Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Ochre: A Dependently Typed Systems Programming Language

Author:
Charlie Lidbury

Supervisor(s):
Steffen van Bakel
Nicolas Wu

June 8, 2024

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

Abstract

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

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

Acknowledgments

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

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

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

Ethical Considerations

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

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

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

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

Contents

Chapter 1

Introduction

(TAKEN FROM AENEAS FOR NOW)

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

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

We have attempted to prove further properties

1.1 The Problem

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

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

for them to go wrong at runtime.

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

This type-checker is being built to hopefully be used for a larger, more useful language in the future, called Ochre. Ochre which will have both the speed of *systems languages* like C and Rust and the ability to reason about runtime behaviour at compile time of *theorem provers* like Agda and Coq. Exactly what systems languages and theorem provers are is discussed in Chapter 2.1.

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

1.1.1 Why Is It Hard?

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

1.2 The Solution

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

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

1.3 Motivation

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

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

1.3.1 Mutation

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

Performance

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

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

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

Usability

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

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

The Immutability Argument

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

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

Popularity

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

1.3.2 Dependent Types

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

Formal Verification

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

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

Usability

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

1.3.3 Mutation + Dependent Types

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

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

1.3.4 This Particular Method

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

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

Chapter 2

Background

2.1 Prerequisite Concepts

This section explains the concepts required to understand this research.

2.1.1 Mutability

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

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

2.1.2 Dependent Types

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

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

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

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

2.1.3 Formal Verification with Dependent Types

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

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

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

2.1.4 Rust

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

Rust is a relatively recent programming language that offers a unique combination of strong (memory) safety guarantees and bare-metal performance.

To generate optimal code, systems languages let the programmer manage their memory, and choose memory layouts. In doing so, they typically sacrifice the memory safety guarantees higher-level languages make due to not being able to check the programmer has

managed their memory correctly, this is the case in C and C++. Rust uses a concept called *ownership* to recover these memory safety guarantees while still giving the programmer sufficient control to match C and C++'s performance.

Ownership

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

There are three rules associated with ownership in Rust:

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

Borrowing And The Borrow Checker

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

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

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

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

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

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

references denoted by `&mut T`. For any given value, you can either hold a single mutable reference or n immutable references, but never both at the same time. This is called the aliasing xor (exclusive or) constraint, or AXM for short.

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

2.1.5 Mutable → Immutable Translation

To reason about and type-check the mutable code from the programmer, the type checker this research presents translates the source code into an immutable version, as outlined in Section 1.2.

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

Listing 2.1: Original

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

Listing 2.2: Translated

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

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

Listing 2.3: Original

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

Listing 2.4: Translated

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

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

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

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

supported by either of these prior works, so I can use the translation algorithms they have already developed.

2.2 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.

2.2.1 Languages with Mutability and Dependent Types

ATS

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

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

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

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

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

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

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

Magmide

The goal of Magmide [**?**] is to "create a programming language capable of making formal verification and provably correct software practical and mainstream". Currently, Magmide is unimplemented, and there are barely even code snippets of it. However, there is extensive design documentation in which the author Blaine Hansen lays out the compiler architecture he intends to use, which involves two internal representations: *logical* Magmide and *host* Magmide.

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

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

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

Low*

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

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

It is very much not a pay-for-what-you-use language, to compile anything you must manually manage things like pushing and popping frames on and off the stack, so even if it can achieve impressive results, it's only useful for teams willing to pay the high price which comes

with verifying the entire program. This research aims to be better by not requiring any effort from the programmer in the case that they do not wish to use dependent types for their reasoning power.

2.2.2 Embedding Mutability in Languages With Dependent Types

Ynot: Dependent Types for Imperative Programs

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

2.2.3 Formal Verification of Low-Level Code

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

Rust Belt

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

I see RustBelt as a great complement to this work in the future: real programs require unsafe code, but you want to avoid having to model your code in a separate proof assistant as little as possible. In Ochre, I imagine the few people who write unsafe code will verify it with something like RustBelt, while the majority won't have to, but will benefit from the guarantees provided by the verified libraries they use which do.

