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 24, 2024

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

Abstract

As much an art as it is a science, programming language design continually improves over time, allowing programmers to write better programs. Some designs enable better compiler optimizations, others enable stronger static analyses; the design space is an ever-evolving field, fraught with trade-offs.

One such trade-off historically has been that between memory safety and performance. Rust has mostly solved this by introducing the borrow checker, which avoids the need for a costly runtime garbage collector while remaining memory-safety.

In this report, we work to push the state-of-the-art of language design further and allow the programmer to prove stronger properties than just memory safety *while remaining performant*. We do this by formally defining and reasoning about Ochre, a dependently typed, low-level systems language. In Ochre, programmers can use the type system to prove strong properties that they cannot in non-dependently typed languages such as Rust or Haskell, and they can do so with a Rust-like memory management strategy that does not require a garbage collector.

Ochre is not the first performant language to feature dependent types, much like Rust was not the first language to offer safe manual memory management, but it achieves this combination of features with a novel technique, which offers a substantial improvement in ergonomics thanks to ownership semantics.

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 also argue there 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 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

In the beginning[1], there was C. When it was released in 1972, it was a revolution in terms of both ergonomics and safety. Its performance and control allowed it to replace programs previously written in assembly code, and its abstractions and type system allowed programmers to focus more on their program logic.

The next big step up from C for low-level systems programming came with C++ in 1985. C++ offered a plethora of abstractions like generics which allow much greater re-use of code, which ultimately resulted in the viability of a rich standard library with reusable data structures, something C lacks.

However, both of these C's have a fatal flaw: lack of safety. When programming in C and C++, it is common to introduce memory safety bugs that cause significant harm to real-world systems. Memory safety bugs account for roughly *70%* of real-world security vulnerabilities [Pro, Mem] and the white house cited it as a matter of national cyber-security [Office of the National Cyber Director, 2024, p. 8]. Software engineers mostly responded by switching to slower but memory-safe languages like Python and Java, accepting the trade-off in performance that came with safety. A lot of code needs to be performant though, which is why there are still so many memory-safety-related vulnerabilities in production.

Rust [Rus, 2015] has mostly solved this trade-off between memory safety and performance, as well as many concurrency bugs, with advancements in static analysis. The Rust *borrow checker* allows you to manage your memory manually like you would in C or C++, but points out any mistakes you make at compile time, which gives you the performance of using C or C++ and the safety of Python or Java.

Safe, performant, languages predate Rust significantly [Morrisett et al.], but Rust brought it to the masses with its unusually ergonomic method of memory management.

Memory safety is important, but it alone is too weak a property - as evidenced by the Herculean efforts being made to formally verify foundational software [Klein et al., 2009, Lorch et al., 2020, Ferraiuolo et al., 2017, Bhargavan et al., 2017]. Alongside this slow, incremental, push towards reliable systems code, the functional programming world has made vast strides in the design and implementation of type systems and the properties they can prove. "Modern" features of Rust, which it is often hailed for, like algebraic data types, type classes, and typed errors, have all been in Haskell in an ergonomic way for decades.

Theorem provers are functional languages with a type system that allows the programmer to prove almost [Kurt Gödel, 1931] any property about the runtime behavior of their pro-

[1] Before C was assembly, Fortran, and the Big Bang; but C is a good starting point.

grams. Even if you do not directly want to prove properties about the code you are writing, you may still benefit from using a language that allows you do to so:

- You may want to prove properties about your code in the future if your requirements for correctness increase.
- Other people working on the same codebases as you may want to prove properties about their programs.
- If you want the libraries you are importing to be proven correct, the library author needs to be using something capable of proofs, and they are probably using the same language as you.

These benefits are certainly achievable without the programming language doing the proving, much like it is possible to prove the memory safety of programs written in memory-unsafe languages [Klein et al., 2009], but this approach requires sacrifices in ergonomics, fundamentally because the language has not been designed from the ground up with this use match in mind. It is analogous to creating an untyped dialect of Haskell, and then using a third-party tool to statically analyze Haskell programs instead of having the type system woven into the design of the language itself.

The goal of Ochre is to solve the design problems which are stopping theorem proving from being brought to the systems programming masses. We do this integrating dependent types into the type system, and broader design of, of a programming language in an unusually ergonomic and performance-compatible way, much like Rust did with memory safety.

Much like Rust has, I believe Ochre will benefit immensely from targetting low-level systems code: systems code is often critical to foundational software like operating systems, databases, and standard libraries; each line is developed with great care, and mistakes frequently cause widespread security vulnerabilities. At the same I consider the power of the type system presented to be a great success, we can type programs with mutability and dependent types tightly interwoven, as shown in Section **??**, time, any performance improvements made to these systems affect a massive install base, so large efforts are made to be fast, which more or less rules out languages that do not give the programmer control over data layout and memory management. These factors come together to make a perfect storm of demand for theorem-proving capabilities, as well as a tolerance for difficult-to-understand language concepts such as dependent types. This is why Rust has succeeded despite ownership being a difficult feature for programmers conceptually.

Somewhat surprisingly, the techniques that Rust uses to achieve memory safety can also be used to make dependent types compatible with systems programming; as this research demonstrates.

1.1 Contributions

The contributions of this report are as follows:

1. The formal definition of Ochre, a language which uses a novel method to support dependent types, mutability, and manual memory management simultaneously in an ergonomic fashion.
2. A demonstration that this method of type-checking works in practice, in the form of typing derivations for a set of motivating example programs.
3. An investigation into the properties of Ochre, including a partial soundness proof.

1.2 Report Outline

Chapter 2 contains an exploration of the concepts required to understand the work presented, as well as a deeper dive into the work Ochre is built upon, specifically Rust and Aeneas [Ho and Protzenko, 2022]. Section 2.5 goes on to review the literature surrounding this work, including other efforts towards high-performance languages with proof capabilities, as well as the broader domain of verification of high-performance programs.

In Chapter 3 Ochre is presented by gradually introducing its language features. This serves to act as a reference for the reader to use throughout the report to remind themselves how various language features behave. We also introduce the static analysis without going into the abstract interpretation, in the form of showing what the various abstract states are in various scenarios.

Then, Ochre is defined formally in Chapter 4, along with the abstract interpretation used to type check it. Section 4.5 contains the bulk of the complexity of this definition, including how to type-check all language constructs, and how to execute them at runtime (which we use to reason about soundness).

Chapter 5 tests the core contributions against their goal of developing a novel way to combine mutability with dependent types in two distinct ways. First, Section 5.1 answers this by using the system to type check a set of motivating programs, then Section 5.2 reasons about how Ochre will behave for arbitrary programs, in the form of a partial proof of soundness, along with an exploration of the other properties required for this soundness proof to hold.

Finally, in Chapter 7 we summarise and evaluate the work presented, and discuss future work which should be undertaken.

Chapter 2

Background

In this chapter, we cover the concepts required to understand how Ochre programs are type checked, as well as the surrounding literature on the topic.

2.1 Dependent Types

In most languages and theories, types and values are separate; the definition of a type cannot depend on a value. Dependently typed languages blur this separation: the definition of a type can use terms. The two dependent types used by this research are *dependent functions* and *dependent pairs*, referred to in the literature as Π and Σ types respectively.

The return type of a dependent function can depend on the *value* of its argument. For example, you could define a function that takes a number n as input, which returns n-tuple of numbers as an output, where the size of n-tuple returned depends on which number was passed as input. This is also possible in dynamic typing, where there are no types at all, but with dependent types, the caller can statically determine which of these two types the return value will be without executing the function body.

Do demonstrate dependent types, we use Caclulus of Constructions, denoted as λC [Barendregt, 1991], which is slightly different from the MLTT norm [Martin-L{\"o}f, Per and Sambin, Giovanni, 1984], but simpler and it is enough to understand this research.

We define an extension to the normal lambda calculus, which introduces the dependent function type constructor $\Pi x : M.N$ and *sorts* s. Are terms and typing context are given in Figure 2.1, our reduction rules in 2.2 and our typing judgements in 2.3. The system is included in its entirity for the sake of completeness, but the readers attention is drawn specifically to Application.

In the application typing rule, the function *type* is evaluated under the context, in much the same way function application is defined in the beta reduction rule for application. This means calculating the return type of a function can trigger arbitrary execution, much like executing one can, which allows the return type to depend on the input type.

While dependent types can be nice to have by themselves, a large part of their motivation is using them to perform formal verification. Appendix A.1 contains an explanation of how types can be used to represent statements, and their programs proofs of those statements. If you are willing to accept that dependent types can be used to perform formal verification,

$$A, B, C ::= x \mid s \mid MM \mid \lambda x : M.N \mid \Pi x : M.N$$
$$\Gamma ::= \emptyset \mid \Gamma, x : A$$

Figure 2.1: λC terms

$$\textsc{Application}\quad \frac{}{(\lambda x : A.B)C \rightarrow_\beta B[C/x]} \qquad \textsc{Argument-Reduction}\quad \frac{A \rightarrow_\beta A'}{\lambda x : A.B \rightarrow_\beta \lambda x : A'.B} \qquad \textsc{Body-Reduction}\quad \frac{B \rightarrow_\beta B'}{\lambda x : A.B \rightarrow_\beta \lambda x : A'.B}$$

$$\textsc{Pi-Argument-Reduction}\quad \frac{A \rightarrow_\beta A'}{\Pi x : A.B \rightarrow_\beta \Pi x : A'.B} \qquad \textsc{Pi-Body-Reduction}\quad \frac{B \rightarrow_\beta B'}{\Pi x : A.B \rightarrow_\beta \Pi x : A'.B}$$

Figure 2.2: Beta-Reduction Rules, $=_\beta$ is the transitive, reflexive closure of $\rightarrow_\beta$

$$\textsc{Axiom}\quad \frac{}{\vdash * : \Box} \qquad \textsc{Start}\quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : A \vdash A : B} \qquad \textsc{Weakening}\quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$$

$$\textsc{Application}\quad \frac{\Gamma \vdash C : \Pi x : A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash Ca : B[a/x]} \qquad \textsc{Conversion}\quad \frac{\Gamma \vdash A : B \qquad B =_\beta B' \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}$$

$$\textsc{Product}\quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2} \qquad \textsc{Abstraction}\quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash b : B \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \lambda : A.b : \Pi x : A.B}$$

Figure 2.3: Typing Rules for λC

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 to add dependent types to a language, not use them to prove properties ourselves.

2.2 Abstract Interpretation

This system presented is an abstract interpretation. This fact is not leveraged in any proofs, and no results from abstract interpretation are brought over, but having an understanding of the concepts behind abstract interpretation will be useful for understanding the architecture of the presented language.

First formalized by Partick Cousout and Radhia Cousot in the late 1970s [Cousot and Cousot, 1977], abstract interpretation is a method of soundly approximating the semantics of programs, based on monotonic functions over ordered sets. Abstractly interpreting a program is similar to regular interpretation, like a Python interpreter might do, but instead of keeping track of the precise program state, it keeps track of an approximation of it.

In this section, I will explain abstract interpretation in a less general way than Cousout an Cousout did, to give the reader an easier time understanding the relevant aspects. The more general explanation can be found in Appendix A.2 and a more formal one in **?**.

Abstract interpretation can be seem as a framework for developing program analyses, which requires the semanticist to define the following mathematical objects:

Concrete and Abstract Set - Each element of the *concrete set* represents a program state. For example, if you are analysing a function which increments a number by one, it would map the concrete state 1 to the concrete state 2. Both 1 and 2 are elements of our concrete set, $\mathbb{N}$. Each element of the *abstract set* represents an approximation of the concrete states the program could be in. Continuing the above example, we could store the possible *parities* of the number in our abstract state. The increment function would map $\{\mathsf{odd}\}$ to $\{\mathsf{even}\}$, and our abstract set would be $\mathcal{P}(\{\mathsf{true}, \mathsf{false}\})$. The fact it maps $\{\mathsf{odd}\}$ to $\{\mathsf{even}\}$ is intuitively: if the input is an odd numbers, our output is an even number. Our abstract set here is the power set of $\{\mathsf{true}, \mathsf{false}\}$ instead of just $\{\mathsf{true}, \mathsf{false}\}$ to represent uncertainty: $\{\mathsf{true}, \mathsf{false}\}$ represents "the state could be odd, or even".

Concrete and Abstract Semantics - The *concrete semantics* for a program maps a set of possible start states to a set of possible end states. The concrete semantics f for our above increment program might look something like $f(x) = x + 1$. The *abstract semantics* for a program map the start abstract states to the end abstract state. The abstract semantics f' for the continuing example would express the fact that incrementing a number always flips its parity: $f(l) = \{\mathsf{odd} \mid \mathsf{even} \in l\} \cup \{\mathsf{odd} \mid \mathsf{even} \in l\}$.

Concretization Function and Abstraction Function - The *concretisation function* maps from an approximation to the set of concrete values they could be approximating. The *abstraction function* maps from a concrete state to an approximation (element of our abstract set). It answers: for a given concrete value, what is its approximation? In the above, it would map 1 to $\{\text{odd}\}$ and 5 to $\{\text{odd}\}$, reflecting the fact that the input is definitely odd.

With this framework in place, the research can be summarised as an abstract interpretation where:

- The abstract set is a mapping from variables to types, and the concrete set is a mapping from variables to values.
- Our abstract semantics are type checking, and our concrete semantics are execution or reduction.
- Our abstraction function maps a value to its type, and our concretization function maps a type to its set of values.

As is typical with abstract interpretations, or abstract set is ordered, and our abstract semantics are monotonic, as shown in Section 5.2, but I will leave the explanation here for now.

2.3 Rust

In 2006, in response to a crash in his building's elevator's firmware[1], Gradon Hoare designed Rust, 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.

Since then it has enjoyed significant success, surpassing Haskell, Kotlin, Scala, and Ruby in market share [TIO] and being voted the most loved language for 7 years in a row as of 2023 Sta.

2.3.1 Performance

Rust is a fast language. Its performance is roughly equivalent to that of C and C++ Ben, which are generally accepted as the benchmark of language performance. Rust has enduring performance problems Rus [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].

[1] I cannot find the source of this anecdote, but it is mentioned in the introduction to Ho and Protzenko [2022] and dozens of secondary sources on the internet, search "Rust language elevator firmware origins".

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 match 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.

2.3.2 Ownership

Ownership is the concept that data is a movable and finite resource; it cannot be used in multiple places without first being duplicated. This materializes as a set of rules in Rust that prevent a user from using the same piece of data in multiple places at once. This intuition of data having a location, and needing to be in the right one, is what gives *move semantics* its name. In a language with move semantics, like Rust, using a value destroys it and prevents it from being used again. Preventing this old (moved) value from being used is useful for performance: if a data structure is in a variable, we know that we have exclusive *ownership* of the data structure and all its memory allocations. Therefore, when the variable goes out of scope, we know that nobody will have access to it, and therefore its memory allocations can be given back to the operating system without causing memory safety issues.

