a proof about one of your inputs, which immutably borrows that input, causing it to be invalidated if the data structure is ever mutated. This would not involve strong updates.
2. It would complicate the type system and syntax further.
3. People can still do strong updates by moving the data structure in and out of a function instead of giving it a borrow. This is even possible if the caller only has a mutable reference to the data because strong updates are allowed locally.

Unboxed types

All values in Ochre are one machine word long, which involves pairs being boxed. Unboxing data would require me to reason about the size of types at compile time, which would have complicated the type system further and detracted from the core contributions. Unboxing pairs should be very possible for Ochre in the future because it already has ownership and it will do generics via monomorphisation like Rust and C++. The complexity will arise because, unlike Rust, the type of data can change due to a mutation, and therefore its size. I will get around this via explicit boxing: a pointer to a heap allocation is always one machine word long, so you can change the size of the data behind it without changing the size of the data structure the pointer lies within. Unboxed types could be introduced in the future via the method laid out in Appendix A.2.

Primitive data types

As presented, Ochre doesn't expose key data types such as machine integers which can be used to generate efficient arithmetic. This is a major problem for its short-term usefulness because all numeric arithmetic must be done with inefficient algorithms over heap-allocated Peano numbers. I think this is a reasonable omission because this work is mostly a proof of concept, and efficiently type-checking and compiling these primitives is well-explored and will be introduced into Ochre in the future.

Chapter 5

Analysis

This chapter analyses whether the abstract interpretation, as defined in Chapter 4, is capable of accepting correct programs and rejecting incorrect ones, which is the goal of this research.

Section 5.1 answers this by using the abstract interpretation manually on specific programs, and making sure it rejects incorrect programs and accepts correct ones.

Section 5.2 answers this by reasoning more generally about the properties held by the abstract interpretations, including soundness.

5.1 Individual Programs

This section starts by type-checking simple programs that other languages can already type-check to explore the basics of the abstract interpretation, then gradually introduces features which other languages cannot check.

The goal of this section is to convince the reader that the abstract interpretation as defined does work for interesting programs and to provide the reader with reference when they are curious about how a particular detail plays out in a real program.

Notation

To allow for easier navigation and layout around large derivations, this section uses a non-standard notation for derivation trees. First, the derivations are displayed upside-down, with the conclusions at the top. Second, all premises are stacked vertically underneath, instead of across the page. Thirdly, all premises are indented to the right, so you can tell them apart from sub-premises. This notation is shown in Figure 5.1.

This allows each line in the derivation to be much longer, which allows the right of the

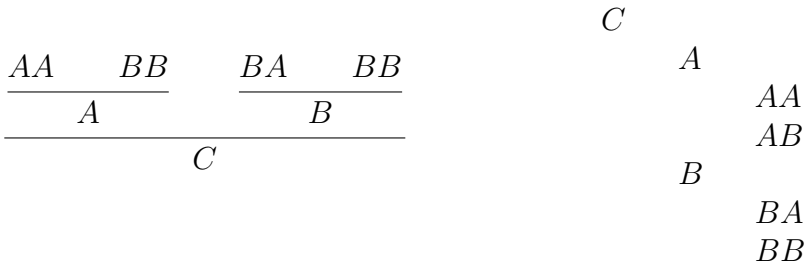


Figure 5.1: On the left: standard typing notation. On the right: the non-standard notation used in this section.

```
1                                  // Ω0 = ∅
2   pair = ('world, 'hello);       // Ω1 = ∅, pair ↦ (’world, _ → ’hello)
3   temp = pair.0;                 // Ω2 = ∅, pair ↦ (⊤, _ → ’hello), temp ↦ ’world
4   pair.0 = pair.1;               // Ω3 = ∅, pair ↦ (’hello, _ → ⊤), temp ↦ ’world
5   pair.1 = temp;                 // Ω4 = ∅, pair ↦ (’hello, _ → ’world), temp ↦ ⊤
6
7   pair // (’hello, _ → ’world)
```

Listing 18: Hello world program. The state of the environment after each line is shown in the comments.

page to be used to declare variables for use in the derivation, such as commonly used types and environments.

5.1.1 Hello World

Listing 18 defines an unconventional hello world program which starts by putting the atoms ’hello and ’world in a pair the wrong way around, then swaps them via a third variable.

This demonstrates how the abstract interpretation keeps track of which values have been moved and which haven’t in the abstract environment and acts as a good demonstration of the inference rules that make up the abstract interpretation.

An alternative version of Listing 18 could swap the values in the pair over with `(pair.0, pair.1) =` or simply `pair = (pair.1, pair.0)`. Due to not having unboxed pairs, this would cause an extra allocation.

We define meta-level shortcuts for each of the lines:

```
1                                  // Ω₀ = ∅
2   Same = ('a | 'b, L -> L);      // Ω₁ = ∅, Same ↦ ({'a, 'b}, L → L)
3
4   overwrite = (p: &mut Same) -> 'unit {
5                  // Ω₁₀ = Ω₁, p ↦ borrow^m l ({'a, 'b}, L → L), l ↦ ({'a, 'b}, L → L)
6     (*p).0 = 'a; // Ω₁₁ = Ω₁, p ↦ borrow^m l ('a, _ → {'a, 'b}), l ↦ ({'a, 'b}, L → L)
7     (*p).1 = 'a; // Ω₁₂ = Ω₁, p ↦ borrow^m l ('a, _ → 'a), l ↦ ({'a, 'b}, L → L)
8     'unit        // Ω₁₂ ⊢ drop
9   }                              // Ω₂ = Ω₁, overwrite ↦ (p: &mut Same) → 'unit
10
11  pair = ('b, 'b);               // Ω₃ = Ω₂, pair ↦ ('b, _ → 'b)
12  overwrite(&mut pair);          // Ω₄ = Ω₂, pair ↦ ({'a, 'b}, L → L)
13
14  pair // ({'a, 'b}, L → L)
```

Listing 19: A program which mutates a dependent pair correctly.

$$
\begin{aligned}
L_2 &= \texttt{pair = ('world, 'hello)} \\
L_3 &= \texttt{temp = pair.0} \\
L_4 &= \texttt{pair.0 = pair.1} \\
L_5 &= \texttt{pair.1 = temp} \\
L_7 &= \texttt{pair}
\end{aligned}
$$

The program is shown to have type ('hello, _ → 'world) if we can find a derivation of the following form:

$$\emptyset \vdash L_2; L_3; L_4; L_5; L_7 \sqsubseteq * \Rightarrow (\text{'hello}, \_ \to \text{'world})$$

A full derivation of this form is given in Appendix A.3.1.

5.1.2 Mutating Dependent Pairs

This section shows how mutation interacts with dependent types through an example where mutation is deemed correct or not based on a dependent type.

Listing 19 defines a dependent pair type, where the right element must be equal to the left element. There are only two inhabitants of this type: ('a, 'a) and ('b, 'b). A function is defined, `overwrite`, which writes 'a to both the left and the right of a pair of this type, via a mutable reference.

`overwrite` temporarily leaves the pair in an invalid state between lines 6 & 7 (('a, 'b)), but by the end of the function call the pair is left in a valid state, so the program is correct.

$$
\begin{array}{l}
\Omega_1 \vdash p_3 \sqsubseteq p_0 \\
\quad \Omega_1 \vdash \text{'a} \sqsubseteq \{\text{'a}, \text{'b}\} \\
\qquad \{\text{'a}\} \subseteq \{\text{'a}, \text{'b}\} \\
\quad \Omega_1 \vdash \mathsf{comptime} \dashv \Gamma_2 \qquad \Gamma_2 = \Omega_1, \mathtt{L} \mapsto \top \\
\quad \Gamma_2 \vdash \mathtt{L} \dot{\leftsquigarrow} \text{'a} \dashv \Gamma_2' \\
\qquad \Gamma_2' = \Gamma_2 \left[\dfrac{\mathtt{L} \mapsto \text{'a}}{\mathtt{L} \mapsto \top} \right] \\
\quad \Gamma_2 \vdash \_ \dot{\rightsquigarrow} \text{'a} \\
\quad \Gamma_2' \vdash \text{'a} \sqsubseteq \mathtt{L} \dot{\rightsquigarrow} \text{'a} \\
\qquad \Gamma_2' \vdash \text{'a} \dot{\rightsquigarrow} \text{'a} \\
\qquad \Gamma_2' \vdash L \dot{\rightsquigarrow} \text{'a} \\
\qquad\quad \mathtt{L} \mapsto \text{'a} \in \Gamma_2' \\
\qquad \Gamma_2' \vdash \text{'a} \sqsubseteq \text{'a} \\
\qquad\quad \{\text{'a}\} \subseteq \{\text{'a}\}
\end{array}
$$

Figure 5.2: Derivation which checks the post-mutation pair against the `overwrite` function's type signature.

Notice, enforcing AXM is crucial to being able to temporarily invalidate the pair: without AXM, another reference could exist to this pair, and be dereferenced while the pair is in an invalid state.

The caller (line 12) cannot tell from the type signature of `overwrite` what exact value it will set the pair to, it only knows it will be of type `Same`. This means after the call to `overwrite`, as far as the caller is concerned, `pair` could be ('b, 'b), which is why `pair` $\mapsto$ ({'a, 'b}, $L \to L$) in the environment instead of the more precise `pair` $\mapsto$ ('a, _ $\to$ 'a).

The mutation is checked to be correct at the end of the function when the environment is cleaned up. Dropping the reference causes the value in the borrow ('a, _ $\to$ 'a) to be checked against its loan restriction, which is ({'a, 'b}, L $\to$ L), as shown in full by Figure 5.2.

Appendix A.3.2 shows a full derivation of the `overwrite` function.

Rejecting Incorrect Mutation

The function body may temporarily change the type of any references to any type, but by the end of the function body, they must all be of the correct type. Listing 20 mutates the pair it is given reference to incorrectly: while 'b is a valid value for the right-hand side of the pair, it can only be 'b when the left is also 'b, which in this case, it is not.