Chapter 3

Ochre

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

	Runtime Semantics	Static Analysis
Intuition Building	Section 3.1 Ochre, by Example	Section 3.2 Type & Borrow Checking, by Example
Formal	Section 3.3 Concrete Interpretation	Section 3.4 Abstract Interpretation

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

Attribution

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

The mutation & memory management techniques presented originated in 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; for example `'one`: 'one. TypeScript partly does this with literal types, producing results like `5:5`, but Ochre takes this to its logical conclusion where even functions are their own type, and there is almost no distinction between types and terms apart from the requirement that all types are resolved at compile time.

explain double page + hyper links thing

do hyperlinks thing

3.1 Ochre, by Example

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

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

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

Pairs

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

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

Move Semantics

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

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

```
1  x = 'one;
2  y = x;
3  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);
```

Ochre is structurally typed, like TypeScript. 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 evaluates to the type of atoms which can any of `'a`, `'b`, or `'c`.

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

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.

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

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

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

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

Comptime vs Runtime

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

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

Functions

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

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

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

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

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

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

Case Statements

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

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

Dependent Pairs

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

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

Different = (Bool, L -> case L { 'true => 'false, 'false => 'true});
('true, 'false): Different; // valid
('true, 'true): Different; // error!
```

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

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

Listing 3

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

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

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

Type Narrowing

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

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

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

Algebraic Data Types

Take the following definition of Peano naturals in Haskell syntax:

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

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

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

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

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

Recursion

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

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

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

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

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

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

3.2 Type & Borrow Checking, by Example

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 $\{'a, 'b\}$.

```
1    x = 'true; // {x ↦ {'true}}
2    y = 'hello; // {x ↦ {'true}, y ↦ {'hello}}
3    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.

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

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

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

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

References & Mutation

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

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

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

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

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

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

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

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

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

Listing 9: A reborrow

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

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

Listing 10: An immutable re-borrow

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

(Dependent) Pairs

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

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

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

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

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

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

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

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

Type Annotations

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

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

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

Comptime vs Runtime

As you will see in Section 3.4, there are large differences in how the abstract interpretation is performed on runtime and comptime terms; however, for the most part, they map very similarly to the abstract environment. Following from the syntax level distinction, an entry in the abstract environment is marked as runtime or comptime by the variable identifier being capitalized or not.