Rust's innovation was introducing the concept of *borrowing*: in Rust, you can make a pointer to a variable, and the existence of that pointer disallows the data from being mutated by any means other than that pointer. You may mutate data through a pointer if and only if it is the only pointer that exists to that data. Pointers with these restrictions are so different to use in practice than pointers they are referred to with a separate name, *references*.

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
```

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. It is an error to have more than one mutable reference to a variable simultaneously:

```
let x = 5;
let rx1 = &mut x;
let rx2 = &mut x; // rx1 dropped here
*rx1 = 42; // error: rx1 not in scope
*rx2 = 43;
```

Both of these references cannot exist at the same time, so Rust implicitly *drops* the first when the second is constructed. This causes our usage of the first reference to error, it is no longer in scope for usage.

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. Take the following function:

```
1  fn choose<'a>(returnLeft: bool, left: &'a mut i32, right: &'a mut i32) -> &'a mut i32
2    if returnLeft { left } else { right }
3  }
4
5  let x = 5;
6  let y = 5;
7  let rx = choose(true, &mut x, &mut y);
8  *rx = 42;
9  // lifetime 'a ends
10 println!("{}", x);
```

In the above, `choose` is generic over a lifetime `’a`. The type signature reads “for any lifetime `’a`, if you input a boolean and two references which last *at least* as long as `’a`, the function will return a reference which lasts *at least* as long as `’a`”.

When `choose` is used on line 7, Rust implicitly passes a lifetime which ends between the usage of `rx` and the final print of `x`. During lifetime `’a`, `rx` can be used and `x` cannot.

The following usage of choose *does not* work:

```
1  let x = 5;
2  let y = 5;
3  let rx = choose(true, &mut x, &mut y);
4  println!("{}", x);
5  *rx = 42;
```

Rust cannot find an appropriate lifetime which satisfies all the constraints, because the usage of `x` on line 4 must be after the lifetime ends, and the usage of `rx` must be before the lifetime ends.

There is much more to Rust, but this concept of ownership and borrowing are the relevant parts to Ochre which we build upon.

2.4 Aeneas & The LLBC

Aeneas [Ho and Protzenko, 2022] is a verification toolchain for Rust programs which works by transpiling code written in Rust into the F* theorem prover[2]. This allows the programmer to execute the Rust version of their program, which is performant, and prove properties about the F* version of it, which has dependent types and is a full theorem prover.

Aeneas takes MIR (mid-level IR) code as input, as it is much simpler than the source Rust langauge, and has already goen through type & borrow checking, which means Aeneas does

[2] F* is the first amongst many backends for Aeneas

not have to do either. It then translates this MIR internally to the LLBC (Low-Level Borrow Calculus) which they define.

Once an LLBC term is obtained, Aeneas undergoes a very similar analysis to the one Ochre does, which is where I got the idea for Ochre's general analysis. As it scans through the LLBC, Aeneas passes around a piece of state called the *evaluation context*, which maps each variable to its type and borrow state. A variable's borrow state tells you which of the following statements is true for any given variable:

- This variable owns a value.
- This variable is a mutable or immutable reference to a value.
- This variable owns a value, but has lent that ownership out to a reference.

When a value is borrowed, it is replaced by a *loan*, which, for mutable borrows, prevents the value being read from that location. Below is a poigant example which demonstrates how Aeneas tracks mutable borrows in its evaluation context, which is shown in comments after each line:

```
1   let mut x = 0;         // x ↦ 0
2   let mut px = &mut x;   // x ↦ loan^m l, px ↦ borrow^m l 0
3   *px = 5;               // x ↦ loan^m l, px ↦ borrow^m l 5
4   println!("{}", x);     // x ↦ 5, px ↦ ⊥
```

When the mutable reference is constructed on line 2, it causes x to be *borrowed*. This means the value currently in x is replaced with a loan with a matching *loan identifier* (l) to the constructed borrow. Now the value 0 is stored under the entry for px in the context instead of the entry for x. Line 3 can now mutate the value without causing non-local side effects because the value is the rx entry in the context. The interesting operation is line 4, where we want to read x from the environment but we cannot, because the value has been borrowed. We match the loan identifier stored in x with the loan identifier stored in px to determine that $\mathsf{borrow}^{\mathsf{m}}\, l\, 5$ must be dropped. Dropping the borrow replaces $\mathsf{loan}^{m}\, l$ with 5 in the context, allowing us to read and print x.

As Aeneas is performing this analysis, it outputs dependently typed pure functional code. The details of how this is achieved are omitted because this part of Aeneas is not used in Ochre.

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.

2.5.1 ATS

ATS [Xi, 2017] is the most mature dependently typed programming language with dependent types 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 can be thought of as an eagerly evaluated functional language like OCaml, 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.

2.5.2 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 match, 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 match that they do not wish to use dependent types for their reasoning power.

2.5.3 Magmide

The goal of Magmide [Mag, 2024] 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. Other than that, Magmide is a promising project, although without results it is hard to evaluate or build off of.

2.5.4 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 mutation. 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.

2.5.5 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.

2.5.6 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 of program executions, then by looking at how the environment changes due to different language constructs.

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.

3.1 The Language

This section covers Ochre in a gradual, example-heavy manner, much like programming language tutorials like The Rust Book [?]. 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].

```
  x = 'one;
  rx1 = &x;
  rx2 = &x;
  x; // 'one
  *rx1; // 'one
  *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.

```
  x = ('one, 'two);
  x.0; // 'one
  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:

```
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
```

Match Statements

In Ochre, atoms can be branched on via a match statement. The discriminant of the match statement must be a comptime expression which evaluates to a type, and there must be exactly one branch for each possible value of the scruitinee. Match statements only *read* the scruitinee instead of consuming it, which allows the scruitinee to be used in the branch statements.

```
Bool = 'true | 'false;
not = (b: Bool) -> Bool {
  match 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 -> match 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 -> match 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 {
  match p.0 {
    'true => match p.1 { 'true => 'unit }, // p.1: 'true
    'false => match p.1 { 'false => 'unit }, // p.1: 'false
  }
}
```

Listing 4: Match 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 match, nothing is stored, so the payload is ’unit, in the successor match, we store the natural that we are the successor of, so our payload is Nat.

```
// "manual" ADT encoding
Nat = (T: 'zero | 'succ, match 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 {
  match 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 match 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 match, 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 {
  // ...
}
```

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 $\perp$ in the abstract environment:

```
x = 'hello; // $\{x \mapsto \text{'hello}\}$
y = x; // $\{x \mapsto \bot, y \mapsto \text{'hello}\}$
```

References & Mutation

When you construct a reference, the value is *borrowed*. In the match 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; // $\{x \mapsto \text{'one}\}$
  rx = &mut x; // $\{x \mapsto \mathsf{loan}^m\, l, rx \mapsto \mathsf{borrow}^\mathsf{m}\, l\text{'one}\}$
  *rx = 'two; // $\{x \mapsto \mathsf{loan}^m\, l, rx \mapsto \mathsf{borrow}^\mathsf{m}\, l\text{'two}\}$
  // rx dropped
  x; // $\{x \mapsto \text{'two}, rx \mapsto \bot\}$
```

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; // $\{x \mapsto \text{'one}\}$
  rx = &mut x; // $\{x \mapsto \mathsf{loan}^m\, l, rx \mapsto \mathsf{borrow}^\mathsf{m}\, l\text{'one}\}$
  *rx = 'two; // $\{x \mapsto \mathsf{loan}^m\, l, rx \mapsto \mathsf{borrow}^\mathsf{m}\, l'two\}$
  // rx dropped
  x; // $\{x \mapsto 'two, rx \mapsto \bot\}$
```

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; // $\{x \mapsto 'one\}$
  rx = &x; // $\{x \mapsto \mathsf{loan}^\mathsf{s}\, l'one,\ rx \mapsto \mathsf{borrow}^\mathsf{s}\, l'one\}$
```

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

```
x = 'one; // $\{x \mapsto one\}$
rx1 = &mut x; // $\{\mathtt{x} \mapsto \mathsf{loan}^{m}\, l,\ \mathtt{rx} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, 'one\}$
rx2 = &mut *rx1; // $\{\mathtt{x} \mapsto \mathsf{loan}^{m}\, l,\ \mathtt{rx} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, (\mathsf{loan}^{m}\, l'), \mathtt{rx2} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, 'one\}$
```

Listing 9: A reborrow

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

```
x = 'one; // $\{x \mapsto one\}$
rx1 = &x; // $\{\mathtt{x} \mapsto \mathsf{loan}^{\mathsf{s}}\, l\, 'one,\ \mathtt{rx} \mapsto \mathsf{borrow}^{\mathsf{s}}\, l\, 'one\}$
rx2 = &*rx1; // $\{\mathtt{x} \mapsto \mathsf{loan}^{\mathsf{s}}\, l\, (\mathsf{loan}^{\mathsf{s}}\, l'\, 'one),\ \mathtt{rx} \mapsto \mathsf{borrow}^{\mathsf{s}}\, l\, (\mathsf{loan}^{\mathsf{s}}\, l'\, 'one), rx2 \mapsto \mathsf{borrow}^{\mathsf{s}}\, l'\, 'one\}$
rx3 = &*rx2; // $\{\mathtt{x} \mapsto \mathsf{loan}^{\mathsf{s}}\, l\, (\mathsf{loan}^{\mathsf{s}}\, l'\, (\mathsf{loan}^{\mathsf{s}}\, l''\, 'one)),\ \mathtt{rx} \mapsto \mathsf{borrow}^{\mathsf{s}}\, l\, (\mathsf{loan}^{\mathsf{s}}\, l'\, (\mathsf{loan}^{\mathsf{s}}\, l''\, 'one)),$
             // $\mathtt{rx2} \mapsto \mathsf{borrow}^{\mathsf{s}}\, l'\, (\mathsf{loan}^{\mathsf{s}}\, l''\, 'one), \mathtt{rx3} \mapsto \mathsf{borrow}^{\mathsf{s}}\, l''\, 'one\}$
```

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: $(\{'true, 'false\}, \mathtt{L} \to \mathtt{L})$. 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 match 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;                    // $\{Bool \mapsto \{\text{'true}, \text{'false}\}\}$
BoolPair = (Bool, Bool);                  // $\{..., BoolPair \mapsto (\{\text{'true}, \text{'false}\}, \_ \to \mathtt{Bool})\}$
Same = (Bool, L -> L);                    // $\{..., Same \mapsto (\{\text{'true}, \text{'false}\}, \mathtt{L} \to \mathtt{L}\}$
specificPair = ('true, 'true);            // $\{..., specificPair \mapsto (\text{'true}, \_ \to \text{'}\mathtt{true})\}$
widenedPair = ('true, 'true): Same;       // $\{..., widenedPair \mapsto (\{\text{'true}, \text{'false}\}, \mathtt{L} \to \mathtt{L}\}$
```

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 $(\{\text{'true}, \text{'false}\}, \mathtt{L} \to \mathtt{L})$, which isn't strictly true. Type union between pairs will make the right depend on the left by producing a match statement for each of the possible left atoms, so `('true, 'true) | ('false, 'false` would instead evaluate to $(\{\text{'true}, \text{'false}\}, \mathtt{L} \to$ `match 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.

```
x = 'true; // {x ↦ 'true}
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 $\{\text{'true}, \text{'false}\}$

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.

```
x = 'one; // {x ↦ 'one}
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.

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

X = 'comptime; // {..., X ↦ 'comptime}
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.

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

Match Statements

In each of the branches of a match statement, the type of the scrutinee is narrowed down to a specific atom. This is useful when the match 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 match statement will modify the environment in some possibly different way. These are combined into one environment via an environment-wide union opera-

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

tion, which is the output environment.

```
1    f = (b: 'true | 'false) -> 'unit {
2                            // {b ↦ {'true, 'false}}
3      match 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 match 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 $\{\text{'hello}, \text{'world}\}$. 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 $\{\text{'true}, \text{'false}\}$.

Complex Example Programs

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

Chapter 4

Ochre

Ochre is defined through a syntax, a concrete interpretation on the terms, and an abstract interpretation on the terms. The concrete semantics defined in our abstract interpretation models the programs execution (as a set of reduction rules would) and the abstract semantics defines how the type of a term is derived (as a set of typing rules would).

This chapter starts by defining the prerequisite mathematical objects our abstract interpretation will be operating over in Sections 4.1, 4.2, and 4.3, then Sections 4.4 and 4.5 will define the abstract interpretation itself.

Finally, a discussion defending the choice of language features is left in Section 4.6 for the curious reader.

In Ochre, a type is a set of possible values. There is no way to describe two types that differ only in name, as they will both become aliases to the same type, this is unlike Rust or Haskell where two types can store the same data with the same field names, but be incompatible with each other.

Unlike most languages, type constructors and term constructors are typically the same object. For example, the type of a pair is itself a pair: `('true, 'false): (Bool, Bool)`. In Haskell you have similar syntax, but the type of a pair is a fundamentally different object in the language than a pair itself. This is not so in Ochre. This extends to all language constructs: the type of every term is itself. In fact, a term M is *the most* precise type you can assign to a term M, type annotations serve only to lose type information. This even extends to functions: the type of the identity function is `T -> T`, and an implementation of the identity function is `T -> T`.

The concept of type *width* is used throughout this chapter to help the reader visualize what is going on. A type is wider than another if it is a supertype, and narrower if it is a subtype. Subtype will be defined formally in Section 4.3.

4.1 Syntax

Ochre has categories of terms, *statements* and *expressions*. All expressions are statements, but variable assignment (mutation) and match statements can only occur in statements. Figure 4.1 shows the grammar for these two categories of terms.

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

Figure 4.1: Ochre syntax

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.

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. This simplifies type checking: 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. RustBelt's λ_{Rust} [Jung et al., 2018], Aeneas' LLBC [Ho and Protzenko, 2022, Section 4.3], and Mezzo Protzenko [2014] also enforce the same restriction on their equivilants of match statements.

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 with the following rewrite rule: replace all occurances of $\texttt{match}\ M\ \texttt{\{}\ \overrightarrow{M'\ \texttt{=>}\ S;S'}\ \texttt{\}}$ with $\texttt{match}\ M\ \texttt{\{}\ \overrightarrow{M'\ \texttt{=>}\ S}\ \texttt{\}}\ ;S'$. This 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 restriction. This rewrite rule has the disadvantage of causing an exponential blowup in code size/interpretation derivation size, and more importantly, an exponential blowup in the complexity of the analysis once implemented.

Types can be treated as terms. This is useful in a few rules where we need to construct a term which always interprets to the same type, such as in $\langle \textit{def.} \overset{(\cdot)}{\Rightarrow} \textit{for}\, M\,.\,\mathtt{0}\rangle$.