The derivation for this incorrect program is almost identical to the derivation for its correct counterpart, apart from p_3 = ('a, _ $\to$ 'b) instead of p_3 = ('a, _ $\to$ 'a). The attempted derivation for the final type check is shown in Figure 5.3, but it does not work ultimately because {'b} $\not\subseteq$ {'a}.

```
1    incorrect_overwrite = (p: &mut Same) -> 'unit {
2                // Ω20 = Ω1, p ↦ borrow^m l ({'a, 'b}, L → L), l ↦ ({'a, 'b}, L → L)
3      p.0 = 'a; // Ω21 = Ω1, p ↦ borrow^m l ('a, _ → {'a, 'b}), l ↦ ({'a, 'b}, L → L)
4      p.1 = 'b; // Ω21 = Ω1, p ↦ borrow^m l ('a, _ → 'b), l ↦ ({'a, 'b}, L → L)
5                // ×, cannot drop p, loan restriction mismatch
6    }
```

Listing 20: An incorrect program, which does not leave the pair in a valid state

$$
\begin{array}{ll}
\Omega_1 \vdash p_3 \sqsubseteq p_0 & p_0 = (\{\text{'a}, \text{'b}\}, L \to L) \\
\quad \Omega_1 \vdash \text{'a} \sqsubseteq \{\text{'a}, \text{'b}\} & p_3 = (\text{'a}, \_ \to \text{'b}) \\
\qquad \{\text{'a}\} \subseteq \{\text{'a}, \text{'b}\} & \\
\quad \Omega_1 \vdash \text{comptime} \dashv \Gamma_2 & \Gamma_2 = \Omega_1, \mathtt{L} \mapsto \top \\
\quad \Gamma_2 \vdash \mathtt{L} \dot{\leftsquigarrow} \text{'a} \dashv \Gamma_2' & \\
\qquad \Gamma_2' = \Gamma_2 \left[\dfrac{\mathtt{L} \mapsto \text{'a}}{\mathtt{L} \mapsto \top} \right] & \\
\quad \Gamma_2 \vdash \_ \dot{\leftsquigarrow} \text{'a} & \\
\quad \Gamma_2' \vdash \text{'b} \sqsubseteq \mathtt{L} \dot{\rightsquigarrow} \text{'a} & \\
\qquad \Gamma_2' \vdash \text{'b} \dot{\rightsquigarrow} \text{'b} & \\
\qquad \Gamma_2' \vdash L \dot{\rightsquigarrow} \text{'a} & \\
\qquad\quad \mathtt{L} \mapsto \text{'a} \in \Gamma_2' & \\
\qquad \Gamma_2' \vdash \text{'b} \sqsubseteq \text{'a} & \\
\qquad\quad \{\text{'b}\} \subseteq \{\text{'a}\} & \text{// } \times\text{, type mismatch: } p \text{ is not of type } (\{\text{'a}, \text{'b}\}, L \to L)
\end{array}
$$

Figure 5.3: Derivation which checks the post-mutation pair for the `incorrect_overwrite` function body

```
1  Swap = T -> (x: &mut T, y: &mut T) -> 'unit {
2    (*x, *y)  = (*y, *x)
3  }
```

Listing 21: Polymorphic Swap

5.1.3 Polymorphic Swap Function

Listing 21 shows a function that takes two references, and swaps the values referenced by them. `Swap` is a comptime function which takes a type `T` as its only argument, and returns a runtime function. This is how generics are done in Ochre, which is conceptually similar to Rust's monomorphisation: a separate function is generated for every combination of input arguments. The runtime function it returns takes the two mutable references and swaps their value.

`Swap` is an example of a function that takes advantage of Aeneas' more precise method of borrow checking: Rust cannot type check Listing 21, despite it being correct. This is because Rust does not allow moving a value from behind a mutable reference, even if you put a valid value back into it by the end of the function call. Any mutable reference in Rust must at all points be valid and pointing to a constant type. This is a very nice consequence of using Aeneas as the borrow checker instead of something more approximate. There are ongoing efforts to make a new borrow checker for Rust which is more precise [Pol].

annotate environment within `Swap`

5.1.4 Peano Numbers and Add

Sections 5.1.1 and 5.1.2 analyze simple programs thoroughly. This section analyses complex programs but instead of showing full typing derivations, it only shows the abstract environments after every program line.

We do this so we can cover more features in a single program, to uncover more edge cases.

Every listing in this section leads on from the last: a variable defined in a listing is available in all subsequent listings.

Listing 23 defines Peano natural numbers and addition on them. Peano numbers use the typical ADT encoding: a Peano number is a pair where the left determines whether it is zero,

```
Nat = ('zero, 'unit) | ('succ, Nat);

add: (x: Nat, x: Nat) -> Nat = (x: Nat, y: Nat) -> Nat {
  match x.0 {
    'zero => y,
    'succ => ('succ, add(x.1, y)),
  }
};
```

Listing 22: Definition of `Nat` and `add`.

```
add: (x: Nat, x: Nat) -> Nat = (x: Nat, y: Nat) -> Nat {
  match x.0 {
    'zero => y,
    'succ => (x.1 = add(x.1, y); x),
  }
};
```

Listing 23: More efficient definition of `add`.

or the successor of another number. If it is the successor of another number, the right of the pair stores that number.

Annoyingly, `add` must be given an explicit type annotation on the left of the assignment as well as the right its type needs to be added to the environment before it is evaluated, so the recursive call can be type-checked.

As defined, addition causes $O(n)$ memory allocations: for each iteration, a new successor node is allocated, which is wasteful. Instead, we can re-use the allocation we already have for `x`, which we do not need once we have already read `x`, as shown in Listing **??**.

You cannot do this in languages like Haskell because you cannot guarantee you have unique access to the allocation, so a new node is always allocated instead. Substantial efforts have been made to optimize re-use in scenarios like this [Ningning et al., 2021], but until it is solved in the general case, systems languages will have to give the programmer enough control to perform optimizations like the above themselves. Ochre, like Rust, can give the programmer this control safely by tracking ownership.

annotate add functions with environments

5.2 Properties and Proofs

This section discusses the various properties of the presented abstract interpretation should have/do have/do not have.

Statements stacked vertically denote conjunction: "$\begin{matrix}A\\B\end{matrix}$" means "$A$ and B".

5.2.1 Soundness

Ochre programs are a statement S. Type-checking and runtime execution both start with empty environments. Therefore, the following is our central soundness property:

Prop. 5.2.1: Statement Interpretation Soundness ($\emptyset$)

$$\begin{matrix}\emptyset \vdash S \sqsubseteq \_ \Rightarrow t\\ \emptyset \vdash S \sqsubseteq \_ \Rightarrow v\end{matrix} \text{ implies } \emptyset \vdash v : t$$

Figure 5.4: Ochre's central soundness property

Which reads "If the abstract interpretation runs to t, and the concrete interpretation runs to v, then $v : t$". It also reflects the fact that program execution always starts with a statement in an empty environment.

Every property in this document after this point is a requirement for proving this soundness property.

Property **??** assumes both $\Rightarrow$ and $\Rightarrow$ instead of only assuming the former and concluding the latter because that would equate to "if it type-checks, it executes" which is undecidable for Turing complete languages [Turing, 1937], which Ochre almost certainly is.

The proof is done by induction on the $\Rightarrow$ derivation and the $\Rightarrow$ derivation simultaneously. So that a stronger inductive hypothesis can be assumed, we prove this stronger property:

Prop. 5.2.2: Statement interpretation Soundness

$$\begin{matrix}\Omega \vdash S \sqsubseteq \_ \Rightarrow t\\ \Delta \vdash S \sqsubseteq \_ \Rightarrow v\\ \Delta : \Omega\end{matrix} \text{ implies } \Omega \vdash v : t$$

Abstract and concrete interpretation only differ in three language constructs: function application $\langle$*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $MN\rangle$, function definition $\langle$*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $M \to S_t\{S\}\rangle$, and type annotation $\langle$*def.* $\overset{(\cdot)}{\Rightarrow}$, $\overset{(\cdot)}{\Leftarrow}$ *for* $M : T\rangle$. The proof is more simple in the cases where the abstract and concrete interpretations are similar, so in order to more efficiently detect soundness issues I have prioritized these three cases. Appendix A.4.1 contains a partial proof of Property 5.2.1 which covers these three cases along with another two.

As well as statement interpretation soundness, we also have expression interpretation soundness.

Prop. 5.2.3: Expression Read Soundness

for all $\diamond$ in $\{ \rightarrow , \Rightarrow \}$:

$$\begin{array}{c} \Omega \vdash M \dot{\diamond} t \dashv \Omega' \\ \Delta \vdash N \diamond v \dashv \Delta' \\ \Delta : \Omega \end{array} \text{ implies } \begin{array}{c} \Delta' : \Omega' \\ \Omega' \vdash v : t \end{array}$$

Prop. 5.2.4: Expression Write Soundness

for all $\diamond$ in $\{ \leftarrow , \Leftarrow \}$:

$$\begin{array}{c} \Omega \vdash M \dot{\diamond} t \dashv \Omega' \\ \Delta \vdash M \diamond v \dashv \Delta' \\ \Delta : \Omega \\ \Omega \vdash v : t \end{array} \text{ implies } \Delta' : \Omega'$$

In the interpretation $\Omega \vdash M \diamond t \dashv \Omega'$, Ω is always an input, Ω' is always an output, and t is an input during write and an output during read. This difference between t being an input and an output is why read interpretation properties and write interpretation properties are usually separate.

5.2.2 Monotonicity

In this context, monotonicity means "if the input is narrowed, the output is narrowed". It is useful to think of the width of a type or an environment in terms of information: concluding a value has a narrow type gives you more information than concluding it has a wide type.

The extreme of this is the $\top$ type which gives you no information about the value it is typing whatsoever. This is why it is used to represent uninitialized data. The environment equivalent of this is the $\emptyset$ environment, which effectively maps every variable to $\top$ (due to rearrangements allowing $x \mapsto \top$ to be introduced at any point for any variable via ⟨Allocation⟩). $\top$ and $\emptyset$ are the widest types and environments respectively.

Note on Graphical Representations

Occasionally properties have graphical representations. In these representations, the origin represents no information (maximally wide type/environment, $\top$, $\emptyset$) and distance from the origin represents information gain/type narrowing. Lines represent a derivation that relates the input objects (x-axis) and the output objects (y-axis).