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

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

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

```
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}

Case Statements

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

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

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

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

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

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

Complex Example Programs

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

3.3 Concrete Interpretation

The primary contribution of this research is the abstract interpretation because that is what will be used in the compiler for the type checking, but the regular interpretation is still useful to better understand the language features and to better understand the abstract interpretation. If you're only interested in how Ochre is type-checked, not its exact behavior at runtime, skip to Section 3.4.

Ochre has an extremely permissive syntax and heavily relies on typing rules to reject ill-formed programs. This leads to some usual results like `(x = y) = z` being syntactically well-formed. In practice, this doesn't cause issues, because the typing rules are strict enough. The grammar for Ochre is shown in Figure 3.1.

$T, U, M, N ::=$

`x` | `y` | `z`	// *runtime variable identifier*
`X` | `Y` | `Z`	// *comptime variable identifier*
`'`a	// *atom construction*
M`,`N	// *pair construction*
M`.0`	// *pair left access*
M`.1`	// *pair right access*
`*`M	// *dereference*
`&`M | `&mut` M	// *borrow constructor*
$M N$	// *function application*
M `->` T `{` N `}`	// *function definition (optional runtime body)*
M`=`N	// *assignment*
M`;`N	// *sequence*
`case` M `{` $\overrightarrow{\text{'}a \texttt{ => } N}$ `}`	// *case statement*
`_`	// *uninitialised*
`*`	// *top*
T `|` U	// *type union*
M`:`T	// *type constraint*

Figure 3.1: Ochre syntax

We define two judgments over this syntax: $\Delta \vdash \texttt{M} \Rightarrow v \dashv \Delta'$ and $\Delta \vdash \texttt{M} \Leftarrow v \dashv \Delta'$ where Δ and $Delta'$ are the stack states before and after, `M` is a piece of syntax, and v is a runtime value. The syntax for Δ and v are given in Figure 3.5

`M` $\Rightarrow$ v defines what it means to evaluate a piece of syntax, and `M` $\Leftarrow$ v defines what it means to *write* to a piece of syntax. Not all syntax can be written to, but parts can. This roughly equates to the concept of an lvalue in other languages.

As an example: writing the value 'five, to the syntax `x` will mutate the entry for `x` on the stack and set it to 'five. Formally, this is expressed via the following definition:

$$\frac{\Delta' = \Delta \left[\frac{\mathtt{x} \mapsto v}{\mathtt{x} \mapsto \bot} \right]}{\Delta \vdash \mathtt{x} \Leftarrow v \dashv \Delta'}$$

Figure 3.2: Definition of $\Leftarrow$ for variable identifiers

This reads "the value v is written to the syntax x when the stack is Δ, it will modify Δ such that x now maps to v instead of the previous $\bot$".