4.2 Environment, Values, and Types

Our type checker is defined as an abstract interpretation that interprets terms while modifying an *abstract environment*. The abstract environment contains everything we know statically about the program at a given point in the interpretation. Its most common use is storing the type of variables, but it is also used for storing *loan restrictions*, which are used in function checking to ensure function bodies mutate their arguments correctly. Figure 4.2 shows our definition of the abstract environment.

Representation of Types

The abstract environment maps variables to types. Atom types are a mathematical set of atoms, which represent an exhaustive list of atoms inhabiting this type.

Functions store syntax which can be used to turn input into output. If you have a value of type t, and you pass it to a function with type $T \to U$, its return type is calculated by writing t to T, then reading the return type from U, as defined in Section 4.5.3

Pair types store two components, the type of the left element t, and syntax which can be used to turn the type of the left element into the type of the right element $T \to U$. By storing how to turn left into right, instead of storing right directly, we can make the right type *depend* on the left type, this is what makes Ochre pairs dependent.

Borrows and loans are used to track ownership information in the environment. If a type is a loan, it means the underlying value has been borrowed, and cannot be mutated. If the type is a borrow, it means you have borrowed the value, and must give it back eventually, as explained in Section 3.2.

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.

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

Figure 4.2: Abstract/Concrete Environment and Types

Concrete Values

If a type is a singleton type (a type with one inhabitant), it is referred to as a value. For example $\{\text{'a}\}$ is a concrete value/type, but $\{\text{'a}, \text{'b}\}$ is not. Figure 4.2 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 4.2.

Def. 4.2.1: Drop and Concrete Operations

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

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 judgments.

Def. 4.2.2: Environment Rearrangement

$$\frac{\Omega' = \Omega, xX \ \mapsto\ \top}{\Omega \hookrightarrow \Omega'}\ \textsc{Allocation} \qquad \frac{\Omega', x \ \mapsto\ \top = \Omega}{\Omega \hookrightarrow \Omega'}\ \textsc{Deallocation}$$

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

$$\forall \diamond \in \{\, \dot{\rightsquigarrow}, \overset{(\cdot)}{\rightarrow}, \overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \dot{\leftsquigarrow} \,\} \begin{bmatrix} \textsc{Rearrange-Before} \\ \dfrac{\Omega \hookrightarrow \Omega' \quad \Omega' \vdash M \diamond v \dashv \Omega''}{\Omega \vdash M \diamond v \dashv \Omega''} \end{bmatrix}$$

$$\forall \diamond \in \{\, \dot{\rightsquigarrow}, \overset{(\cdot)}{\rightarrow}, \overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \dot{\leftsquigarrow} \,\} \begin{bmatrix} \textsc{Rearrange-After} \\ \dfrac{\Omega \vdash M \diamond v \dashv \Omega' \quad \Omega' \hookrightarrow \Omega''}{\Omega \vdash M \diamond v \dashv \Omega''} \end{bmatrix}$$

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. 4.2.3: Environment Helpers

Ω	$\Gamma \vdash \textsf{comptime}$	$\Delta \vdash \textsf{concrete}$	$\Omega \vdash \textsf{drop}$	$\Omega \sqsubseteq \Omega'$
$\emptyset$	$\dfrac{}{\emptyset \vdash \textsf{comptime}}$	$\dfrac{}{\emptyset \vdash \textsf{concrete}}$	$\dfrac{}{\emptyset \vdash \textsf{drop}}$	$\dfrac{}{\Omega \sqsubseteq \emptyset}$
$\Omega, x \mapsto t$		$\dfrac{\begin{array}{c}\textsf{concrete}\, t \\ \Omega \vdash \textsf{concrete}\end{array}}{\Omega, x \mapsto t \vdash \textsf{concrete}}$	$\dfrac{\begin{array}{c}\Omega \vdash \textsf{drop}\, t \dashv \Omega' \\ \Omega' \vdash \textsf{drop}\end{array}}{\Omega, x \mapsto t \vdash \textsf{drop}}$	$\dfrac{\begin{array}{c}x \mapsto t \in \Omega \\ \Omega \vdash t \sqsubseteq t' \\ \Omega \sqsubseteq \Omega'\end{array}}{\Omega \sqsubseteq \Omega', x \mapsto t'}$
$\Omega, X \mapsto t$	$\dfrac{\Omega \vdash \textsf{comptime}}{\Omega, X \mapsto t \vdash \textsf{comptime}}$	$\dfrac{\Omega \vdash \textsf{concrete}}{\Omega, X \mapsto t \vdash \textsf{concrete}}$	$\dfrac{\Omega \vdash \textsf{drop}}{\Omega, X \mapsto t \vdash \textsf{drop}}$	$\dfrac{\begin{array}{c}X \mapsto t \in \Omega \\ \Omega \vdash t \sqsubseteq t' \\ \Omega \sqsubseteq \Omega'\end{array}}{\Omega \sqsubseteq \Omega', X \mapsto t'}$
$\Omega, l \mapsto t$	*// 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 match. 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.

4.3 Type Operations

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

Def. 4.3.1: Type Operations

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

We also define the lesser-used type operator (as opposed to *sub*-type operator) which additionally asserts that the left hand of the colon is a concrete type; that is, a singleton.

Def. 4.3.2: Type Operator

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

4.4 Interpretation Introduction

Now we have defined the objects our interpretation operates over, this section covers the core of the system: the abstract interpretation which defines. This interpretation takes a term and outputs a type while modifying an abstract environment. This abstract interpretation is given in terms of two judgments:

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

Figure 4.3: Exhaustive table of the interpretations which make up Ochre

- $\Omega \vdash S \sqsubseteq S_t \diamond t$ where $\diamond$ is one of $\{\overset{(\cdot)}{\Rrightarrow}, \overset{\cdot}{\rightsquigarrow}\}$ which reads "when the statement S is interpreted under Ω with a bound of S_t, a value of type t is produced.". The bound S_t is a statement that must always be wider than S, which is checked and upheld throughout the interpretations.
- $\Omega \vdash M \diamond t \dashv \Omega'$ where $\diamond$ is one of $\{\overset{\cdot}{\rightsquigarrow}, \overset{(\cdot)}{\rightarrow}, \overset{(\cdot)}{\Rrightarrow}, \overset{(\cdot)}{\Lleftarrow}, \overset{(\cdot)}{\leftarrow}, \overset{\cdot}{\leftsquigarrow}\}$.
  - If $\diamond$ is a read arrow (rightward pointing), it reads "Under Ω, M is interpretted to a value of type t, and updates the environment to Ω'."
  - If $\diamond$ is a write arrow (leftward pointing), it reads "under Ω, if t is written to M, the environment is updated to Ω'".

This is analogous to a Hoare triple where Ω is the pre-condition, and Ω' is the post-condition.

We first introduce key concepts that are required to understand the definitions; then subsequent sections 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.

It is best to conceptualize the arrows as a single interpretation with many modalities. Figure 4.3 shows the different modalities and their respective arrows.

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

Ochre has two distinct sorts of term: runtime terms, and comptime (compile-time) terms.

Runtime terms must run efficiently on hardware, and are constrained to support this. This involves move semantics to enable manual memory management and allows the in-place mutation of data structures.

Comptime terms are only used to compute types, thus they are erased at compile time. Inefficient but automatic memory management strategies can be used for comptime terms, such as reference counting, because they are not used by the concrete interpretation. This removes the need for move semantics, which allows types to be used multiple times without explicit copying.

Comptime terms do not have move semantics, so they cannot have mutation[1], and thus we do not need immutable references. By not supporting mutation within comptime terms, the programmer does not have to reason about the side effects of evaluating types. The evaluation of types occurs implicitly in situations such as type checking a function call site (see $\langle$*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $MN\rangle$).

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".

The motivation behind having concrete semantics is so we can reason about the system's soundness, it is not useful for making efficient programs or otherwise implementing Ochre because hte concrete semantics as described here would not be performant.

Execution and type checking are very closely related in Ochre; execution is the precise version of type checking, where the inferred type is always a singleton type. They abstract and concrete modalities differ only in how they treat type annotations, and how they treat functions:

- $M : T$ runs to T under abstract interpretation, and M under concrete interpretation.
- MN causes the return type of the function to be executed under abstract interpretation, and the body fo the function to be executed under concrete interpretation.

This guarantees concrete interpretation will output a precise result (singleton type). 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, as enforced by type union interpretation only being defined for $\dot{\rightsquigarrow}$.

In the interpretation rules, the following definition is used occasionally to make assertions conditional:

[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.

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

Figure 4.4: Reading removes a value from the environment, whereas writing adds a value.

Def. 4.4.1: Concrete and Abstract Arrow Designations

$$\forall\diamond \in \{\overset{\cdot}{\rightsquigarrow}, \overset{\cdot}{\rightarrow}, \overset{\cdot}{\Rightarrow}, \overset{\cdot}{\Leftarrow}, \overset{\cdot}{\leftarrow}, \overset{\cdot}{\leftsquigarrow}\}\left[\frac{}{\textsf{abstract}\,\diamond}\right] \qquad \forall\diamond \in \{\rightarrow, \Rightarrow, \Leftarrow, \leftarrow\}\left[\frac{}{\textsf{concrete}\,\diamond}\right]$$

Read vs Write Modality

The read modality in Ochre evaluates a term to a value. This is analagous to reduction in the λ-calculus. The write modality writes a value to a term. The write modality allows one to write a value to a term; this brings variables into scope. Figure 4.4 illustrates the difference between the read and write modality with the example of variable interpretation. 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. 4.4.2: Read/Write Arrow Designations

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

4.5 Interpretation Judgements

This section defines the abstract and concrete semantics which define Ochre's type checking and runtime semantics respectively using the concepts introduced by Section **??**

Definitions are identified by $\langle \textit{def.} M \textit{for} \;\rightarrow\rangle$, where $\rightarrow$ 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 4.3 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

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

Figure 4.5: An example of quantification over all interpretation arrows

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 its 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, as shown in Figure 4.5.

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: $\langle \textit{def.} X \textit{for} \ \dot{\rightsquigarrow} \rangle$ exists but $\langle \textit{def.} X \textit{for} \ \dot{\rightarrow} \rangle$ does not.

Figure 4.6 shows which interpretations are defined for which constructs, and where to find its definition. References to definitions in prose link to this table for easy reference, for example: $\langle \textit{def.} \ \dot{\Rightarrow} \ \textit{for}\, M N \rangle$. Each of the following sections will define the typing judgments formally and explain their definition, as per the lookup table.

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

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

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

M	$\underset{\text{max}}{\overset{\cdot}{\Leftarrow}}$	$\underset{\text{max}}{\overset{\cdot}{\leftsquigarrow}}$
// expressions	Section 4.5.7	
$\texttt{'}a$	✓	✓
`_`	✓	✓
x	✓	✓
X		✓
$\texttt{\&}M$	✓	✓
$\texttt{\&mut}\ M$	✓	✓
$M\,\texttt{,}\,N$	✓	✓
$M : T$	✓	✓

Figure 4.6: Interpretation Definition Lookup Table

Def. 4.5.2: Variable Write Interpretations

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

The definition of the various interpretations of variables are a great demonstration of the differences between the different modalities. Compare ⟨*def.* $\overset{\cdot}{\Rightarrow}$ *for* x⟩ with ⟨*def.* $\overset{\cdot}{\rightarrow}$ *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 match) as well as runtime variables (lower match), 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	$M \overset{\cdot}{\rightsquigarrow} t$	$M \overset{(\cdot)}{\rightarrow} t$	$M \overset{(\cdot)}{\Rightarrow} t$	$M \overset{(\cdot)}{\Leftarrow} t$	$M \overset{(\cdot)}{\leftarrow} t$	$M \overset{\cdot}{\leftsquigarrow} t$
`'a`	$\forall \diamond \in \{ \overset{\cdot}{\rightsquigarrow}, \overset{(\cdot)}{\rightarrow}, \overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \overset{\cdot}{\leftsquigarrow} \}. \left[\dfrac{}{\Omega \vdash \texttt{'a} \diamond {}'a}\right]$					
`_`	$\forall \diamond \in \{ \overset{\cdot}{\rightsquigarrow}, \overset{(\cdot)}{\rightarrow}, \overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \overset{\cdot}{\leftsquigarrow} \}. \left[\dfrac{}{\Omega \vdash \texttt{\_} \diamond \top}\right]$					

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	$M \overset{(\cdot)}{\rightarrow} t$	$M \overset{(\cdot)}{\Rightarrow} t$
$*M$	$\dfrac{\Omega \vdash M \overset{(\cdot)}{\rightarrow} \mathsf{borrow}^{\mathsf{s}}\, l\, v}{\Omega \vdash *M \overset{(\cdot)}{\rightarrow} v}$	
	$\dfrac{\Omega \vdash M \overset{(\cdot)}{\rightarrow} \mathsf{borrow}^{\mathsf{m}}\, l\, v}{\Omega \vdash *M \overset{(\cdot)}{\rightarrow} v}$	$\dfrac{\begin{array}{c}\Omega \vdash M \overset{(\cdot)}{\Rightarrow} \mathsf{borrow}^{\mathsf{m}}\, l\, v \dashv \Omega' \\ \Omega' \vdash M \overset{(\cdot)}{\Leftarrow} \mathsf{borrow}^{\mathsf{m}}\, l\, \bot \dashv \Omega''\end{array}}{\Omega \vdash *M \overset{(\cdot)}{\Rightarrow} v \dashv \Omega''}$

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

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 match, 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* $M\,.\,\mathtt{0}, M\,.\,\mathtt{1}\rangle$ are good examples of this pattern in other language constructs.

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 deref-

[2] aliasing xor mutability

erence is not defined for comptime interpretations.

Def. 4.5.5: Reference Construction Interpretations

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

To construct a mutable reference (⟨*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* &mut M⟩), 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 ⟨*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* &mut M⟩ uses both move and write, it also enforces the location being referenced isn't behind an immutable reference.

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

Concrete vs abstract differences

Apart from $\langle$*def.* $\overset{(\cdot)}{\Rightarrow}$, $\overset{(\cdot)}{\Leftarrow}$ *for* $M : T\rangle$, 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 ($M \rightarrow S_t$) whereas abstraction will run to the body ($M \rightarrow S$). 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, $\Omega \vdash S \sqsubseteq S_t \Rrightarrow u$. 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 5.2.3 and 5.2.4.

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	$\mathsf{left}\, t$	$\mathsf{right}\, t$
$(t, T \to S)$	$\dfrac{}{\Omega \vdash \mathsf{left}\,(t, T \to S) = t}$	$\dfrac{\begin{array}{c}\Omega \vdash \mathsf{comptime} \dashv \Gamma \\ \Gamma \vdash T \overset{\cdot}{\leftsquigarrow} t \dashv \Gamma' \\ \Gamma' \vdash S \sqsubseteq \_ \overset{\cdot}{\rightsquigarrow} u\end{array}}{\Omega \vdash \mathsf{right}\,(t, T \to S) = u}$

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.