Prop. 5.2.5: Statement Interpretation Monotonicity

for all $\diamond$ in $\{ \dot{\rightsquigarrow} , \overset{(\cdot)}{\Rightarrow} \}$:

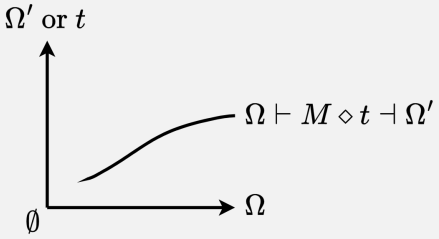
$$\begin{array}{c} \Omega_0 \vdash S \sqsubseteq S' \diamond v_0 \\ \Omega_1 \sqsubseteq \Omega_0 \end{array} \text{ implies there exists } v_1 \text{ s.t. } \begin{array}{c} \Omega_1 \vdash S \sqsubseteq S' \diamond v_1 \\ \Omega_1 \vdash v_1 \sqsubseteq v_0 \end{array}$$

// visualisation: monotic line + ...

Prop. 5.2.6: Expression Read Monotonicity

for all $\diamond$ in $\{ \overset{(\cdot)}{\to}, \overset{(\cdot)}{\Rightarrow} \}$:

$$\begin{array}{c}\Omega_0' \sqsubseteq \Omega_0 \\ \Omega_0 \vdash M \diamond t \dashv \Omega_1\end{array} \text{ implies there exists } \Omega_1', t' \text{ s.t. } \begin{array}{c}\Omega_1' \sqsubseteq \Omega_1 \\ \Omega_1' \vdash t' \sqsubseteq t \\ \Omega_0' \vdash M \diamond t' \dashv \Omega_1'\end{array}$$



Prop. 5.2.7: Expression write-interpretation is monotonic

for all $\diamond$ in $\{ \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow} \}$:

$$\begin{array}{c}\Omega_0' \sqsubseteq \Omega_0 \\ \Omega_0' \vdash t' \sqsubseteq t \\ \Omega_0 \vdash M \diamond t \dashv \Omega_1\end{array} \text{ implies there exists } \Omega_1' \text{ s.t. } \begin{array}{c}\Omega_1' \sqsubseteq \Omega_1 \\ \Omega_0' \vdash M \diamond t' \dashv \Omega_1'\end{array}$$

Prop. 5.2.8: Statement bounds are respected

for all $\diamond$ in $\{ \to, \Rightarrow \}$:

$$\Omega \vdash S \sqsubseteq S' \diamond t \text{ implies there exists a } u \text{ s.t. } \begin{array}{r}\Omega \vdash S \sqsubseteq \_ \diamond u \\ \Omega \vdash S' \sqsubseteq \_ \dot{\rightsquigarrow} t \\ \Omega \vdash u \sqsubseteq t\end{array}$$

5.2.3 Subtyping Preservation

We introduce subtyping between terms, like so;

Def. 5.2.1: Expression Subtyping

$$\frac{\begin{array}{c}\Omega \vdash M \Rightarrow m \dashv \Omega_m \\ \Omega \vdash N \Rightarrow n \dashv \Omega_n \\ \Omega_m \sqsubseteq \Omega_n \\ \Omega_m \vdash m \sqsubseteq n\end{array}}{\Omega \vdash M \sqsubseteq N}$$

Def. 5.2.2: Statement Subtyping

$$\frac{\Omega \vdash S_a \sqsubseteq S_b \Rightarrow t}{\Omega \vdash S_a \sqsubseteq S_b}$$

Prop. 5.2.9: Expression Subtyping Preservation

$$\frac{\Omega \vdash M \sqsubseteq N}{\Omega' \sqsubseteq \Omega} \text{ implies } \Omega' \vdash M \sqsubseteq N$$

Boolean `not` counter example

The fact this does not hold is the motivation behind bounded statement interpretation because, with bounded statement interpretation, we can define statement subtyping and an equivalent subtyping preservation theorem. This is the primary reason why statements and expressions are separated from expressions.

Prop. 5.2.10: Statement Subtyping Preservation

$$\frac{\Omega \vdash S_a \sqsubseteq S_b}{\Omega' \sqsubseteq \Omega} \text{ implies } \Omega' \vdash S_a \sqsubseteq S_b$$

Statement Bound Motivation

Def. 5.2.3: Expression-Like Statement Subtyping

$$\frac{\Omega \vdash S_a \sqsubseteq * \Rightarrow t_a \quad \Omega \vdash S_b \sqsubseteq * \Rightarrow t_b \quad \Omega \vdash t_a \sqsubseteq t_b}{\Omega \vdash S_a \hat{\sqsubseteq} S_b}$$

Prop. 5.2.11: Expression-Like Statement Subtyping Non-Preservation

$$\frac{\Omega \vdash S_a \sqsubseteq S_b}{\Omega' \sqsubseteq \Omega} \text{ does not imply } \Omega' \vdash S_a \sqsubseteq S_b$$

5.2.4 Determinism

include or remove this section

5.2.5 Information Gain/Loss

Prop. 5.2.12: Read-Interpretation Widens

for all $\diamond$ in $\{ \Rightarrow, \dashrightarrow, \dashrightarrow \}$

$$\Omega \vdash M \diamond t \dashv \Omega' \text{ implies } \Omega \sqsubseteq \Omega'$$

// below $\Omega = \Omega'$ line

Prop. 5.2.13: Write-Interpretation Narrows

for all $\diamond$ in $\{ \Leftarrow, \dashleftarrow, \dashleftarrow \}$

$$\Omega \vdash M \diamond t \dashv \Omega' \text{ implies } \Omega' \sqsubseteq \Omega$$

// above $\Omega = \Omega'$ line

The narrower an environment is, the more information it contains. These two theorems state that reading from the environment always *uses* information and writing to the environment *adds* information.

Prop. 5.2.14: Type Union is Precise

$$\Omega \vdash t_0 \sqcup t_1 = t \text{ iff } \Omega \vdash t \sqsubseteq t_0 \text{ or } \Omega \vdash t \sqsubseteq t_1$$

Prop. 5.2.15: Environment Union is not Precise

// todo

Chapter 6

Evaluation

Ochre has two goals: to allow the programmer to encode strong properties in the type system, and to be efficiently executable on hardware. Section 6.2 evaluates Ochre against the former goal, and Section 6.1 evaluates against the latter.

A secondary goal of this research is to make a pleasant and powerful language with useful abstractions. This is discussed in Section **??**.

The contribution of this research is the design and specification of a language, as opposed to an implementation, so the contributions will be evaluated against how well the design & type checking algorithm lays the foundations for a future implementation.

6.1 Type System

The type system presented must reject well-formed programs, and reject ill-formed ones. This section evaluates whether or not this is the case in two distinct ways: firstly Section 5.1 shows the typing rules in action for a collection of programs which should or shouldn't be accepted. Then, Section 5.2 states a set of theorems which should hold.

6.2 Performance

maybe explicitly talk about cache coherency and pointer indirection and function pointers

For the presented design to be good, it must describe a language with semantics that can easily be translated into efficient machine code by a compiler. This section discusses the various language features of Ochre and their performance implications.

Due to having a borrow checker, the abstract interpretation introduced in Section 4 statically determines when objects are dropped, which means it can insert any necessary memory frees into the resultant binary, removing the requirement for a garbage collector.

Being able to mutate data structures in place also allows programmers to express efficient algorithms provided they don't break the *aliasing xor mutability* invariant, like Rust.

Ochre does not have native machine integers, which restricts the programmer to using Peano arithmetic or similar. This is disastrous for performance, and would not be tolerated in even the slowest languages. However, the design presented, and its type checker are perfectly compatible with integers (they would be similar to atoms), so while this work does not directly include efficient integers, I consider them compatible with it, and adding them would be a matter of engineering. The decision not to include them is discussed further in Section 4.6.

As presented, the type system does not support unboxed pairs, which means Ochre programs as currently stated have a lot of unnecessary indirection in their data structures. Much like not supporting machine integers, this is intolerable for a production systems language. However, the type system as presented is compatible with adding them in the future, and adding them at this point would detract from the core concepts. To demonstrate this compatibility, Appendix A.2 lays out a potential method for adding unboxed types.

Ochre does not have efficient contiguous arrays, which hurts the implementation of several dynamic structures, but after unboxed pairs are implemented, contiguous arrays are just many nested pairs. It is unclear how you would efficiently lookup the n^{th} element in such a structure, but I am hopeful it would just be a matter of engineering/adding the right optimizations.

Chapter 7

Conclusion

7.1 Future Work

7.1.1 Reduce Feature Set, Increase Rigor

Section section:properties attempts to argue that the system presented is sound, somewhat unconvincingly. Although I believe it could be proven by a better semanticist than myself, and/or with much more time, I think the best way of getting to that point is by reducing the feature set and progressively building them back. I suggest two such feature sets:

Och

Och would lack dependent types and mutability, but keep the core principle that terms are their own type, and subtyping/structural typing in general.

Och would have the following syntax and environment:

$$
\begin{array}{lll}
M, N ::= & & \textit{// term} \\
 & x \mid y \mid z & \textit{// runtime variable} \\
 & X \mid Y \mid Z & \textit{// comptime variable} \\
 & \texttt{'a} & \textit{// atom construction} \\
 & M\texttt{,}N & \textit{// pair construction} \\
 & M\texttt{.0} & \textit{// pair left access} \\
 & M\texttt{.1} & \textit{// pair right access} \\
 & MN & \textit{// application} \\
 & M\ \texttt{->}\ N & \textit{// abstraction} \\
 & \_ & \textit{// uninitialised} \\
 & T \texttt{ | } U & \textit{// type union} \\
 & M\texttt{:}T & \textit{// type constraint} \\
 & M = N & \textit{// assignment} \\
 & \texttt{match}\, M\, \{\, \overrightarrow{M'\ \texttt{=>}\ S}\, \} & \textit{// match statement}
\end{array}
\qquad
\begin{array}{lll}
\Omega\ ::= & & \textit{// environment} \\
 & \emptyset & \textit{// empty env.} \\
 & \Omega, x \mapsto v & \textit{// runtime variable} \\
 & \Omega, X \mapsto v & \textit{// comptime variable} \\
 & & \\
t, u ::= & & \textit{// type/value} \\
 & \{\overrightarrow{\texttt{'a}}\} & \textit{// atom} \\
 & (t, u) & \textit{// pair} \\
 & (t \to u) & \textit{// function} \\
 & \top & \textit{// top}
\end{array}
$$

Figure 7.1: Och syntax

Because Och does not have references or mutation, it would not need move semantics, and therefore would not need the destructive/non-destructive modality. With this change, the difference between the runtime and comptime modalities might become negligible to the point of redundancy, which would reduce the interpretation down to only the read/write and abstract/concrete modalities.