For this rule to be applicable, x must be uninitialized in the stack, which requires both that x is in the stack at all, and that x hasn't been written to previously, which is a very specific state for the stack to be in. There is a set of operations that can be performed at any time which perform things like memory allocation/deallocation, which one can use during derivations to get the stack into the right state, which will be covered later.

Figures 3.3 and 3.4 contain the concrete semantics for every piece of syntax. Often the rule for reading from a piece of syntax and writing to it differs only in which arrow is used; in these cases, I quantify over arrows using $\diamond$ to mean "either arrow".

	$\Rightarrow$	$\Leftarrow$
`x`	$\dfrac{\texttt{x} \mapsto v \in \Delta}{\Delta \vdash \texttt{x} \Rightarrow v}$	$\dfrac{\Delta' = \Delta\left[\dfrac{\texttt{x} \mapsto v'}{\texttt{x} \mapsto \bot}\right]}{\Delta \vdash \texttt{x} \Leftarrow v' \dashv \Delta'}$
`'a`	$\forall \diamond \in \{\Rightarrow, \Leftarrow\}. \left[\dfrac{}{\Delta \vdash \texttt{'a} \diamond \texttt{'a}}\right]$	
`M, N`	$\forall \diamond \in \{\Rightarrow, \Leftarrow\}. \left[\dfrac{\Delta \vdash \texttt{M} \diamond v \dashv \Delta' \quad \Delta' \vdash \texttt{N} \diamond w \dashv \Delta''}{\Delta \vdash \texttt{M, N} \diamond (v, w) \dashv \Delta''}\right]$	
`M.0`	$\dfrac{\Delta \vdash \texttt{M} \Rightarrow (v, \_) \dashv \Delta'}{\Delta \vdash \texttt{M.0} \Rightarrow v \dashv \Delta'}$	$\dfrac{\Delta \vdash \texttt{M} \Rightarrow (\bot, w) \dashv \Delta' \quad \Delta' \vdash \texttt{M} \Leftarrow (v', w) \dashv \Delta''}{\Delta \vdash \texttt{M.0} \Rightarrow v' \dashv \Delta''}$
`M.1`	$\dfrac{\Delta \vdash \texttt{M} \Rightarrow (\_, w) \dashv \Delta'}{\Delta \vdash \texttt{M.1} \Rightarrow w \dashv \Delta'}$	$\dfrac{\Delta \vdash \texttt{M} \Rightarrow (v, \bot) \dashv \Delta' \quad \Delta' \vdash \texttt{M} \Leftarrow (v, w') \dashv \Delta''}{\Delta \vdash \texttt{M.1} \Rightarrow w' \dashv \Delta''}$
`*M`	$\forall \diamond \in \{\Rightarrow, \Leftarrow\}. \left[\dfrac{\Delta \vdash \texttt{M} \Rightarrow p \dashv \Delta' \quad \Delta' \vdash p \diamond v \dashv \Delta''}{\Delta \vdash \texttt{*M} \diamond v \dashv \Delta''}\right]$	
`M:T`	$\forall \diamond \in \{\Rightarrow, \Leftarrow\}. \left[\dfrac{\Delta \vdash \texttt{M} \diamond v \dashv \Delta'}{\Delta \vdash \texttt{M:T} \diamond v \dashv \Delta'}\right]$	
`_`	$\forall \diamond \in \{\Rightarrow, \Leftarrow\}. \left[\dfrac{}{\Delta \vdash \_ \diamond \bot}\right]$	

Figure 3.3: Concrete interpretation for the pieces of syntax which can be both read and written to.

	$\Rightarrow$
`&M`	$\dfrac{}{\Delta \vdash \texttt{\&x} \Rightarrow \texttt{x}} \quad \dfrac{\Delta \vdash \texttt{M} \Rightarrow p}{\Delta \vdash \texttt{\&M.0} \Rightarrow p.0} \quad \dfrac{\Delta \vdash \texttt{M} \Rightarrow p}{\Delta \vdash \texttt{\&M.1} \Rightarrow p.1}$
`&mut M`	$\dfrac{\Delta \vdash \texttt{\&M} \Rightarrow p}{\Delta \vdash \texttt{\&mut M} \Rightarrow p}$