Def. 4.5.8: Pair Construction Interpretations

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

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.* $\overset{(\cdot)}{\Rightarrow}$, $\overset{(\cdot)}{\rightarrow}$ *for* M, N⟩, the constructed pair ($(m, \_ \rightarrow n)$) is always non-dependent. Whereas in ⟨*def.* $\overset{\cdot}{\rightsquigarrow}$ *for* M, N⟩ the constructed pair can be dependent ($(t, T' \rightarrow S)$). 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.* $\overset{\cdot}{\rightsquigarrow}$ *for* M, N⟩ this is put in the environment for later re-use.

Def. 4.5.9: Pair Elimination Interpretations

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

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* $M\,.\,\mathsf{0}$⟩ and ⟨*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $M\,.\,\mathsf{1}$⟩ replacing the pair with a non-dependent pair ($(\top, \_ \rightarrow t_1)$ and $(t_0, \_ \rightarrow \top)$ respectively).

Def. 4.5.10: Pair Write Interpretations

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

Destructuring is done via $\langle$*def.* $\overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \overset{\cdot}{\leftsquigarrow}$ *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	$M \mathrel{\dot{\rightsquigarrow}} t$	$M \overset{(\cdot)}{\rightarrow} t$	$M \overset{(\cdot)}{\Rightarrow} t$	$M \overset{(\cdot)}{\Leftarrow} t$	$M \overset{(\cdot)}{\leftarrow} t$	$M \mathrel{\dot{\leftsquigarrow}} t$
`M:T`	$\forall\diamond \in \{\dot{\rightsquigarrow}, \dot{\rightarrow}, \dot{\Rightarrow}\}. \left[\dfrac{\begin{array}{c}\Omega \vdash M \diamond m \dashv \Omega' \\ \Omega' \vdash T \mathrel{\dot{\rightsquigarrow}} t \\ \Omega' \vdash m \sqsubseteq t\end{array}}{\Omega \vdash M : T \diamond t \dashv \Omega'}\right]$			$\forall\diamond \in \{\dot{\leftsquigarrow}, \dot{\leftarrow}, \dot{\Leftarrow}\}. \left[\dfrac{\begin{array}{c}\Omega \vdash M \diamond m \dashv \Omega' \\ \Omega' \vdash T \mathrel{\dot{\rightsquigarrow}} t \\ \Omega' \vdash m \sqsubseteq t\end{array}}{\Omega \vdash M : T \diamond m \dashv \Omega'}\right]$		
	$\forall\diamond \in \{\rightarrow, \Rightarrow, \Leftarrow, \leftarrow\}. \left[\dfrac{\Omega \vdash M \diamond m \dashv \Omega' \quad \text{// ignores type annotations}}{\Omega \vdash M : T \diamond m \dashv \Omega'}\right]$					

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	$T \mathrel{\dot{\rightsquigarrow}} t$
$T \mid U$	$\dfrac{\begin{array}{c}\Omega \vdash T \mathrel{\dot{\rightsquigarrow}} t \dashv \Omega' \\ \Omega' \vdash U \mathrel{\dot{\rightsquigarrow}} u \dashv \Omega'' \\ \Omega'' \vdash t \sqcup u = t'\end{array}}{\Omega \vdash T \mid U \mathrel{\dot{\rightsquigarrow}} t' \dashv \Omega''}$
v	$\dfrac{}{\Omega \vdash v \mathrel{\dot{\rightsquigarrow}} v}$

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

All expressions are themselves statements, when this is the match, 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	$\Omega \vdash S \sqsubseteq S_t \overset{(\cdot)}{\Rightarrow} v$	$\Omega \vdash S \overset{\cdot}{\rightsquigarrow} t$
$M{=}N\,;S$	$\forall \diamond \in \{\overset{\cdot}{\rightsquigarrow}, \overset{(\cdot)}{\Rightarrow}\}.\ \left[\dfrac{\begin{array}{cc} \multicolumn{2}{c}{\text{// \textit{peform assignment}}} \\ \multicolumn{2}{c}{\Omega \vdash N \diamond v \dashv \Omega'} \\ \multicolumn{2}{c}{\Omega' \vdash M\ (\mathsf{flip}\diamond)\ v \dashv \Omega''} \\ \multicolumn{2}{c}{\text{// \textit{assert: compatible LHS syntax}}} \\ \mathsf{runtime}\ M & \mathsf{if\ concrete}\diamond \\ \mathsf{comptime}\ M & \mathsf{if\ squiggily}\diamond \\ \multicolumn{2}{c}{\text{// \textit{assert: bound can only tighten}}} \\ \Omega \vdash S_t \overset{\cdot}{\rightsquigarrow} t & \mathsf{if\ abstract}\diamond \\ \Omega'' \vdash S_t \overset{\cdot}{\rightsquigarrow} t' & \mathsf{if\ abstract}\diamond \\ \Omega'' \vdash t' \sqsubseteq t & \mathsf{if\ abstract}\diamond \\ \multicolumn{2}{c}{\text{// \textit{execute remaining computation}}} \\ \multicolumn{2}{c}{\Omega'' \vdash S \sqsubseteq S_t \overset{(\cdot)}{\Rightarrow} w} \end{array}}{\Omega \vdash M{=}N\,;S \sqsubseteq S_t \overset{(\cdot)}{\Rightarrow} w} \right]$	$\dfrac{\begin{array}{c} \Omega \vdash N \overset{\cdot}{\rightsquigarrow} v \dashv \Omega' \\ \Omega' \vdash M \overset{\cdot}{\leftsquigarrow} v \dashv \Omega'' \\ \Omega'' \vdash S_t \overset{\cdot}{\rightsquigarrow} w \end{array}}{\Omega \vdash M{=}N\,;S_t \overset{\cdot}{\rightsquigarrow} w}$

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 **??**. 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 5.2.3.

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

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

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	$M \underset{\mathsf{max}}{\overset{\cdot}{\Leftarrow}} t$	$T \underset{\mathsf{max}}{\overset{\cdot}{\leftsquigarrow}}$
$\text{’}a$	$\forall\diamond \in \{\underset{\mathsf{max}}{\overset{\cdot}{\Leftarrow}}, \underset{\mathsf{max}}{\overset{\cdot}{\leftsquigarrow}}\}. \left[\dfrac{}{\Omega \vdash \text{’}a \diamond \text{’a}}\right]$	
$\_$	$\forall\diamond \in \{\underset{\mathsf{max}}{\overset{\cdot}{\Leftarrow}}, \underset{\mathsf{max}}{\overset{\cdot}{\leftsquigarrow}}\}. \left[\dfrac{}{\Omega \vdash \_ \diamond \top}\right]$	
x	$\forall\diamond \in \{\underset{\mathsf{max}}{\overset{\cdot}{\Leftarrow}}, \underset{\mathsf{max}}{\overset{\cdot}{\leftsquigarrow}}\}. \left[\dfrac{}{\Omega \vdash x \diamond \top}\right]$	
X	$\dfrac{}{\Omega \vdash X \diamond \top}$	
$M\,,N$	$\forall\diamond \in \{\underset{\mathsf{max}}{\overset{\cdot}{\Leftarrow}}, \underset{\mathsf{max}}{\overset{\cdot}{\leftsquigarrow}}\}. \left[\dfrac{\begin{array}{c}\Omega \vdash M \diamond m \dashv \Omega' \\ \Omega' \vdash N \diamond n \dashv \Omega''\end{array}}{\Omega \vdash M, N \diamond (m, \_ \to n) \vdash \Omega''}\right]$	
$M : T$	$\forall\diamond \in \{\overset{\cdot}{\Leftarrow}, \overset{\cdot}{\leftsquigarrow}\}. \left[\dfrac{\begin{array}{c}\Omega \vdash T \overset{\cdot}{\rightsquigarrow} t \dashv \Omega' \\ \Omega' \vdash M \diamond t \dashv \Omega''\end{array}}{\Omega \vdash M : T \underset{\mathsf{max}}{\diamond} t \vdash \Omega''}\right]$	

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

4.6 Design Decisions

This section covers the broader design decisions of Ochre that are not tied to a particular language construct, a the motivation behind their inclusion or exclusion.

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.

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.

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. I leave this to future work

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 match 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. 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.

2. It would complicate the type system and syntax further.

3. I predict that it will be idiomatic in Ochre to separate data structures from proofs about their structure. If this is the match, 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.

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.3.

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 Specific Program Checks

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

$$\dfrac{\dfrac{AA \qquad BB}{A} \qquad \dfrac{BA \qquad BB}{B}}{C}
\qquad\qquad
\begin{array}{lll}
C & & \\
 & A & \\
 & & AA \\
 & & AB \\
 & B & \\
 & & BA \\
 & & BB
\end{array}$$

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

```
1                                 // $\Omega_0 = \emptyset$
2   pair = ('world, 'hello); // $\Omega_1 = \emptyset, \texttt{pair} \mapsto (\text{'world}, \_ \to \text{'hello})$
3   temp = pair.0;               // $\Omega_2 = \emptyset, \texttt{pair} \mapsto (\top, \_ \to \text{'hello}), \texttt{temp} \mapsto \text{'world}$
4   pair.0 = pair.1;             // $\Omega_3 = \emptyset, \texttt{pair} \mapsto (\text{'hello}, \_ \to \top), \texttt{temp} \mapsto \text{'world}$
5   pair.1 = temp;               // $\Omega_4 = \emptyset, \texttt{pair} \mapsto (\text{'hello}, \_ \to \text{'world}), \texttt{temp} \mapsto \top$
6
7   pair // $(\text{'hello}, \_ \to \text{'world})$
```

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

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:

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

The program is shown to have type $(\text{'hello}, \_ \to \text{'world})$ if we can find a derivation of the

```
    Repetative = ('repeated, Repetative); // ∅, Repetative ↦ ('repeated, _ → Repetative)
    Repetative.1.1.0 // ”repeated
```

Listing 19: An example of a comptime pair being defined in terms of itself.

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.4.1.

5.1.2 Recursion

When defining a variable, the variable can use itself, since everything initially maps to top. This allows for the definition of recursive structures in comptime terms, as shown in Listing 19. This example demonstrates the power of storing syntax in the type for dependent pairs. When type checking `('repeated, Repetative)`, we only had the information that `Repetative` $\mapsto \top$, but this didn't cause any problems because we do not need any typing about variable information to be able to put it in a pair.

Then, the assignment narrows the type of `Repetative` from $\top$ down to ('repeated, _ → `Repetative`). This is sound due to the monotonicity of typing judgments (Property 5.2.2), which will be covered more in Section 5.2, but for now I give the following intuition: ('repeated, _ → `Repetative`) encodes more information than $\top$, so if we can conclude something with only the information $\top$, we can certainly include it with ('repeated, _ → `Repetative`).

Accessing the right side of the pair with `Repetative.1` then executes this syntax stored in the environment under this new, narrower environment, which gives a pair. A derivation of this is given in Appendix A.4.2.

5.1.3 Mutating Dependent Pairs

This section shows how mutation interacts with dependent types through an example where the correctness of a mutation depends on a dependent pair.

Listing 20 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.

```
1                                  // $\Omega_0 = \emptyset$
2    Same = ('a | 'b, L -> L); // $\Omega_1 = \emptyset, \texttt{Same} \mapsto (\{\text{'a}, \text{'b}\}, L \to L)$
3
4    overwrite = (p: &mut Same) -> 'unit {
5                          // $\Omega_{10} = \Omega_1, \text{p} \mapsto \text{borrow}^{\text{m}}\, l\, (\{\text{'a}, \text{'b}\}, L \to L), l \mapsto (\{\text{'a}, \text{'b}\}, L \to L)$
6      (*p).0 = 'a; // $\Omega_{11} = \Omega_1, \text{p} \mapsto \text{borrow}^{\text{m}}\, l\, (\text{'a}, \_ \to \{\text{'a}, \text{'b}\}), l \mapsto (\{\text{'a}, \text{'b}\}, L \to L)$
7      (*p).1 = 'a; // $\Omega_{12} = \Omega_1, \text{p} \mapsto \text{borrow}^{\text{m}}\, l\, (\text{'a}, \_ \to \text{'a}), l \mapsto (\{\text{'a}, \text{'b}\}, L \to L)$
8      'unit          // $\Omega_{12} \vdash \text{drop}$
9    }                             // $\Omega_2 = \Omega_1, \texttt{overwrite} \mapsto \texttt{(p: \&mut Same)} \to \texttt{'unit}$
10
11   pair = ('b, 'b);             // $\Omega_3 = \Omega_2, \texttt{pair} \mapsto (\text{'b}, \_ \to \text{'b})$
12   overwrite(&mut pair);        // $\Omega_4 = \Omega_2, \texttt{pair} \mapsto (\{\text{'a}, \text{'b}\}, \text{L} \to \text{L})$
13
14   pair // $(\{\text{'a}, \text{'b}\}, L \to L)$
```

Listing 20: A program which mutates a dependent pair correctly.

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 $(\text{'b}, \text{'b})$, which is why $\texttt{pair} \mapsto (\{\text{'a}, \text{'b}\}, L \to L)$ in the environment instead of the more precise $\texttt{pair} \mapsto (\text{'a}, \_ \to \text{'a})$.

At the end of the derivation for the `overwrite` function, the mutation is checked to be correct, triggered by the environment being cleaned up. Dropping the reference causes the value in the borrow $(\text{'a}, \_ \to \text{'a})$ to be checked against its loan restriction, which is $(\{\text{'a}, \text{'b}\}, \text{L} \to \text{L})$. The sub-derivation which performs this checking is shown in by Figure 5.2.

Appendix A.4.3 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 21 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 match, it is not.