I speculate that these simplifications would make a soundness proof feasible, and undertaking such a proof effort would cause changes that would leave the language on firmer foundations.

Ochr

Once the core subtyping system is shown to be sound, useful, and implementable, dependent types should be added.

Efforts should be made to not introduce syntax into the abstract environment for the sake of calculating return types as the current version of Ochre has, as this introduces a large amount of complexity into the soundness proof and makes the re-usability of subtyping derivations very labor intensive. This could be done by having the syntax wrapped in a construct that captures the environment needed to evaluate it, similar to how Haskell functions are embedded within the definition of `Value` in Löh et al. [2010], Section 2.4.

Ochr would be Och extended as minimally as possible to include dependent types.

Unlike Ochre as presented, Ochr would allow for dependencies between variables in the abstract environment, which would make environment union precise. This would solve the issue of computing the join, or the "merge problem" in Mezzo [Protzenko, 2014], and en-

able match statements to occur in non-terminal positions without duplication of the code afterward.

7.1.2 Increase Feature Set, Increase Usability

There are many important features missing from Ochre which will be needed in order for it to become a useful tool for formal verification. These include:

Returning References From Functions

Aeneas' method of borrow checking is compatible with returning references from functions, so that method should be portable to Ochre.

This is an extremely important feature for user-defined containers. Without references, any getter you define must move the objects out of the container, which prevents efficient in-place mutation.

Performance Critical Features

As discussed in Section 6.2, there are a couple of features which would drastically increase the performance with relatively little effort: primitive numeric data types, and unboxed pairs.

These would be mostly a matter of engineering, and I believe would complicate the system, so just like I believe Och and Ochr should be explored fully before Ochre, I believe Ochre should be explored fully before adding these features.

Ergonomic Improvements

Functions should be able to capture non-comptime environments, as you can with closures in Rust.

Scoping does not currently respect brackets. There is no syntactic difference to the user between defining a new variable and mutating an existing one. These are not interesting for research purposes but are important for having predictable and compositional semantics.

Stretch Features

A `?` operator which passed the continuation to an expression - This would allow the programmer to write expressions like `foo(x -> bar(y -> x + y))` as `foo? + bar?`. If `foo`

has the type $(A \to B) \to B$, then `foo?` has the type A.

This would be useful for defining constructs like early returns, async/await, yield/generators, and give a powerful abstraction for programmers to use in libraries, like incremental computation.

It could also have utility while using Ochre as a theorem prover because it has a very similar type signature to RAA.

Reverse Functions - Writing to a function application is undefined, but maybe it should be. The interpretation of writing to a function application could be to run some sort of reverse function, which determines for a given value how to write it to the argument. This could be used to define custom pattern matching.

7.1.3 Undo Regrettable Design Decisions

There are a couple of design mistakes that I would not have made with the knowledge I have now.

Inprecise Environment Union - Environment type union cannot be precise because dependencies between variables are not supported (see Property 5.2.5 and surrounding discussion). Initially, dependencies between variables were avoided to simplify, but it turns out that not having a precise environment union introduces the requirement for statements, which complicates things much further. I hope in the future I can change this and simplify the presented system.

Storing Terms in Abstract Environment - The terms used to define a function or a pair are stored in the environment, so they can be re-used to extract their precise types later. They do not store the environment in which they were first evaluated. A large reason for this design decision was to support recursion efficiently; take the following definition of Peano naturals:

```
1  Nat = ('zero, 'unit) | ('succ, Nat);
2  Nat.1.1.1.1.0; // perfectly valid, and equal to {'zero, 'succ}
```

When the right-hand side of the assignment is evaluated, `Nat` is mapped to $\top$ in the environment, to reflect the fact we do not yet know anything about `Nat`. Right-element access then re-executes the `Nat` term, with a narrower context that has a more precise definition of `Nat`.

This takes advantage of the fact that it is evaluated with a more accurate environment later, so I could not remove this immediately, but I believe with future work it could be removed.

7.1.4 Implementation

The goal of this project is to enable formal verification of low-level systems code. In order to do that Ochre must have an implementation, and be usable.

To do this I would implement Ochre in a Rust macro, so it could be invoked as such:

```
fn main() {
  let result = ochre! {
    Nat = ('zero, 'unit) | ('succ, Nat);
    add(x: &Nat, y: &Nat): Nat = {
      match x {
        ('zero, 'unit) => ('zero, 'unit),
        ('succ, px) => ('succ, add(px, y))
      }
    };
    one = ('succ, ('zero, 'unit));
    two = ('succ, ('succ, ('zero, 'unit)));
    add(&one, &two)
  };

  println!("{}", result); // ('succ, ('succ, ('succ, ('zero, 'unit))))
}
```

And potentially allow importing of pure Ochre files from this macro, like such:

```
  fn main() {
    let result = ochre! { import("./add.oc") };

    println!("{}", result); // ('succ, ('succ, ('succ, ('zero, 'unit))))
  }
```

The Rust library which defines this macro could also define traits (type classes) that allow the programmer to define how Rust types are converted to and from Ochre types so that they can pass them to Ochre functions, and use the results in Rust.

The parser for a large subset of a previous version of Ochre is implemented in this way, and code generation/abstract interpretation for a few simple constructs (assignment, atoms, pairs) is implemented, but for this project I decided to prioritize the theory work.

With this interface, programmers could take an existing Rust codebase, and incrementally convert it to Ochre as and when they need stronger properties about their program proven.

Bibliography

ATS-Home, a. URL https://www.cs.bu.edu/~hwxi/atslangweb/Home.html. pages 12

ATS-Implements, b. URL https://www.cs.bu.edu/~hwxi/atslangweb/Implements.html. pages 12

C gcc vs Classic Fortran - Which programs are fastest? (Benchmarks Game). URL https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/gcc-ifc.html. pages 7

Polonius - Current status and roadmap. URL https://rust-lang.github.io/polonius/current_status.html. pages 64

The Rust Programming Language - The Rust Programming Language, a. URL https://doc.rust-lang.org/book/title-page.html. pages 19

Rust vs C++ g++ - Which programs are fastest? (Benchmarks Game), b. URL https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gpp.html. pages 7

StackOverflow developer survey, 2021. URL https://insights.stackoverflow.com/survey/2021. pages 2

Are we stack-efficient yet?, November 2022. URL https://web.archive.org/web/20221128082216/https://arewestackefficientyet.com/. pages 7

Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. $\Pi\Sigma$: Dependent types without the sugar. In *Functional and Logic Programming: 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings 10*, pages 40–55. Springer, 2010. pages 16

Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, et al. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. pages 2

Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305, 2017. pages 2

Son Ho and Jonathan Protzenko. Aeneas: Rust Verification by Functional Translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP):711–741, August 2022. ISSN 2475-1421. doi: 10.1145/3547647. URL http://arxiv.org/abs/2206.07185. pages 11, 14, 20, 36, 39, 57

Graydon Hoare, February 2022. URL https://twitter.com/graydon_pub/status/1492792051657629698. pages 2

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, January 2018. ISSN 2475-1421. doi: 10.1145/3158154. URL https://dl.acm.org/doi/10.1145/3158154. pages 36

Charlie Lidbury. *Ochre: A Dependently Typed Systems Programming Language*. MEng Individual Project, Imperial College London, 2024. pages iv

Andres Löh, Conor McBride, and Wouter Swierstra. A Tutorial Implementation of a Dependently Typed Lambda Calculus. *Fundamenta Informaticae*, 102(2):177–207, 2010. ISSN 01692968. doi: 10.3233/FI-2010-304. URL https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/FI-2010-304. pages 74

Jacob R Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R Wilcox, and Xueyuan Zhao. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–210, 2020. pages 2

Xie Ningning, Leonardo De Moura, Daan Leijen, and Alex Reinking. Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 96–111, Virtual Canada, June 2021. ACM. ISBN 978-1-4503-8391-2. doi: 10.1145/3453483.3454032. URL https://dl.acm.org/doi/10.1145/3453483.3454032. pages 65

Jonathan Protzenko. *Mezzo: A Typed Language for Safe Effectful Concurrent Programs*. PhD thesis, Université Paris Diderot - Paris 7, September 2014. URL https://inria.hal.science/tel-01086106. pages 74

A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. ISSN 1460-244X. doi: 10.1112/plms/s2-42.1.230. URL https://onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230. pages 66

Sebastian Ullrich. Kha/electrolysis, January 2024. URL https://github.com/Kha/electrolysis. pages 20, 57

Leslie Blackett Wilson and Robert G. Clark. *Comparative Programming Languages*. International Computer Sciences Series. Addison-Wesley, Harlow London New York [etc.], 3rd ed edition, 2001. ISBN 978-0-201-71012-0. pages 7

Ludwig Wittgenstein. *Tractatus Logico-Philosophicus*. Oxford World's Classics. Oxford University press, Oxford, 1922. ISBN 978-0-19-886137-9. URL https://www.gutenberg.org/files/5740/5740-pdf.pdf. pages iv

Appendix A

Appendicies

A.1 Formal Verification using (Dependent) Types

The primary motivation behind adding dependent types to a language is so you can perform theorem proving/formal verification in the type system. In some languages, like Lean, this is done to mechanize mathematical proofs to prevent errors and/or shorten the review process; in other languages, like F*, Idris or ATS this is done to allow the programmer to reason about the runtime properties of their programs. However, they are all just pure functional languages with dependent types, whether you choose to use this expressive power for maths or programs the underlying type system is the same.

So the question is how can you represent logical statements as (potentially dependent) types and use the type checker to prove them? This is best understood via a simpler version: proving logical tautologies using Haskell's type system.