`M N`	$\dfrac{\begin{array}{ll}\Delta \vdash \texttt{M} \Rightarrow (\texttt{I} \to \texttt{O}) \dashv \Delta' & /\!/ \textit{ evaluate function} \\ \Delta' \vdash \texttt{N} \Rightarrow n \dashv \Delta'' & /\!/ \textit{ evaluate argument} \\ \Delta'' \vdash \texttt{I} \Leftarrow n \dashv \Delta''' & /\!/ \textit{ push argument to stack} \\ \Delta''' \vdash \texttt{O} \Rightarrow v \dashv \Delta'''' & /\!/ \textit{ execute function body}\end{array}}{\Delta \vdash \texttt{M N} \Rightarrow v \dashv \Delta''''}$
`M -> T{N}`	$\dfrac{}{\Delta \vdash \texttt{M -> T \{ N \}} \Rightarrow (\texttt{M} \to \texttt{N})}$
`M = N`	$\dfrac{\Delta \vdash \texttt{N} \Rightarrow v \dashv \Delta' \quad \Delta' \vdash \texttt{M} \Leftarrow v \dashv \Delta''}{\Delta \vdash \texttt{M = N} \Rightarrow \text{'unit} \dashv \Delta''}$
`M; N`	$\dfrac{\Delta \vdash \texttt{M} \Rightarrow \text{'unit} \dashv \Delta' \quad \Delta' \vdash \texttt{N} \Rightarrow n \dashv \Delta''}{\Delta \vdash \texttt{M = N} \Rightarrow n \dashv \Delta''}$
`case M { 'a0 => N0, ... 'ak => Nk, }`	$\dfrac{\begin{array}{ll}\Delta \vdash \texttt{M} \Rightarrow \text{'a}_i \dashv \Delta' & /\!/ \textit{ interrogant matches } i^{th} \textit{ branch} \\ \Delta' \vdash \texttt{N}_i \Rightarrow n_i \dashv \Delta'' & /\!/ \textit{ execute } i^{th} \textit{ branch}\end{array}}{\Delta \vdash \texttt{case M \{ 'a}_0 \texttt{ => N}_0 \ \ldots \ \texttt{\}} \Rightarrow n_i \dashv \Delta''}$

Figure 3.4: Concrete interpretation for the pieces of syntax which can be evaluated but not written to.

3.4 Abstract Interpretation

Type checking is done via an abstract interpretation using 6 arrows. All λ_{Ochre} syntax is defined for some subset of these arrows.

Almost all typing rules take the form $\Omega \vdash M \diamond v \dashv \Omega'$ where $\diamond$ is one of the below 6 arrows, M is Ochre syntax, Ω and Ω' are the abstract environments before and after, and m is the value which has been read or written.

	Read	Write
Destructive	$\overset{(\cdot)}{\Rightarrow}$	$\overset{(\cdot)}{\Leftarrow}$
	move	*write*
Non-destructive	$\overset{(\cdot)}{\rightarrow}$	$\overset{(\cdot)}{\leftarrow}$
	read	*type narrow*
Compile time only	$\rightsquigarrow$	$\leftsquigarrow$
	erased read	*erased write*

Explain all 6 by looking at the interpretations of the variable x.

3.4.1 Moving From Syntax

Destructive operations assert that a piece of syntax has a runtime influence on the data in memory, and updates the environment accordingly. As Ochre has Rust-like ownership semantics, so do these assertions; this means that reading from a variable will *move* the value out of wherever it was previously, and prevent it from being used again. For example $\{x \mapsto 5\} \vdash x \Rightarrow 5 \dashv \{x \mapsto \bot\}$ means when x is 5 in the environment, you can read a 5 from it, and can't read it again in the future. Writes are destructive because writing a value into memory requires the old value to be deallocated first. The full definition of $\Rightarrow$

Non-destructive operations

Compile-time only operations

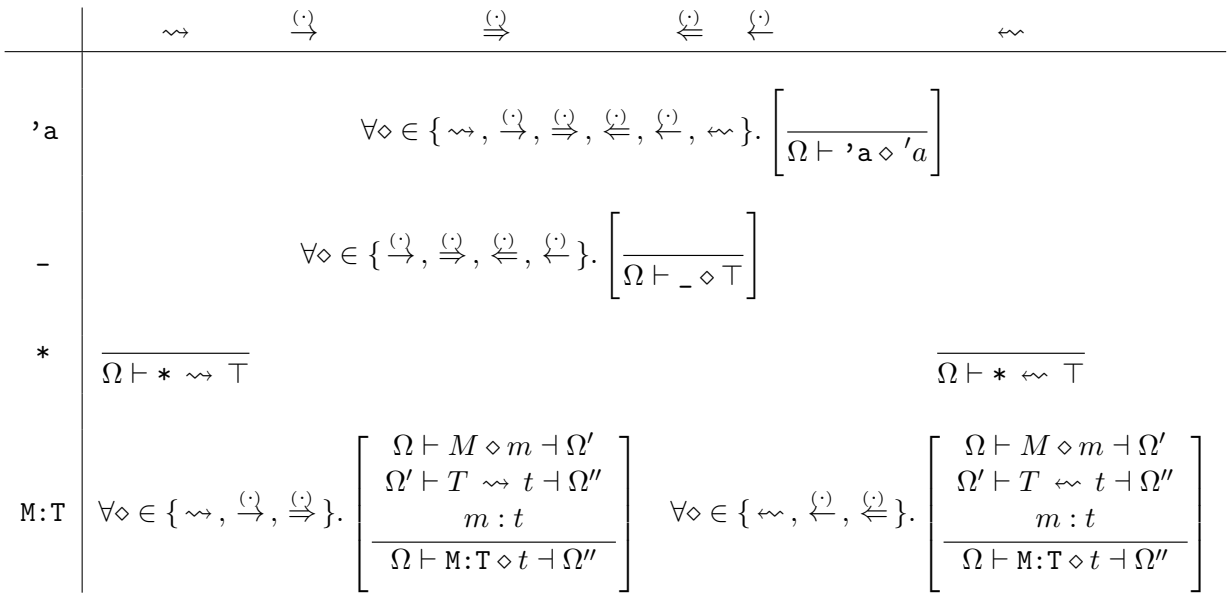

Figure 3.6: everythingable

Abstract Environment

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