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

$$
\begin{array}{ll}
\Omega_1 \vdash p_3 \sqsubseteq p_0 & \Omega_1 = \emptyset, \mathtt{Same} \mapsto (\{\text{'a}, \text{'b}\}, L \to L) \\
\quad \Omega_1 \vdash \text{'a} \sqsubseteq \{\text{'a}, \text{'b}\} & p_0 = (\{\text{'a}, \text{'b}\}, L \to L) \\
\quad\quad \{\text{'a}\} \subseteq \{\text{'a}, \text{'b}\} & p_3 = (\text{'a}, \_ \to \text{'a}) \\
\quad \Omega_1 \vdash \mathsf{comptime} \dashv \Gamma_2 & \Gamma_2 = \Omega_1, \mathtt{L} \mapsto \top \\
\quad \Gamma_2 \vdash \mathtt{L} \overset{\cdot}{\leftsquigarrow} \text{'a} \dashv \Gamma_2' & \\
\quad\quad \Gamma_2' = \Gamma_2 \left[\dfrac{\mathtt{L} \mapsto \text{'a}}{\mathtt{L} \mapsto \top}\right] & \Gamma_2' = \Omega_1, \mathtt{L} \mapsto \text{'a} \\
\quad \Gamma_2 \vdash \_ \overset{\cdot}{\leftsquigarrow} \text{'a} & \\
\quad \Gamma_2' \vdash \text{'a} \sqsubseteq \mathtt{L} \overset{\cdot}{\rightsquigarrow} \text{'a} & \\
\quad\quad \Gamma_2' \vdash \text{'a} \overset{\cdot}{\rightsquigarrow} \text{'a} & \\
\quad\quad \Gamma_2' \vdash L \overset{\cdot}{\rightsquigarrow} \text{'a} & \\
\quad\quad\quad \mathtt{L} \mapsto \text{'a} \in \Gamma_2' & \\
\quad\quad \Gamma_2' \vdash \text{'a} \sqsubseteq \text{'a} & \\
\quad\quad\quad \{\text{'a}\} \subseteq \{\text{'a}\} &
\end{array}
$$

Figure 5.2: Derivation which checks the post-mutation pair against the `overwrite` function's type signature from Listing 20. See Appendix A.4.3 for surrounding derivation.

```
incorrect_overwrite = (p: &mut Same) -> 'unit {
                      // $\Omega_{20} = \Omega_1, \mathrm{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, (\{\text{'a}, \text{'b}\}, L \to L), l \mapsto (\{\text{'a}, \text{'b}\}, L \to L)$
  p.0 = 'a;           // $\Omega_{21} = \Omega_1, \mathrm{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, (\text{'a}, \_ \to \{\text{'a}, \text{'b}\}), l \mapsto (\{\text{'a}, \text{'b}\}, L \to L)$
  p.1 = 'b;           // $\Omega_{21} = \Omega_1, \mathrm{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, (\text{'a}, \_ \to \text{'b}), l \mapsto (\{\text{'a}, \text{'b}\}, L \to L)$
                      // $\times$, cannot drop p, loan restriction mismatch
}
```

Listing 21: An incorrect program, which does not leave the pair in a valid state

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

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

```
Swap = T -> (x: &mut T, y: &mut T) -> 'unit {
                                      // $\Omega_0 = \emptyset, l_x \mapsto \top, l_y \mapsto \top$
                                      // $\Omega_1 = \Omega_0, \mathrm{x} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l_x\, \top, \mathrm{y} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l_y\, \top$
  (*x, *y) = (*y, *x); // $\Omega_1 = \Omega_0, \mathrm{x} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l_x\, \top, \mathrm{y} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l_y\, \top$
  'unit
}
```

Listing 22: Polymorphic Swap

5.1.4 Polymorphic Swap Function

Listing 22 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 type you want to instantiate the function with. 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 22, 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].

5.1.5 Peano Numbers and Add

Sections 5.1.1 and 5.1.3 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 24 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, or the successor of another number. If it is the successor of another number, the right of the

```
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 23: 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 24: More efficient definition of `add`.

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 match, 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.

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 ($\emptyset$)

$$\begin{array}{r} \emptyset \vdash S \sqsubseteq \_ \;\dot{\Rightarrow}\; t \\ \emptyset \vdash S \sqsubseteq \_ \Rightarrow v \end{array} \text{ implies } \emptyset \vdash v : t$$

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 direct or indirect requirement for proving the central soundness property (5.2.1).

Our soundness property assumes both $\dot{\Rightarrow}$ 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 $\dot{\Rightarrow}$ 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

$$\begin{array}{r} \Omega \vdash S \sqsubseteq \_ \;\dot{\Rightarrow}\; t \\ \Delta \vdash S \sqsubseteq \_ \Rightarrow v \\ \Delta : \Omega \end{array} \text{ implies } \Omega \vdash v : t$$

Abstract and concrete interpretation only differ in three language constructs: function application $\langle$*def.* $\overset{(\cdot)}{\Rightarrow}$ *for* $M N\rangle$, 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.5.1 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 $\{\rightarrow, \Rightarrow\}$.

$$\begin{gathered}\Omega \vdash M \mathbin{\dot{\diamond}} t \dashv \Omega' \\ \Delta \vdash N \diamond v \dashv \Delta' \\ \Delta : \Omega\end{gathered}$$

implies

$$\begin{gathered}\Delta' : \Omega' \\ \Omega' \vdash v : t\end{gathered}$$

Prop. 5.2.4: Expression Write Soundness

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

$$\begin{gathered}\Omega \vdash M \mathbin{\dot{\diamond}} t \dashv \Omega' \\ \Delta \vdash M \diamond v \dashv \Delta' \\ \Delta : \Omega \\ \Omega \vdash v : t\end{gathered}$$

implies

$$\Delta' : \Omega'$$

Expression interpretation soundness also includes the output *environments* being well-typed.

The type on the right-hand side of the arrow, usually t or v, acts as an input for write interpretations. This means that instead of write soundness concluding $v : t$, it takes it in as a premise. This reflects the fact that write interpretations do not produce a concrete value, they require one.

Expression soundness is required directly in every match of our central soundness proof.

5.2.2 Monotonicity

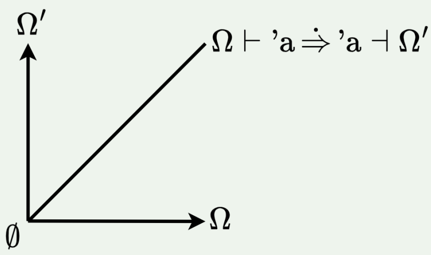
In this context, monotonicity means "if the input is narrowed, the output is narrowed". It is useful to think of the width of a type or an environment in terms of information: concluding a value has a narrow type gives you more information than concluding it has a wide type.

This a crucial property for all abstract and concrete semantics to have if they are to be used in abstract interpretation. We do not take advantage of results from abstract interpretion, but these properties happen to be required in the proofs nonetheless.

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 $\langle$Allocation$\rangle$). $\top$ and $\emptyset$ are the widest types and environments respectively.

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), similar to how you might label a line on a graph $y = 5x + 2$ in mathematics. Lines pass through the origin iff their statment is valid in an empty environment.

Example


Ω'
$\Omega \vdash \text{'a} \mathrel{\dot{\Rightarrow}} \text{'a} \dashv \Omega'$
$\emptyset$
Ω

Atoms leave Ω unchanged

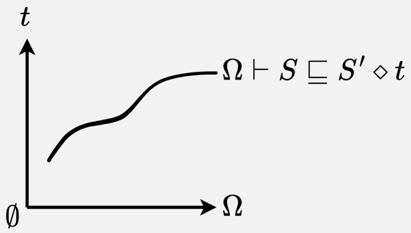
Prop. 5.2.5: Statement Interpretation Monotonicity

for all $\diamond$ in $\{\, \dot{\rightsquigarrow}\,, \overset{(\cdot)}{\Rrightarrow} \,\}$.

$$\Omega_0 \vdash S \sqsubseteq S' \diamond t_0$$
$$\Omega_1 \sqsubseteq \Omega_0$$

implies there exists t_1 s.t.

$$\Omega_1 \vdash S \sqsubseteq S' \diamond t_1$$
$$\Omega_1 \vdash t_1 \sqsubseteq t_0$$



Always up and right

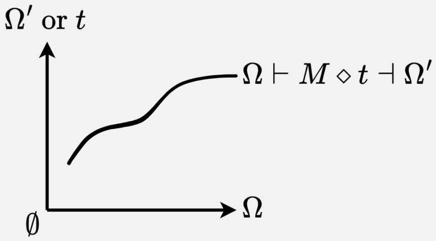
Prop. 5.2.6: Expression Read Monotonicity

for all $\diamond$ in $\{\, \overset{(\cdot)}{\rightarrow}\,, \overset{(\cdot)}{\Rrightarrow} \,\}$.

$$\Omega_0' \sqsubseteq \Omega_0$$
$$\Omega_0 \vdash M \diamond t \dashv \Omega_1$$

implies there exists a Ω_1' and t' s.t.

$$\Omega_1' \sqsubseteq \Omega_1$$
$$\Omega_1' \vdash t' \sqsubseteq t$$
$$\Omega_0' \vdash M \diamond t' \dashv \Omega_1'$$



Always up and right

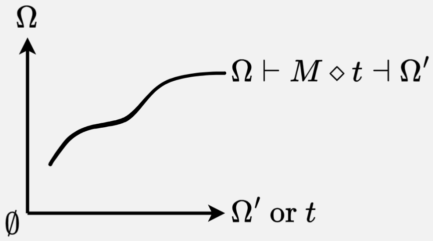
Prop. 5.2.7: Expression Write Monotonicity

for all $\diamond$ in $\{\, \overset{(\cdot)}{\Lleftarrow}\,, \overset{(\cdot)}{\leftarrow} \,\}$.

$$\Omega_0' \sqsubseteq \Omega_0$$
$$\Omega_0' \vdash t' \sqsubseteq t$$
$$\Omega_0 \vdash M \diamond t \dashv \Omega_1$$

there exists Ω_1' s.t.

$$\Omega_1' \sqsubseteq \Omega_1$$
$$\Omega_0' \vdash M \diamond t' \dashv \Omega_1'$$



Always up and right

5.2.3 Statement Bounds

Statement evaluation occurs with a *bound*. At all future program points, the bound statement must be wider than the statement on the left. This is how function bodies are type checked against their type.

This property is upheld by each of the statement rules. When doing variable assignment $\langle \textit{def.}\ \dot{\Rightarrow}\ \textit{for}\ M = N; S\rangle$ we have to check that each assignment does not break our statement bounds.

Prop. 5.2.8: Statement Bounds Respected

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

$$\Omega \vdash S \sqsubseteq S' \diamond t$$

there exists a u s.t.

$$\Omega \vdash S \sqsubseteq \_ \diamond u$$
$$\Omega \vdash S' \sqsubseteq \_ \dot{\rightsquigarrow} t$$
$$\Omega \vdash u \sqsubseteq t$$

5.2.4 Subtyping Preservation

We introduce subtyping between terms, like so;

Def. 5.2.1: Expression Subtyping

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

Def. 5.2.2: Statement Subtyping

$$\frac{\Omega \vdash S \sqsubseteq S_t\ \dot{\Rightarrow}\ t}{\Omega \vdash S \sqsubseteq S_t}$$

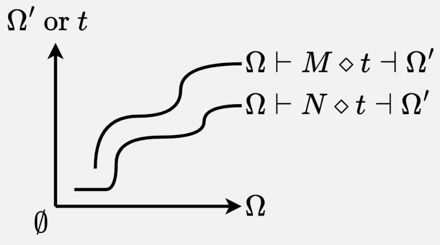
Prop. 5.2.9: Expression Subtyping Preservation

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

$$\Omega_0 \vdash M \sqsubseteq_{\diamond} N$$
$$\Omega_1 \sqsubseteq \Omega_0$$

implies

$$\Omega_1 \vdash M \sqsubseteq_{\diamond} N$$


Ω' or t
$\Omega \vdash M \diamond t \dashv \Omega'$
$\Omega \vdash N \diamond t \dashv \Omega'$
$\emptyset$
Ω

Lines never cross

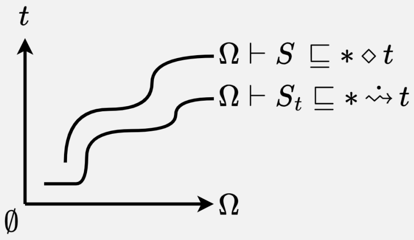
Prop. 5.2.10: Statement Subtyping Preservation

for all $\diamond$ in $\{\Rightarrow, \dot{\rightsquigarrow}\}$.

$$\Omega_0 \vdash S \sqsubseteq_\diamond S_t$$
$$\Omega_1 \sqsubseteq \Omega_0$$

implies

$$\Omega_1 \vdash S \sqsubseteq_\diamond S_t$$


t
$\Omega \vdash S \sqsubseteq * \diamond t$
$\Omega \vdash S_t \sqsubseteq * \dot{\rightsquigarrow} t$
Ω
$\emptyset$

Lines never cross

Statement Bound Motivation

Term subtyping preservation means that any relationship we conclude between two terms under one environment, can still be used under all environments which contain more information. This is useful in proofs when we do a series of operations on the environment which narrow it, and we want to re-use an earlier conclusion.

Expression subtyping and statement subtyping are defined in a very different way (5.2.4 vs 5.2.4). This is because narrowing the input environment for two statements can cause their output in different ways. For example, take the following example

```
id = (b: 'true | 'false) -> b {
  match b {
    'true => 'false,
    'false => 'true
  }
}
```

The function body, when executed under x $\mapsto$ $\{'true,' false\}$, evaluates to x $\mapsto$ $\{'true,' false\}$ reflecting the fact it could be either true or false, and so does the body. However, if we then narrow this input down to x $\mapsto$ $'true$, the return type and the function body narrow in different ways, one to 'true, the other to 'false.

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.

Def. 5.2.3: Expression-Like Statement Subtyping

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

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

$$\begin{matrix}\Omega \vdash S_a \hat{\sqsubseteq} S_b \\ \Omega' \sqsubseteq \Omega\end{matrix} \text{ does not imply } \Omega' \vdash S_a \hat{\sqsubseteq} S_b$$

5.2.5 Information Gain/Loss

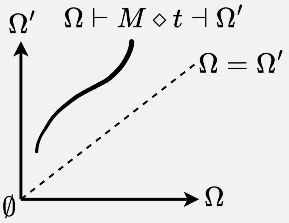
The narrower an environment is, the more information it contains. These two theorems state that reading from the environment always *uses* information and writing to the environment *adds* information.

Evaluating an expression changes how much information there is in the environment. Moving a value from a term removes that information from the environment, moves always widen environments.

Prop. 5.2.12: Move-Interpretation Widens

for all $\diamond$ in $\{\dot{\Rightarrow}\}$

$\Omega \vdash M \diamond t \dashv \Omega'$ implies $\Omega \sqsubseteq \Omega'$



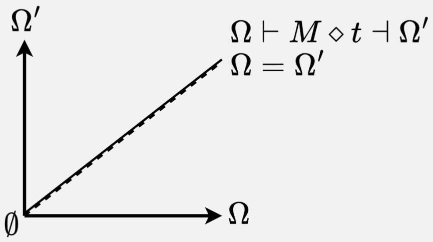
Always above the dotted line

Non-destructive reads never widen or narrow the environment, although they can add loan identifiers so we cannot say they are equal.

Prop. 5.2.13: Read-Interpretation Does Nothing

for all $\diamond$ in $\{\dot{\rightarrow}, \dot{\rightsquigarrow}\}$

$\Omega \vdash M \diamond t \dashv \Omega'$ implies $\begin{matrix}\Omega \sqsubseteq \Omega' \\ \Omega' \sqsubseteq \Omega\end{matrix}$



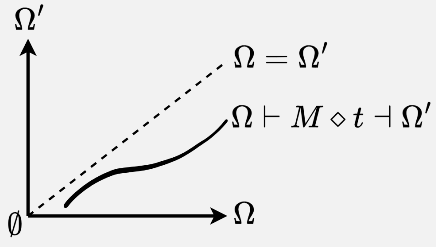
Lines always together

Write interpretation adds information to the environment, so it always narrows it.

Prop. 5.2.14: Write-Interpretation Narrows

for all $\diamond$ in $\{\dot{\Leftarrow}, \dot{\leftarrow}, \dot{\leftsquigarrow}\}$

$\Omega \vdash M \diamond t \dashv \Omega'$ implies $\Omega' \sqsubseteq \Omega$



Always below dotted line

Prop. 5.2.15: 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. 5.2.16: Environment Union is **not** Precise

$\Omega \sqsubseteq \Omega' \sqcup \Omega''$ **not** if and only if $\Omega \sqsubseteq \Omega'$ or $\Omega \sqsubseteq \Omega''$

Chapter 6

Evaluation

As stated in our introduction:

> The goal of Ochre is to solve the design problems which are stopping theorem proving from being brought to the systems programming masses. We do this integrating dependent types into the type system, and broader design of, of a programming language in an unusually ergonomic and performance-compatible way, much like Rust did with memory safety.

This section discusses how successful we were at achieving this goal.

I consider the power of the type system presented to be a great success, we can type programs with mutability and dependent types tightly interwoven, as shown in Section 5.1.3. The more interesting question is do we offer significant ergonomic improvements over the encumbant low level systems languages with dependent types.

6.1 Ergonomics

We evaluate Ochre's ergonomics by the implementation of various programs in Ochre and three other languages:

- Low*, because it is battle tested, and has been used for large-scale verification projects [Zinzindohoué et al., 2017, Ramananandro et al., Tao et al.]. Low* also has state of the art ergonomics for a language of its featureset, as you will see with its comparison with ATS.
- ATS, because it was one of the first and is the most mature language which offers dependent typing of low level systems code.
- Rust, as a baseline, for what ergonomics can look like without dependent types at all.

Low* and ATS require imports to work, which have been omitted from the code examples.