Boolean Tautologies in Haskell

The Curry-Howard correspondence states there is an equivalence between the theory of computation, and logic. Specifically: types are analogous to statements, and terms (values) are analogous to proofs. Under this analogy, $5 : \mathbb{N}$ states that 5 is a proof of $\mathbb{N}$.

We can use this to represent logical statements as types. Here is how various constructs in logic translate over to types (given in Haskell).

Logical Statement	Equivalent Haskell Type	Explanation
$\top$	`()`	Proving true is trivial, so unit type.
$\bot$	`!`	There exists no proof of false, so empty type.
$a \Rightarrow b$	`a -> b`	If you have a proof of a, you can use it to construct a proof of b.
$a \wedge b$	`(a, b)`	A proof of a and a proof of b combined into one proof.
$a \vee b$	`Either a b`	This proof was either constructed in the presence of a proof of a or a proof of b.

For example, to prove the logical statement $(a \wedge b) \Rightarrow a$, we must define a Haskell term with type `(a, b) -> a`, which can be done as such:

```
proof :: (a, b) -> a
proof (a, b) = a
```

For another example, we can prove $((a \wedge b) \vee (a \wedge c)) \Rightarrow (a \wedge (b \vee c))$, which you might want to convince yourself of separately before moving on, by providing a Haskell term of type `Either (a, b) (a, c) -> (a, Either b c)`.

```
proof' :: Either (a, b) (a, c) -> (a, Either b c)
proof' (Left (a, b)) = (a, Left b)
proof' (Right (a, c)) = (a, Right c)
```

With this we can construct proofs for logical tautologies, but how do we go further and construct proofs for statements like "If you get any number and double it, you get an even number".

Dependent Types are Quantifiers

Let's now define a function $even$ which returns a type, such that any term of type $even(n)$ is proof that n is even. To do this, $even$ returns a *type*: $\top$ if n is even, $\bot$ otherwise. I.e. $even(4) = \top$ and $even(5) = \bot$. The logical statement $\forall n : \mathbb{Z}.even(2n)$ can be represented by the type $(n : \mathbb{Z}) \rightarrow even(2 * n)$. If we had a term of this type, we could give it any integer n, and it would return proof that $2n$ is even.

This cannot be represented in Haskell, because $(\mathbf{n} : \mathbb{Z}) \rightarrow even(2 * \mathbf{n})$ is a dependent type, hence we need a dependently typed language like Agda. This is an example of Haskell's non-dependent type system not being able to express quantifiers like $\forall$ or $\exists$ over values.

A.2 Supporting Unboxed Pairs

While atoms, functions, and references are unboxed in Ochre, pairs are always heap-allocated. As discussed in Section 6.2, this will hinder the performance of compiled Ochre programs. This appendix lays out a rough plan for adding unboxed types to the formal semantics for Ochre, to make the point that the research as presented is compatible with such an extension.

A potential method of adding unboxed pairs:

1. **Add** $\text{box}\, t$**, a new type which represents an explicit heap allocation**, along with corresponding constructors and eliminators. This is required so the programmer can heap allocate objects whose size is not comptime known.

2. **Edit the abstract interpretations to pass around (type, size) pairs instead of just types**. This would involve all read arrows returning the size of the value being read, and all write arrows taking the size of the data being written.

   Set the size of pairs to the sum of the size of the elements in the pair. Because the type of the right-hand side can depend on the left, this can cause some sizes to be unknown. Because of this, arrows may return an unknown size, and if the user needs to put something of unknown size on the stack, they must put it in a box.

   The size of a match statement is the largest size of any of its branches.

A.3 Derivations

A.3.1 Hello World Program Derivation

This appendix gives a derivation for the Hello World Program shown in Listing 18.

Derivation:

$$
\begin{array}{l}
\Omega_0 \vdash L_2 ; L_3 ; L_4 ; L_5 ; L_7 \sqsubseteq \texttt{*} \Rightarrow (\text{'hello}, \_ \to \text{'world}) \qquad \Omega_0 = \emptyset \\
\quad \textit{// pair = ('world, 'hello)} \\
\quad \Omega_0 \vdash \texttt{('world, 'hello)} \Rightarrow (\text{'world}, \_ \to \text{'hello}) \\
\qquad \Omega_0 \vdash \texttt{'world} \Rightarrow \text{'world} \\
\qquad \Omega_0 \vdash \texttt{'hello} \Rightarrow \text{'hello} \\
\quad \Omega_0 \vdash \texttt{pair} \Leftarrow (\text{'world}, \_ \to \text{'hello}) \dashv \Omega_1 \qquad \langle\text{Rearrange-Before}\rangle \\
\qquad \Omega_0 \hookrightarrow \Omega_0' \qquad \langle\text{Allocate}\rangle \\
\qquad\quad \Omega_0' = \Omega_0, \texttt{pair} \mapsto \top \qquad \Omega_0' = \emptyset, \texttt{pair} \mapsto \top \\
\qquad \Omega_0' \vdash \texttt{pair} \Leftarrow (\text{'world}, \_ \to \text{'hello}) \dashv \Omega_1 \\
\qquad\quad \Omega_1 = \Omega_0' \left[\dfrac{\texttt{pair} \mapsto (\text{'world}, \_ \to \text{'hello})}{\texttt{pair} \mapsto \top} \right] \qquad \Omega_1 = \emptyset, \texttt{pair} \mapsto (\text{'world}, \_ \to \text{'hello}) \\
\quad \Omega_1 \vdash L_3 ; L_4 ; L_5 ; L_7 \sqsubseteq \texttt{*} \Rightarrow (\text{'hello}, \_ \to \text{'world}) \\
\qquad \mathcal{D}_3
\end{array}
$$

$\mathcal{D}_3 =$

$\Omega_1 \vdash L_3 ; L_4 ; L_5 ; L_7 \sqsubseteq * \Rightarrow$ ('hello, _ → 'world)
  // temp = pair.0
  $\Omega_1 \vdash$ `pair.0` $\Rightarrow$ 'world $\dashv \Omega_1''$
    $\Omega_1 \vdash$ `pair` $\Rightarrow$ ('world, _ → 'hello) $\dashv \Omega_1'$
      $\Omega_1' = \Omega_1 \left[\frac{\texttt{pair} \mapsto \top}{\texttt{pair} \mapsto (\text{'world}, \_ \to \text{'hello})} \right]$ $\quad \Omega_1' = \emptyset, \texttt{pair} \mapsto \top$
    $\Omega_1' \vdash \mathsf{left}\,(\text{'world}, \_ \to \text{'hello}) = \text{'world}$
    $\Omega_1' \vdash \mathsf{right}\,(\text{'world}, \_ \to \text{'hello}) = \text{'hello}$
    $\Omega_1' \vdash$ `pair` $\Leftarrow (\top, \_ \to \text{'hello}) \dashv \Omega_1''$
      $\Omega_1'' = \Omega_1' \left[\frac{\texttt{pair} \mapsto (\top, \_ \to \text{'hello})}{\texttt{pair} \mapsto \top} \right]$ $\quad \Omega_1'' = \emptyset, \texttt{pair} \mapsto (\top, \_ \to \text{'hello})$
  $\Omega_1'' \vdash$ `temp` $\Leftarrow$ 'world $\dashv \Omega_2$ $\quad \langle$Rearrange-Before$\rangle$
    $\Omega_1'' \hookrightarrow \Omega_1'''$ $\quad \langle$Allocate$\rangle$
      $\Omega_1''' = \Omega_1'', \texttt{temp} \mapsto \top$ $\quad \Omega''' = \emptyset, \texttt{pair} \mapsto (\top, \_ \to \text{'hello}), \texttt{temp} \mapsto \top$
    $\Omega_2 = \Omega_1''' \left[\frac{\texttt{temp} \mapsto \text{'world}}{\texttt{temp} \mapsto \top} \right]$ $\quad \Omega_2 = \emptyset, \texttt{pair} \mapsto (\top, \_ \to \text{'hello}), \texttt{temp} \mapsto \text{'world}$
  $\Omega_2 \vdash L_4 ; L_5 ; L_7 \sqsubseteq * \Rightarrow$ ('hello, _ → 'world)
    $\mathcal{D}_4$

$\mathcal{D}_4 =$

$\Omega_2 \vdash L_4 ; L_5 ; L_7 \sqsubseteq * \Rightarrow$ ('hello, _ → 'world)
  // pair.0 = pair.1
  $\Omega_2 \dashv$ `pair.1` $\Rightarrow$ 'hello $\dashv \Omega_2'$
    // Similar to $\Omega_1 \vdash$ *pair.0* $\Rightarrow$ *'world* $\dashv \Omega_1''$ $\quad \Omega_2' = \emptyset, \texttt{pair} \mapsto (\top, \_ \to \top), \texttt{temp} \mapsto \text{'world}$
  $\Omega_2' \dashv$ `pair.0` $\Leftarrow$ 'hello $\dashv \Omega_3$
    $\Omega_2' \vdash$ `pair` $\Rightarrow (\top, \_ \to \top) \dashv \Omega_2''$
      $\Omega_2'' = \Omega_2' \left[\frac{\texttt{pair} \mapsto \top}{\texttt{pair} \mapsto (\top, \_ \to \top)} \right]$ $\quad \Omega_2'' = \emptyset, \texttt{pair} \mapsto \top, \texttt{temp} \mapsto \text{'world}$
    $\Omega_2'' \vdash \mathsf{left}\,(\top, \_ \to \top) = \top$
    $\Omega_2'' \vdash \mathsf{right}\,(\top, \_ \to \top) = \top$
    $\Omega_2'' \vdash$ `pair` $\Leftarrow$ ('hello, _ → $\top$) $\dashv \Omega_2'''$
      $\Omega_3 = \Omega_2'' \left[\frac{\texttt{pair} \mapsto (\text{'hello}, \_ \to \top)}{\texttt{pair} \mapsto \top} \right]$ $\quad \Omega_3 = \emptyset, \texttt{pair} \mapsto (\text{'hello}, \_ \to \top), \texttt{temp} \mapsto \text{'world}$
  $\Omega_3 \vdash L_5 ; L_7 \sqsubseteq * \Rightarrow$ ('hello, _ → 'hello)
    $\mathcal{D}_5$

$\mathcal{D}_5 =$

$\Omega_3 \vdash L_5 ; L_7 \sqsubseteq * \Rightarrow$ ('hello, _ → 'hello)
  // pair.1 = temp
  $\Omega_3 \vdash$ `temp` $\Rightarrow$ 'world $\dashv \Omega_3'$
  $\Omega_3' = \Omega_3 \left[\frac{\texttt{temp} \mapsto \top}{\texttt{temp} \mapsto \text{'world}} \right]$ $\quad \Omega_3' = \emptyset, \texttt{pair} \mapsto (\text{'hello}, \_ \to \top), \texttt{temp} \mapsto \top$
  $\Omega_3' \vdash$ `pair.1` $\Leftarrow$ 'world $\dashv \Omega_4$
  // Similar to $\Omega_2' \dashv$ *pair.0* $\Leftarrow$ *'hello* $\dashv \Omega_3$ $\quad \Omega_4 = \emptyset, \texttt{pair} \mapsto (\text{'hello}, \_ \to \text{'world}), \texttt{temp} \mapsto \top$
  $\Omega_4 \vdash L_7 \sqsubseteq * \Rightarrow$ ('hello, _ → 'world)
    $\mathcal{D}_7$

$\mathcal{D}_7 =$

$\Omega_4 \vdash L_7 \sqsubseteq \texttt{*} \Rightarrow (\text{'hello}, \_ \to \text{'world})$
// *pair*
$\Omega_4 \vdash \texttt{pair} \Rightarrow (\text{'hello}, \_ \to \text{'world}) \dashv \Omega_4'$
$\Omega_4' = \Omega_4 \left[\dfrac{\texttt{pair} \mapsto \top}{\texttt{pair} \mapsto (\text{'hello}, \_ \to \text{'world})} \right] \quad \Omega_4' = \emptyset, \texttt{pair} \mapsto \top, \texttt{temp} \mapsto \top$
$\Omega_4' \vdash \mathsf{drop}$
$\emptyset, \texttt{pair} \mapsto \top, \texttt{temp} \mapsto \top \vdash \mathsf{drop}$
$\emptyset, \texttt{pair} \mapsto \top \vdash \mathsf{drop}\, \top$
$\emptyset, \texttt{pair} \mapsto \top \vdash \mathsf{drop}$
$\emptyset \vdash \mathsf{drop}\, \top$
$\emptyset \vdash \mathsf{drop}$

A.3.2 Mutating a Dependent Pair

This appendix shows the derivation which type checks the definition of the `overwrite` function from Listing 19.

$\Omega_1 \vdash$ `(p:&mut Same) -> 'unit {*p.0 = 'a; *p.1 = 'a; 'unit}` $\Rightarrow$ `(p:&mut Same)` $\to$ `'unit`
$\Omega_1 \vdash$ `(p:&mut Same)` $\underset{\text{max}}{\Leftarrow} \mathsf{borrow}^{\mathsf{m}}\, l\, p_0 \dashv \Omega_{10}$
$\Omega_1 \vdash$ `&mut Same` $\dot{\rightsquigarrow} \mathsf{borrow}^{\mathsf{m}}\, l\, p_0 \dashv \Omega_1'$
$\Omega_1 \vdash$ `Same` $\dot{\rightsquigarrow} p_0$
`Same` $\mapsto p_0 \in \Omega_1$ $\quad p_0 = (\{\text{'a}, \text{'b}\}, L \to L)$
$\Omega_1' = \Omega_1, l \mapsto p_0$ $\quad \Omega_1' = \Omega_1, l \mapsto p_0$
$\Omega_1' \vdash \texttt{p} \Leftarrow \mathsf{borrow}^{\mathsf{m}}\, l\, p_0 \dashv \Omega_{10}$ $\quad \langle$Rearrange-Before$\rangle$
$\Omega_1' \hookrightarrow \Omega_1''$ $\quad \langle$Allocate$\rangle$
$\Omega_1'' = \Omega_1', \texttt{p} \mapsto \top$ $\quad \Omega_1'' = \Omega_1', \texttt{p} \mapsto \top$
$\Omega_1'' \vdash \texttt{p} \Leftarrow \mathsf{borrow}^{\mathsf{m}}\, l\, p_0 \dashv \Omega_{10}$
$\Omega_{10} = \Omega_1' \left[\dfrac{\texttt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_0}{p \mapsto \top} \right]$ $\quad \Omega_{10} = \Omega_1', \texttt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_0$
$\Omega_{10} \vdash$ `*p.0 = 'a; *p.1 = 'a; 'unit` $\sqsubseteq$ `'unit` $\Rightarrow$ `'unit`
$\mathcal{D}_{10}$

$\mathcal{D}_{10} =$

$\Omega_{10} \vdash$ `*p.0 = 'a; *p.1 = 'a; 'unit` $\sqsubseteq$ `'unit` $\Rightarrow$ 'unit $\quad \Omega_{10} \vdash$ `'a` $\Rightarrow$ 'a

$\Omega_{10} \vdash$ `*p.0` $\Leftarrow$ 'a $\dashv \Omega_{11}$ ⟨Rearrange-Before⟩

$\Omega_{10} \hookrightarrow \Omega'_{10}$ // *must set* $*p.0$ *to* $\top$ *to write* ⟨Type-Widen⟩

$\Omega'_{10} = \Omega_{10}\left[\frac{\mathtt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_1}{\mathtt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_0}\right]$ $\quad \Omega'_{10} = \Omega'_1, p \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_1$

$\Omega_{10} \vdash \mathsf{borrow}^{\mathsf{m}}\, l\, p_0 \sqsubseteq \mathsf{borrow}^{\mathsf{m}}\, l\, p_1$ $\quad p_1 = (\top, \_ \to \{\text{'a}, \text{'b}\})$

$\Omega_{10} \vdash p_0 \sqsubseteq p_1$

$\Omega_{10} \vdash \{\text{'a}, \text{'b}\} \sqsubseteq \top$

$\Omega_{10} \vdash \mathsf{comptime} \dashv \Gamma_1$ $\quad \Gamma_1 = \Omega_1, \mathtt{L} \mapsto \top$

$\Gamma_1 \vdash \_ \dot{\leftarrow} \top$

$\Gamma_1 \vdash \mathtt{L} \dot{\leftarrow} \{\text{'a}, \text{'b}\} \dashv \Gamma'_1$

$\Gamma'_1 = \Gamma_1\left[\frac{\mathtt{L} \mapsto \{\text{'a}, \text{'b}\}}{\mathtt{L} \mapsto \top}\right]$

$\Gamma'_1 \vdash \mathtt{L} \sqsubseteq \{\text{'a}, \text{'b}\} \dot{\rightarrow} \{\text{'a}, \text{'b}\}$

$\Gamma'_1 \vdash L \dot{\rightarrow} \{\text{'a}, \text{'b}\}$

$\mathtt{L} \mapsto \{\text{'a}, \text{'b}\} \in \Gamma'_1$

$\Gamma'_1 \vdash \{\text{'a}, \text{'b}\} \dot{\rightarrow} \{\text{'a}, \text{'b}\}$

$\Gamma'_1 \vdash \{\text{'a}, \text{'b}\} \sqsubseteq \{\text{'a}, \text{'b}\}$

$\{\text{'a}, \text{'b}\} \subseteq \{\text{'a}, \text{'b}\}$

$\Omega'_{10} \vdash$ `*p.0` $\Leftarrow$ 'a $\dashv \Omega_{11}$ // *write to* `*p.0`

$\Omega'_{10} \vdash$ `*p` $\Rightarrow p_1 \dashv \Omega'''_{10}$

$\Omega'_{10} \vdash \mathtt{p} \Rightarrow \mathsf{borrow}^{\mathsf{m}}\, l\, p_1 \vdash \Omega''_{10}$

$\Omega''_{10} = \Omega'_{10}\left[\frac{\mathtt{p} \mapsto \top}{\mathtt{p} \mapsto p_1}\right]$ $\quad \Omega''_{10} = \Omega'_1, \mathtt{p} \mapsto \top$

$\Omega''_{10} \vdash \mathtt{p} \Leftarrow \mathsf{borrow}^{\mathsf{m}}\, l\, \top \vdash \Omega'''_{10}$

$\Omega'''_{10} = \Omega''_{10}\left[\frac{\mathtt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, \top}{\mathtt{p} \mapsto \top}\right]$ $\quad \Omega'''_{10} = \Omega'_1, \mathtt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, \top$

$\Omega'''_{10} \vdash \mathsf{left}\, p_1 = \top$

$\Omega'''_{10} \vdash \mathsf{right}\, p_1 = \{\text{'a}, \text{'b}\}$

$\Omega'''_{10} \vdash$ `*p` $\Leftarrow p_2 \dashv \Omega_{11}$ $\quad p_2 = (\text{'a}, \_ \to \{\text{'a}, \text{'b}\})$

$\Omega'''_{10} \vdash \mathtt{p} \Rightarrow \mathsf{borrow}^{\mathsf{m}}\, l\, \top \dashv \Omega''''_{10}$

$\Omega''''_{10} = \Omega'''_{10}\left[\frac{\mathtt{p} \mapsto \top}{p \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, \top}\right]$ $\quad \Omega''''_{10} = \Omega'_1, \mathtt{p} \mapsto \top$

$\Omega''''_{10} \vdash \mathtt{p} \Leftarrow \mathsf{borrow}^{\mathsf{m}}\, l\, p_2 \dashv \Omega_{11}$

$\Omega_{11} = \Omega''''_{10}\left[\frac{\mathtt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_2}{p \mapsto \top}\right]$ $\quad \Omega_{11} = \Omega'_1, \mathtt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_2$

$\Omega_{11} \vdash$ `*p.1 = 'a; 'unit` $\sqsubseteq$ `'unit` $\Rightarrow$ 'unit

$\mathcal{D}_{11}$

$\mathcal{D}_{11} =$

$\Omega_{11} \vdash$ `*p.1 = 'a; 'unit` $\sqsubseteq$ `'unit` $\Rightarrow$ 'unit

$\Omega_{11} \vdash$ `'a` $\Rightarrow$ 'a

$\Omega_{11} \vdash$ `*p.1` $\Leftarrow$ 'a $\dashv \Omega_{12}$ $\quad \Omega_{12} = \Omega'_1, \mathtt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_3$