```
Same = ('true | 'false, L -> L);

overwrite = (p: &mut Same) -> 'unit {
  *p = ('true, 'true);
  'unit
};

pair = ('false, 'false);
overwrite(&mut pair);
```

Listing 25: Ochre implementation of the pair mutation program.

```
1   type Same = | MkSame: b:bool -> b2:bool{b == b2} -> Same
2
3   val overwrite: p:ptr Same -> Stack unit
4   let overwrite p =
5     p := Same false false
6
7   let main () : Stack unit =
8     let pair = alloc (MkSame true true) in
9     overwrite s;
10    ()
```

Listing 26: Low* implementation of the pair mutation program.

6.1.1 Mutating Dependent Pairs

We compare the implementation of a modified version of the *pair mutation* program from Section 5.1.3 in Listings 25, 26, 27, and 28. Originally in Section 5.1.3, our Ochre program mutated each element of the pair on a separate line, for now I have modified the program to mutate both at once because neither ATS nor Low* support *temporarily* breaking type constraints like this.

Comparison with Low* - In Low*, mutation is handled through the `Stack` monad, this allows Low* to be a pure functional language at the surface level, which avoids the issue of combining mutability with dependent types. Subjectively, this is a very elegant approach to combining mutability and dependent types which Low* does well, but it does require always being within the `Stack` monad. While monads have very elegant use cases, requiring them for absolutely everything is frustrating, and certainly would not be popular with low-level systems programmers. It also means there is a distinction between your surface-level variables like `pair`, and the stack allocations they reference, which requires explicit stack allocation (as shown on line 8).

Comparison with ATS - In ATS, most runtime values have a type-level counterpart. You use the runtime value for execution and the type-level counterpart for type signatures. In Listing 27, line 10 constructs the runtime pointer, then ATS implicitly creates a compile-time counterpart called the pointers *address* when `p` is passed to `overwrite`.

```
typedef Same = (bool, bool) where { (x, y) => x == y }

fun overwrite {l:addr} (p: &Same @ l): void = let
in
  p := (false, false)
end

implement main0() = let
  var s: Same = (true, true)
  val p = addr@ s
in
  overwrite(p);
end
```

Listing 27: ATS implementation of the pair mutation program.

```
type Same = (bool, bool);

fn overwrite(p: &mut Same) {
  *p = (true, true);
}

fn main() {
  let pair = (false, false);
  overwrite(&mut pair);
}
```

Listing 28: Rust implementation of the pair mutation program.

Having both a pointer and a type-level address for that pointer clutters the type signature, although does not add too much complexity to the programmer's mental model. This separation between type level objects and value level objects applies to everything in ATS: if a function takes an integer, and the return type depends on that integer, you must have a type level and a value level version of that integer.

Comparison with Rust - Apart from missing the dependent type, and needing a main function, the Rust implementation is almost identical to Ochre's. Apart from Ochre's definition of `Same`, the presence of dependent types do not complicate the Ochre implementation at all.

6.1.2 Mutating Dependent Pairs Field-By-Field

We now consider the pair mutation program as presented originally in Section 5.1.3. Ergonomics suffer immensely in these examples for Low* and ATS, because they do not support breaking the type constraint of a dependent pair temporarily[1]

This a bit of a strawman comparison because in reality, you would rework your codebase to not require this feature, but I think it serves as a good demonstration of the benefits of the type system remaining flexible and safe. It also demonstrates the problems caused when you stray out of a language's typically supported feature set, which people do in pursuit of performance, and will have to do less in Ochre than Low* and ATS, due to differences like this.

Listings 29, 30, 31, and 32 show the program in Ochre, Low*, ATS, and Rust respectively. The reason the Low* and ATS programs look bad is that in order to mutate the pair in place one field at a time, you must use escape hatches to tell the type system to not check whether the mutations you are doing are type-safe, which as well as being error-prone and dangerous, is very labor intensive.

6.2 Performance

While it is impossible to empirically test the performance Ochre enables, due to not having an implementation, we can reason about how compatible it is with performance-relevant language features. To summarise: Ochre as presented would not be performant, but unlike other languages like Java or Haskell, it does not have features that hold its performance back. Making Ochre performant would just be a matter of engineering, not new research.

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.

[1] To motivate this feature a bit: you may want to do in the implementation of data structures, where you have updated parts of it but not all. For example: when resizing a Rust-like vector.

```
Same = ('true | 'false, L -> L);

overwrite = (p: &mut Same) -> 'unit {
  // Does not need to use any escape hatches
  *p.0 = 'true;
  // pair can be used in middle-state
  *p.1 = 'true;
  'unit
};

pair = ('false, 'false);
overwrite(&mut pair);
```

Listing 29: Ochre implementation of the original pair mutation program.

```
type Same = | MkSame: b:bool -> b2:bool{b == b2} -> Same

val overwrite: p:loc Same -> Stack unit
let overwrite p =
  // Temporarily cast away the type invariant
  let ptr = p in

  // Unsafe operations to modify the pair
  let _ = (ptr as ref (bool, bool)) in
  (let r = (ptr as ref (bool, bool)) in r := (false, snd (!r))); // First update
  // pair can be used here
  (let r = (ptr as ref (bool, bool)) in r := (false, false)); // Second update

  ()

let main () : Stack unit =
  let s = alloc (MkSame true true) in
  overwrite s;
  ()
```

Listing 30: Low* implementation of the original pair mutation program.

```
typedef Same = (bool, bool) where { (x, y) => x == y }

fun overwrite {l:addr} (p: &Same @ l): void = let
  val (b1, b2) = !p
  val p2 = cast{(bool, bool) @ l} p // Temporary unsafe cast to bypass the type cons
in
  p2 := (false, b2)
  // pair can be used here
  p2 := (false, false)
end

implement main0() = let
  var s: Same = (true, true)
  val p = addr@ s
in
  overwrite(p);
end
```

Listing 31: ATS implementation of the original pair mutation program.

```
type Same = (bool, bool);

fn overwrite(p: &mut Same) {
  p.0 = true;
  // pair can be used here
  p.1 = true;
}

fn main() {
  let pair = (false, false);
  overwrite(&mut pair);
}
```

Listing 32: Rust implementation of the original pair mutation program.

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.3 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.

6.3 Reusability

This research is intended to be used in the future to form an implementation, and as such it must provide usable foundations to work upon. While Ochre the language is pleasant to work with, the type system as presented is extremely complex and hard to work with. It contains many features that do not directly work towards making a dependently typed systems language, such as pattern matching and the concept of writing to syntax generally. These features make the language more usable, but restrict the re-use of the underlying theory, and certainly make reasoning about its properties harder.

Chapter 7

Conclusion

To summarize, in Chapter 4 we presented the formal semantics for Ochre, a dependently typed systems language. Then in Chapter 5 we show it accepts correct programs and rejects incorrect ones, and in Chapter 6 we demonstrate that it does so with superior ergonomics when compared to languages with comparable feature sets, and even compares well to Rust.

With this work undertaken, what is next?

7.1 Future Work

The two most important pieces of future work are proving the soundness of the presented system and implementing Ochre so that it may be used for useful theorem proving in production systems. I believe the best route to achieving these goals is to follow the following roadmap:

7.1.1 Reduce Feature Set, Increase Rigor

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 a soundness proof is by removing features and progressively building them back. I suggest two such feature sets:

Och

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

Och would have the following syntax and environment:

$$
\begin{array}{lll}
M, N ::= & & \textit{// term} \\
 & x \mid y \mid z & \textit{// runtime variable} \\
 & M N & \textit{// application} \\
 & x : M \ \texttt{->} \ N & \textit{// abstraction} \\
 & \texttt{\_} & \textit{// uninitialised} \\
 & M \ \texttt{|} \ N & \textit{// type union} \\
 & M : N & \textit{// type constraint}
\end{array}
\qquad
\begin{array}{lll}
\Omega \ ::= & & \textit{// environment} \\
 & \emptyset & \textit{// empty env.} \\
 & \Omega, x \mapsto v & \textit{// runtime variable} \\
 & \Omega, X \mapsto v & \textit{// comptime variable} \\
 & & \\
t, u ::= & & \textit{// type/value} \\
 & ((x : T) \to U) & \textit{// function} \\
 & \top & \textit{// top}
\end{array}
$$

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.

Removing mutation and move semantics would also make it sound for functions to capture scope since that scope is guaranteed to never change. This would allow you to use church encodings for data structures and avoid the need for pairs and atoms. Variable assignment could be modelled with function application.

Och, we could focus on the core principle of terms being their own types, and produce a soundness proof and implementation. Once this foundation has been laid, we can introduce the features back gradually. We could also focus on features whose omission hurt Ochre, such as variables in the environment not being able to depend on each other.

Ochr

Once Och is established, the concepts of ownership and *maybe* borrowing should be added to make Ochr. In Ochr, using a value consumes the value, to make this usable the concept of borrow may have to be added, but only immutable borrowing should be added.

Once Ochr is sound and well-understood, the mutability of local variables and mutable references should be introduced.

7.1.2 Increase Feature Set, Increase Usability

Many important features are missing from Ochre which will be needed in order for it to become a useful tool for formal verification. On balance, I believe the theory work laid out above should be prioritized, but I am wary of spending too long developing the theory: Rust has transformed systems programming, and when it was released its specification was informal. To this day there is no official language specification other than the implementation of the reference compiler. Since Rust's inception, and in large part due to its success, academic rigor has been applied retrospectively with projects like Jung et al. [2018]. This approach has its advantages: it avoids investing academic effort into formal reasoning about languages which don't provide any advantage over the status quo for programmers. There is also the concern that the theory work could prove to be a bottomless pit of potential work, and doing it all properly would delay Ochre to the point of never happening, which could totally stop other contributors from joining in on the intellectual efforts. Or, to put it briefly: don't let "Perfect" to be the enemy of "Good", especially when "Good" could offer substantial benefits over the status quo.

Below are the most important features for Ochre to become a useful language:

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 **??**, 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:

```
Nat = ('zero, 'unit) | ('succ, Nat);
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. At least initially I suggest doing this through an embedding in Rust via its powerful (albeit untyped) metaprogramming features. This would allow Ochre to incrementally replace Rust codebases, which is a great potential use match.

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 would 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.

I have already implemented the parser for a previous version of Ochre in this way through macros, and code generation/abstract interpretation for a few simple constructs (assignment, atoms, pairs) is implemented. However, I decided the theory work was a more important contribution and focused my efforts there instead.

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 13

ATS-Implements, b. URL https://www.cs.bu.edu/~hwxi/atslangweb/Implements.html. pages 13

Benchmarks Game. URL https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gpp.html. pages 8

C gcc vs Classic Fortran - Which programs are fastest? (Benchmarks Game). URL https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/gcc-ifc.html. pages 8

Memory safety. URL https://www.chromium.org/Home/chromium-security/memory-safety/. pages 2

Polonius - Current status and roadmap. URL https://rust-lang.github.io/polonius/current_status.html. pages 62

A proactive approach to more secure code | MSRC Blog | Microsoft Security Response Center. URL https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/. pages 2

Stack Overflow Developer Survey 2023. URL https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023. pages 8

TIOBE Index. URL https://www.tiobe.com/tiobe-index/. pages 8

Rust Programming Language, 2015. URL https://www.rust-lang.org/. pages 2

Rust Stack Efficiency, November 2022. URL https://web.archive.org/web/20221128082216/https://arewestackefficientyet.com/. pages 8

Magmide/magmide. magmide, January 2024. URL https://github.com/magmide/magmide. pages 14

Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. ΠΣ: 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

Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796800020025. URL https://www.cambridge.org/core/journals/journal-of-functional-programming/article/introduction-to-generalized-type-systems/869991BA6A99180BF96A616894C6D710. pages 5

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

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '77*, pages 238–252, Los Angeles, California, 1977. ACM Press. doi: 10.1145/512950.512973. URL `http://portal.acm.org/citation.cfm?doid=512950.512973`. pages 7

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 4, 8, 11, 15, 18, 32, 35, 55

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 32, 80

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220, Big Sky Montana USA, October 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596. URL `https://dl.acm.org/doi/10.1145/1629575.1629596`. pages 2, 3

Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931. pages 2

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

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

Martin-L{\"o}f, Per and Sambin, Giovanni. Intuitionistic type theory. *Bibliopolis Naples*, 9, 1984. URL `https://people.csail.mit.edu/jgross/personal-website/papers/academic-papers-local/Martin-Lof80.pdf`. pages 5

Greg Morrisett, James Cheney, Dan Grossman, Michael Hicks, and Yanling Wang. Cyclone: A safe dialect of C. URL `https://homes.cs.washington.edu/~djg/papers/cyclone.pdf`. 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 63

Office of the National Cyber Director. Back to the Building Blocks: A Path Toward Secure and Measurable Software. Technical report, The White House, 2024. URL `https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf`. pages 2

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 32

Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. pages 71

Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V Thakur. DICE : A Formally Verified Implementation of DICE Measured Boot. pages 71

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 64

Sebastian Ullrich. Kha/electrolysis, January 2024. URL `https://github.com/Kha/electrolysis`. pages 18, 55

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 8

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

Hongwei Xi. Applied Type System: An Approach to Practical Programming with Theorem-Proving, March 2017. URL `http://arxiv.org/abs/1703.08683`. pages 13

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, Dallas Texas USA, October 2017. ACM. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134043. URL `https://dl.acm.org/doi/10.1145/3133956.3134043`. pages 71

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:

```haskell
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)`.