// *Similar to* $\Omega'_{10} \vdash$ `*p.0` $\Leftarrow$ 'a $\dashv \Omega_{11}$ $\quad p_3 = (\text{'a}, \_ \to \text{'a})$

$\Omega_{12} \vdash$ `'unit` $\sqsubseteq$ `'unit` $\Rightarrow$ 'unit

$\mathcal{D}_1 2$

$\mathcal{D}_{12} =$

$\Omega_{12} \vdash$ 'unit $\sqsubseteq$ 'unit $\Rightarrow$ 'unit
$\quad \Omega_{12} \vdash$ 'unit $\Rightarrow$ 'unit
$\quad$ // $\Omega_{12} \vdash \mathsf{drop} =$
$\quad \Omega_1, l \mapsto p_0, p \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_3 \vdash \mathsf{drop}$ *// final cleanup*
$\quad\quad \Omega_1, l \mapsto p_0 \vdash \mathsf{drop}\,(\mathsf{borrow}^{\mathsf{m}}\, l\, p_3) \dashv \Omega_1$
$\quad\quad\quad \Omega_1 = \Omega_1, l \mapsto p_0 \backslash \{l \mapsto p_0\}$
$\quad\quad\quad \Omega_1 \vdash p_3 \sqsubseteq p_0$
$\quad\quad\quad\quad \Omega_1 \vdash$ 'a $\sqsubseteq$ {'a, 'b}
$\quad\quad\quad\quad\quad$ {'a} $\subseteq$ {'a, 'b}
$\quad\quad\quad\quad \Omega_1 \vdash \mathsf{comptime} \dashv \Gamma_2$ $\quad \Gamma_2 = \Omega_1, \mathtt{L} \mapsto \top$
$\quad\quad\quad\quad \Gamma_2 \vdash \mathtt{L} \dot{\rightsquigarrow}$ 'a $\dashv \Gamma_2'$
$\quad\quad\quad\quad\quad \Gamma_2' = \Gamma_2 \left[\frac{\mathtt{L} \mapsto \text{'a}}{\mathtt{L} \mapsto \top}\right]$
$\quad\quad\quad\quad \Gamma_2 \vdash \_ \dot{\rightsquigarrow}$ 'a
$\quad\quad\quad\quad \Gamma_2' \vdash$ 'a $\sqsubseteq \mathtt{L} \dot{\rightsquigarrow}$ 'a
$\quad\quad\quad\quad\quad \Gamma_2' \vdash$ 'a $\dot{\rightsquigarrow}$ 'a
$\quad\quad\quad\quad\quad \Gamma_2' \vdash L \dot{\rightsquigarrow}$ 'a
$\quad\quad\quad\quad\quad\quad \mathtt{L} \mapsto$ 'a $\in \Gamma_2'$
$\quad\quad\quad\quad\quad \Gamma_2' \vdash$ 'a $\sqsubseteq$ 'a
$\quad\quad\quad\quad\quad\quad$ {'a} $\subseteq$ {'a}
$\Omega_{12} \vdash$ 'unit $\sqsubseteq * \dot{\rightsquigarrow}$ 'unit
$\Omega_{12} \vdash$ 'unit $\sqsubseteq$ 'unit

A.4 Properties and Proofs

A.4.1 Statement Soundness

This section seeks to prove Property 5.2.1 (Statement Interpretation Soundness):

$$\begin{array}{c} \Delta \vdash S \sqsubseteq * \Rightarrow v \\ \Omega \vdash S \sqsubseteq * \;\Rightarrow\; t \text{ implies } \Omega \vdash v : t \\ \Delta : \Omega \end{array}$$

We start by assuming all of the premises:

$$\begin{array}{c} \Delta \vdash S \sqsubseteq * \Rightarrow v \\ \Omega \vdash S \sqsubseteq * \;\Rightarrow\; t \\ \Delta : \Omega \end{array} \tag{A.1}$$

for arbitrary statements S, environments Ω, Δ and types t, and v.

Then we take the proof by cases on the derivations used to construct $\Omega \vdash S \sqsubseteq * \;\Rightarrow\; t$ and $\Delta \vdash S \sqsubseteq * \Rightarrow v$, then conclude $\Omega \vdash v : t$ by induction. Conceptually, we are defining a recursive function which takes our premise derivations, and returns a derivation of $\Omega \vdash v : t$.

Assumption: Concrete and Abstract Rearrangements are Synced

We assume that the derivations used to construct $\Delta \vdash S \sqsubseteq * \Rightarrow v$ and $\Omega \vdash S \sqsubseteq * \;\dot{\Rightarrow}\; t$ are *in sync*. This would mean that both or neither are an instance of ⟨Rearrange-Before⟩ or ⟨Rearrange-After⟩. The intuition behind this is that the abstract interpretation drops a variable if and only if the concrete interpretation drops a variable. This could be made provable so we would not have to assume it, but I believe that would involve the abstract interpretation somehow inserting drop signals into the terms that the concrete interpretation reads and acts upon. This is possible, and I believe only a matter of effort, but it would complicate the interpretations further, slowing down future more important work.

Case - S is an expression M.

The assumed derivations must be

$$\frac{\Delta \vdash M \Rightarrow v \dashv \Delta' \quad \Delta' \vdash \mathsf{drop}}{\Delta \vdash M \sqsubseteq * \Rightarrow} \text{ and } \frac{\begin{array}{c}\Omega \vdash * \dot{\rightsquigarrow} \top \\ \Omega \vdash M \;\dot{\Rightarrow}\; t \dashv \Omega' \\ \Omega' \vdash t \sqsubseteq \top \quad \Omega' \vdash \mathsf{drop}\end{array}}{\Omega \vdash M \sqsubseteq * \;\dot{\Rightarrow}\; t} \tag{A.2}$$

for some environments Δ' and Ω'.

Using Property 5.2.1 (Expression Read Soundness) on $\Delta \vdash M \Rightarrow v \dashv \Delta'$ and $\Omega \vdash M \;\dot{\Rightarrow}\; t \dashv \Omega'$ we have $\Omega \vdash v : t$ and $\Delta \sqsubseteq \Omega$. □

Case - S is an assignment $M \texttt{ = } N\texttt{;}\ S'$.

Case - S is a match statement $\texttt{match}\ M\ \texttt{\{}\overrightarrow{M' \texttt{ => } S'}\texttt{\}}$.

A.4.2 Expression Read Soundness

This section seeks to prove Property 5.2.1 (Statement Read Soundness):

for all $\diamond$ in $\{ \dot{\rightarrow}, \dot{\Rightarrow} \}$:

$$\begin{array}{l}\Delta \vdash M \diamond v \dashv \Delta' \\ \Omega \vdash M \;\dot{\diamond}\; t \dashv \Omega' \\ \quad \Delta : \Omega\end{array} \text{ implies } \begin{array}{c}\Delta' : \Omega' \\ \Omega' \vdash v : t\end{array}$$

We start by assuming $\Delta \vdash M \diamond v \dashv \Delta'$, $\Omega \vdash M \;\dot{\diamond}\; t \dashv \Omega'$, and $\Delta : \Omega$ for arbitrary expressions M, environments Ω, Ω', Δ' and types t and v.

We then take the proof by an exhaustive set of cases.

Case - M is an atom constructor 'a

The assumed derivations $\Delta \vdash M \diamond v \dashv \Delta'$ and $\Omega \vdash M \mathbin{\dot\diamond} t \dashv \Omega'$ must be $\Delta \vdash$ 'a$\dot\diamond$'a and $\Omega \vdash$ '$a \diamond$'a which tells us $v =$ 'a, $\Delta' = \Delta$, and $\Omega' = \Omega$.

Our proof goals re-written are now $\Omega \vdash$ 'a $\sqsubseteq$ 'a and $\Delta : \Omega$. The former holds directly from $\langle$*def.* $\sqsubseteq$ *for* 'a$\rangle$, and the latter holds because we previously assumed it. □

Case - M is a runtime variable x and $\diamond$ is $\Rightarrow$.

The assumed derivations $\Delta \vdash M \Rightarrow v \dashv \Delta'$ and $\Omega \vdash M \mathbin{\dot\Rightarrow} t \dashv \Omega'$ must be

$$\frac{\Delta' = \Delta\left[\frac{x \mapsto \top}{x \mapsto v}\right]}{\Delta \vdash x \Rightarrow v \dashv \Delta'} \text{ and } \frac{\Omega' = \Omega\left[\frac{x \mapsto \top}{x \mapsto t}\right]}{\Omega \vdash x \mathbin{\dot\Rightarrow} t \dashv \Omega'}. \tag{A.3}$$

Since x now maps to $\top$ in both environments $\Delta' : \Omega'$ because environment subtyping is just variable-wise subtyping and $\Omega' \vdash \top \sqsubseteq \top$. $\top$ is concrete, so Δ' remains concrete. $\Omega' \vdash v : t$ by $\langle$*def.* $:$ *for* $\Omega\rangle$. □.

Case - M is a function application FA and $\diamond$ is $\Rightarrow$.

The assumed derivations $\Delta \vdash M \Rightarrow v \dashv \Delta'$ and $\Omega \vdash M \mathbin{\dot\Rightarrow} t \dashv \Omega'$ must be:

$$\frac{\begin{array}{c}\Delta \vdash F \to (M \to S) \\ \Delta \vdash A \Rightarrow a \dashv \Delta_1 \\ \Delta_1 \vdash M \Leftarrow a \dashv \Delta_2 \\ \Delta_2 \vdash S \Rightarrow v \dashv \Delta'\end{array}}{\Delta \vdash FA \Rightarrow v \dashv \Delta'} \text{ and } \frac{\begin{array}{c}\Omega \vdash F \mathbin{\dot\to} (\dot M \to S_t) \\ \Omega \vdash A \mathbin{\dot\Rightarrow} \dot a \dashv \Omega_1 \\ \Omega_1 \vdash \dot M \mathbin{\dot\Leftarrow} \dot a \dashv \Omega_2 \\ \Omega_2 \vdash S_t \sqsubseteq * \dot\rightsquigarrow t \\ \Omega_1 \vdash \text{drop}\, t \dashv \Omega'\end{array}}{\Omega \vdash FA \mathbin{\dot\Rightarrow} t \dashv \Omega'} \tag{A.4}$$

Subgoal $M = \dot M$ **and** $\Omega \vdash (M \to S) \sqsubseteq (\dot M \to S_t)$ - We use Expression Read Soundness (induction) on $\Delta \vdash F \to (M \to S)$ and $\Omega \vdash F \mathbin{\dot\to} (\dot M \to S_t)$ to conclude that $\Omega \vdash (M \to S) \sqsubseteq (\dot M \to S_t)$. By $\langle$*def.* $\sqsubseteq$ *for* $M \to S\rangle$, we know that their domains M and $\dot M$ must be equal. From here on we will use M in $\dot M$'s place.

Subgoal $\Delta_2 : \Omega_2$ - We use Expression Read Soundness (5.2.1, induction) on $\Delta \vdash A \Rightarrow a \dashv \Delta_1$ and $\Omega \vdash A \mathbin{\dot\Rightarrow} \dot a \dashv \Omega_1$ which gives us $a : \dot a$ and $\Delta_1 : \Omega_1$. This allows us to use Expression Write Soundness (5.2.1) on $\Delta_1 \vdash M \Leftarrow a \dashv \Delta_2$ and $\Omega_1 \vdash \dot M \mathbin{\dot\Leftarrow} \dot a \dashv \Omega_2$ to get $\Delta_2 : \Omega_2$. Δ_2 and Ω_2 are the environments that will be used to execute the function body and return type respectively.

Now we know our concrete and abstract environments going into the function body are well-typed, we can reason about whether or not the two statements are well-typed. We can use the fact that the concrete function is a subtype of the abstract function ($\Omega \vdash (M \to S) \sqsubseteq (\dot M \to S_t)$) to get:

$$\frac{\begin{array}{c}\Omega \vdash \mathsf{comptime} \dashv \Gamma \\ \Gamma \vdash M \underset{\mathsf{max}}{\Leftarrow} m_{max} \dashv \Gamma' \\ \Gamma' \vdash S \sqsubseteq S_t \Rightarrow t_{max}\end{array}}{\Omega \vdash M \to S \sqsubseteq M \to S_t}$$

That derivation gives us $\Gamma' \vdash S \sqsubseteq S_t \Rightarrow t_{max}$, which means the bodies are well-typed against each other with the *widest* possible input type. Since we are writing a narrower value to M at the function call site, and expression write interpretation is monotonic (**??**), the environment we type the function bodies with (Ω_2) must be smaller than Γ'. Therefore, because statement interpretation is monotonic (5.2.2), we can turn $\Gamma' \vdash S \sqsubseteq S_t \Rightarrow t_{max}$ into $\Omega_2 \vdash S \sqsubseteq S_t \Rightarrow t'$ for some t' where $\Omega_2 \vdash t' \sqsubseteq t_{max}$.

Because statement bounds are respected (**??**) there exists a u such that

$$\begin{array}{c}\Omega_2 \vdash S \sqsubseteq * \Rightarrow u \\ \Omega_2 \vdash S_t \sqsubseteq * \dot{\rightsquigarrow} t' \\ \Omega_2 \vdash u \sqsubseteq t'\end{array} \tag{A.5}$$

We now prove $v : u$, $u \sqsubseteq t'$, and $t' \sqsubseteq t$ so we can combine them into $v : t$, which is one of the two statements we need to prove to show function application soundness (the other being $\Delta' : \Omega'$).

Subgoal: $\Omega_2 \vdash v : u$ - We have $\Delta_2 \vdash S \Rightarrow v \dashv \Delta_3$ from our initial derivation of the concrete function application, and $\Omega_2 \vdash S \sqsubseteq * \Rightarrow u$ from (A.5). Because statement interpretation is sound, and $\Delta_2 : \Omega_2$, we have $\Omega_2 \vdash v : u$, as required for this subgoal.

Subgoal: $\Omega_2 \vdash u \sqsubseteq t'$ - Directly given by (A.5).

Subgoal: $\Omega_2 \vdash t' \sqsubseteq t$ - (A.5) gives us $\Omega_2 \vdash S_t \sqsubseteq * \dot{\rightsquigarrow} t'$ and (A.4) gives us $\Omega_2 \vdash S_t \sqsubseteq * \dot{\rightsquigarrow} t$. Because $\Omega_2 \sqsubseteq \Omega_2$ (**??**) and statement interpretation in monotonic (**??**), we have $\Omega_2 \vdash t' \sqsubseteq t$.

Goal: $\Omega_2 \vdash v : t$ - Our first subgoal gives us $\Omega_2 \vdash v \sqsubseteq u$ and $\mathsf{concrete}\, v$ from $\langle \textit{def.} : \textit{for}\, t \rangle$. Therefore, from our subgoals we have $\Omega_2 \vdash v \sqsubseteq u$, $\Omega_2 \vdash u \sqsubseteq t'$, and $\Omega_2 \vdash t' \sqsubseteq t$. Because subtyping is transitive **??**, we can use these to get $\Omega_2 \vdash v \sqsubseteq t$, and since $\mathsf{concrete}\, v$ from our first subgoal, we have $\Omega_2 \vdash v : t$ as required.

The above has proven the soundness of the return value from a function. Now we now reason about the function's side effects, in the form of its effects on the environment (goal: $\Delta' : \Omega'$).

The only side effects functions can have are those caused by dropping references (in their arguments). Functions cannot mutate non-local variables because their definitions only capture comptime variables, which cannot be mutated (or, more formally speaking, can only be narrowed).

When a function body is interpreted, it is ensured that every reference is mutated back to

the type it originally had at the beginning of the function body. Take a function f, which takes in an argument `x` of type $\mathsf{borrow}^{\mathsf{m}}\, l\, t$. The function body is obligated to drop all of its local variables, which includes this borrow. This is checked by introducing a loan restriction into the context at the start of the function body, then only allowing the borrow to be terminated if it matches this loan restriction. Since f has previously passed type checking ($\langle$*def.* $\Rightarrow$ *for* $M \to S\rangle$), we know that its body writes back a value of type t to the referenced loans or a subtype of t. This means it is sound to approximate the function's side effects by immediately dropping its arguments, which is done by $\Omega_1 \vdash \mathsf{drop}\, t \dashv \Omega'$.

It would be very labor-intensive to formalize this logic into a proof and possibly require introducing more logic to the interpretations. This would be worthwhile work if it was not for the fact that I think overall, the current model of side effects is not very elegant and would be worth refining as future work. This refining would make any proofs I make about the current system redundant. Therefore I have chosen to prioritize other cases, such as function application and type annotations.

Case - M is a function definition M'`->`S_t`{`S`}`.

The assumed derivations $\Delta \vdash M \Rightarrow v \dashv \Delta'$ and $\Omega \vdash M \;\Rightarrow\; t \dashv \Omega'$ must be:

$$\frac{/\!/\ \textit{no checking}}{\Delta \vdash M'\texttt{->}S_t\texttt{\{}S\texttt{\}} \Rightarrow M \to S} \text{ and } \frac{\begin{array}{c}\Omega \vdash \mathsf{comptime} \dashv \Gamma \\ \Gamma \vdash M' \underset{\mathsf{max}}{\Leftarrow} m \dashv \Gamma' \\ \Gamma' \vdash S \sqsubseteq S_t \;\Rightarrow\; u\end{array}}{\Omega \vdash M'\texttt{->}S_t\texttt{\{}S\texttt{\}} \;\Rightarrow\; M \to S_t} \tag{A.6}$$

Our goal is to prove the return value is sound ($\Omega \vdash (M \to S) \sqsubseteq (M \to S_t)$) and our output environment is sound ($\Delta : \Omega$).

We immediately have $\Delta : \Omega$ because it is one of our starting premises (A.4.1).

We can use $\Omega \vdash \mathsf{comptime} \dashv \Gamma$, $\Gamma \vdash M' \underset{\mathsf{max}}{\Leftarrow} m \dashv \Gamma'$, and $\Gamma' \vdash S \sqsubseteq S_t \;\Rightarrow\; u$ with $\langle$*def.* $\sqsubseteq$ *for* $M \to S_t\rangle$ to get $(\Omega \vdash (M \to S) \sqsubseteq (M \to S_t))$ immediately. $\square$

This case is so short because $\langle$*def.* $\sqsubseteq$ *for* $M \to S_t\rangle$ is almost an exact match of $\langle$*def.* $\Rightarrow$ *for* M'`->`S_t`{`S`}`$\rangle$. The complexity of function checking is all in the application case. This reflects the fact that defining an ill-typed function never causes soundness issues at runtime, it is *calling* an ill-typed function that makes your state unsound.

Case - M is a type annotation $M' : T$.

The assumed derivations $\Delta \vdash M \Rightarrow v \dashv \Delta'$ and $\Omega \vdash M \;\Rightarrow\; t \dashv \Omega'$ must be:

$$\frac{\Delta \vdash M \Rightarrow v \dashv \Delta'}{\Delta \vdash M' : T \Rightarrow v \dashv \Delta'} \text{ and } \frac{\begin{array}{c}\Omega \vdash M \;\Rightarrow\; m \dashv \Omega' \\ \Omega' \vdash T \dot{\rightsquigarrow} t \\ \Omega' \vdash m \sqsubseteq t\end{array}}{\Omega \vdash M' : T \;\Rightarrow\; t \dashv \Omega'} \tag{A.7}$$

Because expression read interpretation is sound (5.2.1, induction), we can use $\Delta \vdash M \Rightarrow v \dashv \Delta'$ and $\Omega \vdash M \Rightarrow m \dashv \Omega'$ to get $\Omega' \vdash v : m$ and $\Delta' \sqsubseteq \Omega'$.

Our assumed abstract derivation gives us $\Omega' \vdash m \sqsubseteq t$, which can be used with $\Omega' \vdash v : m$ and the transitivity of subtyping (**??**) to get $\Omega' \vdash v : t$. □

Remaining Cases - In this section we have covered every language construct that has a different concrete interpretation to abstract interpretation. I am hopeful that the remaining constructs are provable because there are so few differences between their concrete and abstract interpretations. I leave these cases as future work so that I may prioritize other parts of this research.