```haskell
  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 Abstract Interpretation

Here is a repeat of the Section **??** explanations of the objects in abstract interpretation, in the general form typically used in the literature:

Abstract interpretation can be seem as a framework for developing program analyses, which requires the semanticist to define the following mathematical objects:

Concrete and Abstract Set - Each element of the *concrete set* represents a set of possible concrete states the program could be in. The concrete set if often the power set of some concrete value. For example, if you are analysing a function which increments a number by one, it would map the concrete state $\{1, 5\}$ to the concrete state $\{2, 6\}$. Both $\{1, 5\}$ and $\{2, 6\}$ are elements of our concrete set, $\mathcal{P}(\mathbb{N})$. Each element of the *abstract set* represents an approximation of the concrete states the program could be in. Continuing the above example, we could store the possible *parities* of the number in our abstract state. The increment function would map $\{\mathsf{odd}\}$ to $\{\mathsf{even}\}$, and our abstract set would be $\mathcal{P}(\{\mathsf{true}, \mathsf{false}\})$. The fact it maps $\{\mathsf{odd}\}$ to $\{\mathsf{even}\}$ is intuitively: the input contains only odd numbers, and the output contains only even numbers.

Concrete and Abstract Semantics - The *concrete semantics* for a program maps a set of possible start states to a set of possible end states. The concrete semantics f for our above increment program might look something like $fl = \{x + 1 | x \in l\}$. The *abstract semantics* for a program map the a set of possible start abstract states to possible end abstract states. The abstract semantics f' for the continuing example would express the fact that incrementing a number always flips its parity: $fl = \{\mathsf{odd} | \mathsf{even} \in l\} \cup \{\mathsf{odd} | \mathsf{even} \in l\}$.

Concretization Function and Abstraction Function - The concretisation function maps from a set of possible approximations to the set of concrete values they could be approximating. The *abstraction function* maps from our sets of concrete states (elements of our concrete set) to sets of abstract states (elements of our abstract set). It answers: for a given concrete value, what is its approximation? But lifted to work on sets of concrete values and abstract values. In the above, it would map $\{1, 5\}$ to $\{odd\}$, reflecting the fact that the input is definitely odd.

A.3 Supporting Unboxed Pairs

While atoms, functions, and references are unboxed in Ochre, pairs are always heap-allocated. As discussed in Section **??**, 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** $\mathsf{box}\, t$**, 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.4 Derivations

A.4.1 Hello World Program Derivation

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

Derivation:

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

$\mathcal{D}_3 =$

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

$\mathcal{D}_4 =$

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

$\mathcal{D}_5 =$

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

$$\mathcal{D}_7 =$$

$$\begin{array}{ll}
\Omega_4 \vdash L_7 \sqsubseteq * \overset{\cdot}{\Rightarrow} (\text{'hello}, \_ \rightarrow \text{'world}) & \\
/\!/ \ \mathit{pair} & \\
\Omega_4 \vdash \texttt{pair} \overset{\cdot}{\Rightarrow} (\text{'hello}, \_ \rightarrow \text{'world}) \dashv \Omega_4' & \\
\quad\quad \Omega_4' = \Omega_4 \left[\dfrac{\texttt{pair} \mapsto \top}{\texttt{pair} \mapsto (\text{'hello}, \_ \rightarrow \text{'world})} \right] & \Omega_4' = \emptyset, \texttt{pair} \mapsto \top, \texttt{temp} \mapsto \top \\
\Omega_4' \vdash \mathsf{drop} & \\
\quad\quad \emptyset, \texttt{pair} \mapsto \top, \texttt{temp} \mapsto \top \vdash \mathsf{drop} & \\
\quad\quad\quad\quad \emptyset, \texttt{pair} \mapsto \top \vdash \mathsf{drop}\,\top & \\
\quad\quad\quad\quad \emptyset, \texttt{pair} \mapsto \top \vdash \mathsf{drop} & \\
\quad\quad\quad\quad\quad\quad \emptyset \vdash \mathsf{drop}\,\top & \\
\quad\quad\quad\quad\quad\quad \emptyset \vdash \mathsf{drop} &
\end{array}$$

A.4.2 Recursion

To make the derivation easier to understand, we shorten the program to `R = ('r, R); R.1.0`:

Begin derivation.

$$\begin{array}{ll}
\Omega_1 \vdash \texttt{R = ('r, R); R.1.0} \sqsubseteq * \overset{\cdot}{\rightsquigarrow} \text{'r} & \Omega_1 = \emptyset, \texttt{R} \mapsto \top \\
\quad\quad /\!/ \ \texttt{R = ('r, R)} & \\
\quad\quad \Omega_1 \vdash \texttt{('r, R)} \overset{\cdot}{\rightsquigarrow} (\text{'r}, \_ \rightarrow \texttt{R}) \dashv \Omega_1 & \\
\quad\quad\quad\quad \Omega_1 \vdash \texttt{('r, \_ -> R)} \overset{\cdot}{\rightsquigarrow} (\text{'r}, \_ \rightarrow \texttt{R}) \dashv \Omega_1 & \\
\quad\quad\quad\quad\quad\quad \Omega_1 \vdash \texttt{'r} \overset{\cdot}{\rightsquigarrow} \text{'r} & \\
\quad\quad\quad\quad\quad\quad \Omega_1 \vdash \_ \overset{\cdot}{\Leftarrow} \text{'r} & \\
\quad\quad\quad\quad\quad\quad \Omega_1 \vdash R \overset{\cdot}{\rightsquigarrow} \top /\!/ \ \textit{type check right of pair} & \\
\quad\quad \Omega_1 \vdash \texttt{R} \overset{\cdot}{\Leftarrow} (\text{'r}, \_ \rightarrow \texttt{R}) \dashv \Omega_2 & \Omega_2 = \emptyset, \texttt{R} \mapsto (\text{'r}, \_ \rightarrow \texttt{R}) \\
\quad\quad\quad\quad /\!/ \ \textit{variable write rule ommitted} & \\
\quad\quad /\!/ \ \texttt{R.1.0} & \\
\quad\quad \Omega_2 \vdash \texttt{R.1.0} \overset{\cdot}{\rightsquigarrow} \text{'r} & \\
\quad\quad\quad\quad \Omega_2 \vdash \texttt{R.1} \overset{\cdot}{\rightsquigarrow} (\text{'r}, \_ \rightarrow \texttt{R}) \dashv \Omega_2' & \\
\quad\quad\quad\quad\quad\quad \Omega_2 \vdash \texttt{R} \overset{\cdot}{\rightsquigarrow} (\text{'r}, \_ \rightarrow \texttt{R}) & \\
\quad\quad\quad\quad\quad\quad\quad\quad \Omega_2 \vdash \_ \overset{\cdot}{\leftsquigarrow} \text{'r} & \\
\quad\quad\quad\quad\quad\quad\quad\quad \Omega_2 \vdash \texttt{R} \overset{\cdot}{\rightsquigarrow} (\text{'r}, \_ \rightarrow \texttt{R}) &
\end{array}$$

A.4.3 Mutating a Dependent Pair

This appendix shows the derivation which type checks the definition of the `overwrite` function from Listing 20.

$$\begin{array}{ll}
\Omega_1 \vdash \texttt{(p:\&mut Same) -> 'unit \{*p.0 = 'a; *p.1 = 'a; 'unit\}} \mathrel{\dot{\Rightarrow}} (\texttt{p:\&mut Same}) \to \texttt{'unit} & \\
\quad \Omega_1 \vdash \texttt{(p:\&mut Same)} \underset{\mathsf{max}}{\dot{\Leftarrow}} \mathsf{borrow}^{\mathsf{m}}\, l\, p_0 \dashv \Omega_{10} & \\
\quad\quad \Omega_1 \vdash \texttt{\&mut Same} \mathrel{\dot{\rightsquigarrow}} \mathsf{borrow}^{\mathsf{m}}\, l\, p_0 \dashv \Omega'_1 & \\
\quad\quad\quad \Omega_1 \vdash \texttt{Same} \mathrel{\dot{\rightsquigarrow}} p_0 & \\
\quad\quad\quad\quad \texttt{Same} \mapsto p_0 \in \Omega_1 & p_0 = (\{\text{'a}, \text{'b}\}, L \to L) \\
\quad\quad\quad \Omega'_1 = \Omega_1, l \mapsto p_0 & \Omega'_1 = \Omega_1, l \mapsto p_0 \\
\quad\quad \Omega'_1 \vdash \texttt{p} \mathrel{\dot{\Leftarrow}} \mathsf{borrow}^{\mathsf{m}}\, l\, p_0 \dashv \Omega_{10} & \langle\text{Rearrange-Before}\rangle \\
\quad\quad\quad \Omega'_1 \hookrightarrow \Omega''_1 & \langle\text{Allocate}\rangle \\
\quad\quad\quad\quad \Omega''_1 = \Omega'_1, \texttt{p} \mapsto \top & \Omega''_1 = \Omega'_1, \texttt{p} \mapsto \top \\
\quad\quad\quad \Omega''_1 \vdash \texttt{p} \mathrel{\dot{\Leftarrow}} \mathsf{borrow}^{\mathsf{m}}\, l\, p_0 \dashv \Omega_{10} & \\
\quad\quad\quad\quad \Omega_{10} = \Omega'_1 \left[\dfrac{\texttt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_0}{p \mapsto \top}\right] & \Omega_{10} = \Omega'_1, \texttt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_0 \\
\quad \Omega_{10} \vdash \texttt{*p.0 = 'a; *p.1 = 'a; 'unit} \sqsubseteq \texttt{'unit} \mathrel{\dot{\Rightarrow}} \text{'unit} & \\
\quad\quad \mathcal{D}_{10} &
\end{array}$$

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

$$\begin{array}{lll}
\Omega_{10} \vdash \texttt{*p.0 = 'a; *p.1 = 'a; 'unit} \sqsubseteq \texttt{'unit} \mathrel{\dot{\Rightarrow}} \text{'unit} & \Omega_{10} \vdash \texttt{'a} \mathrel{\dot{\Rightarrow}} \text{'a} & \\
\quad \Omega_{10} \vdash \texttt{*p.0} \mathrel{\dot{\Leftarrow}} \text{'a} \dashv \Omega_{11} & & \langle\text{Rearrange-Before}\rangle \\
\quad\quad \Omega_{10} \hookrightarrow \Omega'_{10} \text{ // } \textit{must set } {*p}.0 \textit{ to } \top \textit{ to write} & & \langle\text{Type-Widen}\rangle \\
\quad\quad\quad \Omega'_{10} = \Omega_{10} \left[\dfrac{\texttt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_1}{\texttt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_0}\right] & & \Omega'_{10} = \Omega'_1, p \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_1 \\
\quad\quad\quad \Omega_{10} \vdash \mathsf{borrow}^{\mathsf{m}}\, l\, p_0 \sqsubseteq \mathsf{borrow}^{\mathsf{m}}\, l\, p_1 & & p_1 = (\top, _ \to \{\text{'a}, \text{'b}\}) \\
\quad\quad\quad\quad \Omega_{10} \vdash p_0 \sqsubseteq p_1 & & \\
\quad\quad\quad\quad\quad \Omega_{10} \vdash \{\text{'a}, \text{'b}\} \sqsubseteq \top & & \\
\quad\quad\quad\quad\quad \Omega_{10} \vdash \mathsf{comptime} \dashv \Gamma_1 & & \Gamma_1 = \Omega_1, \texttt{L} \mapsto \top \\
\quad\quad\quad\quad\quad \Gamma_1 \vdash _ \mathrel{\dot{\leftsquigarrow}} \top & & \\
\quad\quad\quad\quad\quad \Gamma_1 \vdash \texttt{L} \mathrel{\dot{\leftsquigarrow}} \{\text{'a}, \text{'b}\} \dashv \Gamma'_1 & & \\
\quad\quad\quad\quad\quad\quad \Gamma'_1 = \Gamma_1 \left[\dfrac{\texttt{L} \mapsto \{\text{'a}, \text{'b}\}}{\texttt{L} \mapsto \top}\right] & & \\
\quad\quad\quad\quad\quad \Gamma'_1 \vdash \texttt{L} \sqsubseteq \{\text{'a}, \text{'b}\} \mathrel{\dot{\rightsquigarrow}} \{\text{'a}, \text{'b}\} & & \\
\quad\quad\quad\quad\quad\quad \Gamma'_1 \vdash L \mathrel{\dot{\rightsquigarrow}} \{\text{'a}, \text{'b}\} & & \\
\quad\quad\quad\quad\quad\quad\quad \texttt{L} \mapsto \{\text{'a}, \text{'b}\} \in \Gamma'_1 & & \\
\quad\quad\quad\quad\quad\quad \Gamma'_1 \vdash \{\text{'a}, \text{'b}\} \mathrel{\dot{\rightsquigarrow}} \{\text{'a}, \text{'b}\} & & \\
\quad\quad\quad\quad\quad\quad \Gamma'_1 \vdash \{\text{'a}, \text{'b}\} \sqsubseteq \{\text{'a}, \text{'b}\} & & \\
\quad\quad\quad\quad\quad\quad\quad \{\text{'a}, \text{'b}\} \subseteq \{\text{'a}, \text{'b}\} & & \\
\quad\quad \Omega'_{10} \vdash \texttt{*p.0} \mathrel{\dot{\Leftarrow}} \text{'a} \dashv \Omega_{11} \text{ // } \textit{write to } \texttt{*p.0} & & \\
\quad\quad\quad \Omega'_{10} \vdash \texttt{*p} \mathrel{\dot{\Rightarrow}} p_1 \dashv \Omega'''_{10} & & \\
\quad\quad\quad\quad \Omega'_{10} \vdash \texttt{p} \mathrel{\dot{\Rightarrow}} \mathsf{borrow}^{\mathsf{m}}\, l\, p_1 \vdash \Omega''_{10} & & \\
\quad\quad\quad\quad\quad \Omega''_{10} = \Omega'_{10} \left[\dfrac{\texttt{p} \mapsto \top}{\texttt{p} \mapsto p_1}\right] & & \Omega''_{10} = \Omega'_1, \texttt{p} \mapsto \top \\
\quad\quad\quad\quad \Omega''_{10} \vdash \texttt{p} \mathrel{\dot{\Leftarrow}} \mathsf{borrow}^{\mathsf{m}}\, l\, \top \vdash \Omega'''_{10} & & \\
\quad\quad\quad\quad\quad \Omega'''_{10} = \Omega''_{10} \left[\dfrac{\texttt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, \top}{\texttt{p} \mapsto \top}\right] & & \Omega'''_{10} = \Omega'_1, \texttt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, \top \\
\quad\quad\quad \Omega'''_{10} \vdash \mathsf{left}\, p_1 = \top & & \\
\quad\quad\quad \Omega'''_{10} \vdash \mathsf{right}\, p_1 = \{\text{'a}, \text{'b}\} & & \\
\quad\quad\quad \Omega'''_{10} \vdash \texttt{*p} \mathrel{\dot{\Leftarrow}} p_2 \dashv \Omega_{11} & & p_2 = (\text{'a}, _ \to \{\text{'a}, \text{'b}\}) \\
\quad\quad\quad\quad \Omega'''_{10} \vdash \texttt{p} \mathrel{\dot{\Rightarrow}} \mathsf{borrow}^{\mathsf{m}}\, l\, \top \dashv \Omega''''_{10} & & \\
\quad\quad\quad\quad\quad \Omega''''_{10} = \Omega'''_{10} \left[\dfrac{\texttt{p} \mapsto \top}{p \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, \top}\right] & & \Omega''''_{10} = \Omega'_1, \texttt{p} \mapsto \top \\
\quad\quad\quad\quad \Omega''''_{10} \vdash \texttt{p} \mathrel{\dot{\Leftarrow}} \mathsf{borrow}^{\mathsf{m}}\, l\, p_2 \dashv \Omega_{11} & & \\
\quad\quad\quad\quad\quad \Omega_{11} = \Omega''''_{10} \left[\dfrac{\texttt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_2}{p \mapsto \top}\right] & & \Omega_{11} = \Omega'_1, \texttt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_2 \\
\quad \Omega_{11} \vdash \texttt{*p.1 = 'a; 'unit} \sqsubseteq \texttt{'unit} \mathrel{\dot{\Rightarrow}} \text{'unit} & & \\
\quad\quad \mathcal{D}_{11} & &
\end{array}$$

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

$$\begin{array}{ll}
\Omega_{11} \vdash \texttt{*p.1 = 'a; 'unit} \sqsubseteq \texttt{'unit} \;\dot{\Rightarrow}\; \text{'unit} & \\
\quad \Omega_{11} \vdash \texttt{'a} \;\dot{\Rightarrow}\; \text{'a} & \\
\quad \Omega_{11} \vdash \texttt{*p.1} \;\dot{\Leftarrow}\; \text{'a} \dashv \Omega_{12} & \Omega_{12} = \Omega_1', \texttt{p} \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_3 \\
\qquad \textit{// Similar to } \Omega_{10}' \vdash \texttt{*p.0} \;\dot{\Leftarrow}\; \text{'a} \dashv \Omega_{11} & p_3 = (\text{'a}, \_ \to \text{'a}) \\
\quad \Omega_{12} \vdash \texttt{'unit} \sqsubseteq \texttt{'unit} \;\dot{\Rightarrow}\; \text{'unit} & \\
\qquad \mathcal{D}_1 2 &
\end{array}$$

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

$$\begin{array}{ll}
\Omega_{12} \vdash \texttt{'unit} \sqsubseteq \texttt{'unit} \;\dot{\Rightarrow}\; \text{'unit} & \\
\quad \Omega_{12} \vdash \texttt{'unit} \;\dot{\Rightarrow}\; \text{'unit} & \\
\quad \text{// } \Omega_{12} \vdash \mathsf{drop} \; = & \\
\quad \Omega_1, l \mapsto p_0, p \mapsto \mathsf{borrow}^{\mathsf{m}}\, l\, p_3 \vdash \mathsf{drop} & \textit{// final cleanup} \\
\qquad \Omega_1, l \mapsto p_0 \vdash \mathsf{drop}\, (\mathsf{borrow}^{\mathsf{m}}\, l\, p_3) \dashv \Omega_1 & \\
\qquad\quad \Omega_1 = \Omega_1, l \mapsto p_0 \backslash \{l \mapsto p_0\} & \\
\qquad\quad \Omega_1 \vdash p_3 \sqsubseteq p_0 & \\
\qquad\qquad \Omega_1 \vdash \text{'a} \sqsubseteq \{\text{'a}, \text{'b}\} & \\
\qquad\qquad\quad \{\text{'a}\} \subseteq \{\text{'a}, \text{'b}\} & \\
\qquad\qquad \Omega_1 \vdash \mathsf{comptime} \dashv \Gamma_2 & \Gamma_2 = \Omega_1, \texttt{L} \mapsto \top \\
\qquad\qquad \Gamma_2 \vdash \texttt{L} \;\dot{\leftsquigarrow}\; \text{'a} \dashv \Gamma_2' & \\
\qquad\qquad\quad \Gamma_2' = \Gamma_2 \left[\dfrac{\texttt{L} \mapsto \text{'a}}{\texttt{L} \mapsto \top} \right] & \Gamma_2' = \Omega_1, \texttt{L} \mapsto \text{'a} \\
\qquad\qquad \Gamma_2 \vdash \_ \;\dot{\leftsquigarrow}\; \text{'a} & \\
\qquad\qquad \Gamma_2' \vdash \text{'a} \sqsubseteq \texttt{L} \;\dot{\rightsquigarrow}\; \text{'a} & \\
\qquad\qquad\quad \Gamma_2' \vdash \text{'a} \;\dot{\rightsquigarrow}\; \text{'a} & \\
\qquad\qquad\quad \Gamma_2' \vdash L \;\dot{\rightsquigarrow}\; \text{'a} & \\
\qquad\qquad\qquad \texttt{L} \mapsto \text{'a} \in \Gamma_2' & \\
\qquad\qquad\quad \Gamma_2' \vdash \text{'a} \sqsubseteq \text{'a} & \\
\qquad\qquad\qquad \{\text{'a}\} \subseteq \{\text{'a}\} & \\
\quad \Omega_{12} \vdash \texttt{'unit} \sqsubseteq * \;\dot{\rightsquigarrow}\; \text{'unit} & \\
\quad \Omega_{12} \vdash \text{'unit} \sqsubseteq \text{'unit} &
\end{array}$$

A.5 Properties and Proofs

A.5.1 Statement Soundness

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

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

We start by assuming all of the premises:

$$\begin{gathered} \Delta \vdash S \sqsubseteq * \Rightarrow v \\ \Omega \vdash S \sqsubseteq * \mathrel{\dot{\Rightarrow}} t \\ \Delta : \Omega \end{gathered} \tag{A.1}$$

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 $\Omega \vdash S \sqsubseteq * \mathrel{\dot{\Rightarrow}} t$ 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 the abstract interpretation drops a variable if and only if the concrete interpretation drops a variable. Formally: either $\Delta \vdash S \sqsubseteq * \Rightarrow v$ *and* $\Omega \vdash S \sqsubseteq * \mathrel{\dot{\Rightarrow}} t$ are rearrangements (⟨Rearrange-Before⟩ or ⟨Rearrange-After⟩), or neither are. 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.

Match - S is an expression M.

The assumed derivations must be

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

for some environments Δ' and Ω'.

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

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

The assumed derivations must be

$$\frac{\begin{array}{c}\Delta \vdash N \Rightarrow v' \dashv \Delta' \\ \Delta' \vdash M \Leftarrow{}'v \dashv \Delta'' \\ \textsf{runtime}\ M \\ \textit{// execute remaining computation} \\ \Delta'' \vdash S' \sqsubseteq * \Rightarrow v\end{array}}{\Delta \vdash M\texttt{=}N\,;'S' \sqsubseteq S_t \Rightarrow v} \quad \text{and} \quad \frac{\begin{array}{c}\Omega \vdash N \ \dot{\Rightarrow}\ t' \dashv \Omega' \\ \Omega' \vdash M \ \dot{\Leftarrow}\ t' \dashv \Omega'' \\ \textsf{runtime}\ M \\ \textit{// execute remaining computation} \\ \Omega'' \vdash S' \sqsubseteq * \ \dot{\Rightarrow}\ t\end{array}}{\Omega \vdash M\texttt{=}N\,;'S' \sqsubseteq S_t \ \dot{\Rightarrow}\ t} \tag{A.3}$$

for some environments Δ, Δ', Δ'', Ω, Ω', Ω'' and types v and t.

By the soundness of read interpretation on expressions (5.2.1), with $\Delta \vdash N \Rightarrow v \dashv \Delta'$ and $\Omega \vdash N \ \dot{\Rightarrow}\ t \dashv \Omega'$ from A.5.1, we get $\Omega' \vdash v : t$ and $\Delta' \sqsubseteq \Omega'$.

Using this with soundness of *write* soundness for expressions (5.2.1) with $\Delta' \vdash M \Leftarrow{}'v \dashv \Delta''$ and $\Omega' \vdash M \ \dot{\Leftarrow}\ t' \dashv \Omega''$ and the previous conclusions $\Omega' \vdash v : t$ and $\Delta' \sqsubseteq \Omega'$, we get $\Delta'' \sqsubseteq \Omega''$.

Because we have $\Delta'' \sqsubseteq \Omega''$, we can use statement soudness (5.2.1) on $\Delta'' \vdash S' \sqsubseteq * \Rightarrow v$ and $\Omega'' \vdash S' \sqsubseteq * \ \dot{\Rightarrow}\ t$ to get $\Omega'' \vdash v : t$, as required.

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

Match omitted.

A.5.2 Expression Read Soundness

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

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

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

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.

Match - M is an atom constructor 'a

The assumed derivations $\Delta \vdash M \diamond v \dashv \Delta'$ and $\Omega \vdash M \ \dot{\diamond}\ t \dashv \Omega'$ 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. □

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

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

$$\frac{\Delta' = \Delta \left[\frac{x \mapsto \top}{x \mapsto v}\right]}{\Delta \vdash x \Rightarrow v \dashv \Delta'} \text{ and } \frac{\Omega' = \Omega \left[\frac{x \mapsto \top}{x \mapsto t}\right]}{\Omega \vdash x \overset{\cdot}{\Rightarrow} t \dashv \Omega'}. \tag{A.4}$$

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* $\Omega\rangle$. □.

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

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

$$\frac{\begin{array}{c}\Delta \vdash F \rightarrow (M \rightarrow S) \\ \Delta \vdash A \Rightarrow a \dashv \Delta_1 \\ \Delta_1 \vdash M \Leftarrow a \dashv \Delta_2 \\ \Delta_2 \vdash S \Rightarrow v \dashv \Delta'\end{array}}{\Delta \vdash FA \Rightarrow v \dashv \Delta'} \text{ and } \frac{\begin{array}{c}\Omega \vdash F \overset{\cdot}{\rightarrow} (\dot{M} \rightarrow S_t) \\ \Omega \vdash A \overset{\cdot}{\Rightarrow} \dot{a} \dashv \Omega_1 \\ \Omega_1 \vdash \dot{M} \overset{\cdot}{\Leftarrow} \dot{a} \dashv \Omega_2 \\ \Omega_2 \vdash S_t \sqsubseteq * \overset{\cdot}{\leadsto} t \\ \Omega_1 \vdash \text{drop}\, t \dashv \Omega'\end{array}}{\Omega \vdash FA \overset{\cdot}{\Rightarrow} t \dashv \Omega'} \tag{A.5}$$

Subgoal $M = \dot{M}$ **and** $\Omega \vdash (M \rightarrow S) \sqsubseteq (\dot{M} \rightarrow S_t)$ - We use Expression Read Soundness (induction) on $\Delta \vdash F \rightarrow (M \rightarrow S)$ and $\Omega \vdash F \overset{\cdot}{\rightarrow} (\dot{M} \rightarrow S_t)$ to conclude that $\Omega \vdash (M \rightarrow S) \sqsubseteq (\dot{M} \rightarrow S_t)$. By $\langle$*def.* $\sqsubseteq$ *for* $M \rightarrow S\rangle$, 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 $\Omega \vdash A \overset{\cdot}{\Rightarrow} \dot{a} \dashv \Omega_1$ 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 $\Omega_1 \vdash \dot{M} \overset{\cdot}{\Leftarrow} \dot{a} \dashv \Omega_2$ 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 ($\Omega \vdash (M \rightarrow S) \sqsubseteq (\dot{M} \rightarrow S_t)$) to get:

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

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

$$\begin{gathered} \Omega_2 \vdash S \sqsubseteq * \Rightarrow u \\ \Omega_2 \vdash S_t \sqsubseteq * \dot{\rightsquigarrow} t' \\ \Omega_2 \vdash u \sqsubseteq t' \end{gathered} \tag{A.6}$$

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.6). 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.6).

Subgoal: $\Omega_2 \vdash t' \sqsubseteq t$ - (A.6) gives us $\Omega_2 \vdash S_t \sqsubseteq * \dot{\rightsquigarrow} t'$ and (A.5) 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 $\textsf{concrete}\, v$ from ⟨*def. : for t*⟩. 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 $\textsf{concrete}\, v$ 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 $\textsf{borrow}^{\textsf{m}}\, l\, t$. 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 (⟨*def.* $\Rightarrow$ *for* $M \to S$⟩), 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.

Match - M is a function definition M'->$S_t\{S\}$.

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

$$\frac{\textit{// no checking}}{\Delta \vdash M'\texttt{->}S_t\{S\} \Rightarrow M \to S} \text{ and } \frac{\begin{array}{c}\Omega \vdash \mathsf{comptime} \;\dashv \Gamma \\ \Gamma \vdash M' \underset{\mathsf{max}}{\dot{\Leftarrow}} m \dashv \Gamma' \\ \Gamma' \vdash S \sqsubseteq S_t \;\dot{\Rightarrow}\; u\end{array}}{\Omega \vdash M'\texttt{->}S_t\{S\} \;\dot{\Rightarrow}\; M \to S_t} \tag{A.7}$$

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.5.1).

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

This match is so short because ⟨*def.* $\sqsubseteq$ *for* $M \to S_t$⟩ is almost an exact match of ⟨*def.* $\dot{\Rightarrow}$ *for* M'->$S_t\{S\}$⟩. The complexity of function checking is all in the application match. 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.

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

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

$$\frac{\Delta \vdash M \Rightarrow v \dashv \Delta'}{\Delta \vdash M' {:} T \Rightarrow v \dashv \Delta'} \text{ and } \frac{\begin{array}{c}\Omega \vdash M \;\dot{\Rightarrow}\; m \dashv \Omega' \\ \Omega' \vdash T \dot{\rightsquigarrow} t \\ \Omega' \vdash m \sqsubseteq t\end{array}}{\Omega \vdash M' {:} T \;\dot{\Rightarrow}\; t \dashv \Omega'} \tag{A.8}$$

Because expression read interpretation is sound (5.2.1, induction), we can use $\Delta \vdash M \Rightarrow v \dashv \Delta'$ and $\Omega \vdash M \;\dot{\Rightarrow}\; m \dashv \Omega'$ 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.