Figure 3.5: Runtime state

Type Operations

3.4.2 Writing To Syntax

The $\overset{(\cdot)}{\Leftarrow}$ defines what it means to *write* to a given piece of syntax. This is how values get brought into the context, for instance, an assign (=) works by evaluating the RHS, and then writing it to the LHS, which is usually just a variable x, but in the case of destructuring could also be a pair (x, y).

	$\overset{(\cdot)}{\Rightarrow}$	$\overset{(\cdot)}{\to}$	$\rightsquigarrow$
`x` or `X`	$\dfrac{\Omega' = \Omega\left[\dfrac{\mathtt{x} \mapsto \bot}{\mathtt{x} \mapsto m}\right]}{\Omega \vdash \mathtt{x} \overset{(\cdot)}{\Rightarrow} m \dashv \Omega'}$	$\dfrac{\Omega' = \Omega\left[\dfrac{\mathtt{x} \mapsto m'}{\mathtt{x} \mapsto m}\right] \quad m : m'}{\Omega \vdash \mathtt{x} \overset{(\cdot)}{\to} m' \dashv \Omega'}$	$\dfrac{\Omega' = \Omega\left[\dfrac{\mathtt{X} \mapsto m'}{\mathtt{X} \mapsto m}\right] \quad m : m'}{\Omega \vdash \mathtt{X} \rightsquigarrow m' \dashv \Omega'}$
`M,N`	$\forall \diamond \in \{\overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\to}, \rightsquigarrow\}. \left[\dfrac{\Omega \vdash \mathtt{M} \diamond m \dashv \Omega' \quad \Omega' \vdash \mathtt{N} \diamond n \dashv \Omega''}{\Omega \vdash \mathtt{M,N} \diamond (m, n, \_ \to \_) \dashv \Omega''}\right]$		
`M.0`	$\dfrac{\Omega \vdash \mathtt{M} \overset{(\cdot)}{\Rightarrow} (v, \mathtt{T} \to \mathtt{U}) \dashv \Omega' \quad \Omega' \vdash \mathtt{T} \twoheadleftarrow v \dashv \Omega'' \quad \Omega'' \vdash \mathtt{U} \rightsquigarrow w \dashv \Omega' \quad \Omega' \vdash \mathtt{M} \overset{(\cdot)}{\Leftarrow} (\top, \_ \to w) \dashv \Omega'''}{\Omega \vdash \mathtt{M.0} \overset{(\cdot)}{\Rightarrow} m \dashv \Omega'''}$		$\forall \diamond \in \{\overset{(\cdot)}{\to}, \rightsquigarrow\}. \left[\dfrac{\Omega \vdash \mathtt{M} \diamond (m, \_) \dashv \Omega'}{\Omega \vdash \mathtt{M.0} \diamond m \dashv \Omega'}\right]$
`M.1`	$\forall \diamond \in \{\overset{(\cdot)}{\Rightarrow}, \overset{(\cdot)}{\to}, \rightsquigarrow\}. \left[\dfrac{\begin{array}{ll}\Omega \vdash \mathtt{M} \diamond (v, \mathtt{T} \to \mathtt{U}) \dashv \Omega' & /\!/ \textit{ get pair} \\ \Omega' \vdash \mathtt{T} \twoheadleftarrow v \dashv \Omega'' & /\!/ \textit{ calculate right restriction} \\ \Omega'' \vdash \mathtt{U} \rightsquigarrow w \dashv \Omega' & /\!/ \textit{ calculate right restriction} \\ \Omega' \vdash \mathtt{M} \overset{(\cdot)}{\Leftarrow} (m, \mathtt{T} \to \_) \dashv \Omega''' & /\!/ \textit{ replace pair} \quad \textit{if } \diamond = \overset{(\cdot)}{\Rightarrow} \\ \Omega''' = \Omega' & \textit{if } \diamond \neq \overset{(\cdot)}{\Rightarrow} \end{array}}{\Omega \vdash \mathtt{M.1} \diamond w \dashv \Omega'''}\right]$		
`*M`		$\dfrac{\Omega \vdash \mathtt{M} \overset{(\cdot)}{\to} \mathsf{borrow}^s\, l\, v}{\Omega \vdash \mathtt{*M} \overset{(\cdot)}{\to} v}$	
	$\dfrac{\Omega \vdash \mathtt{M} \overset{(\cdot)}{\Rightarrow} \mathsf{borrow}^m\, l\, v \dashv \Omega' \quad \Omega' \vdash \mathtt{M} \overset{(\cdot)}{\Leftarrow} \mathsf{borrow}^m\, l\, \bot \dashv \Omega''}{\Omega \vdash \mathtt{*M} \overset{(\cdot)}{\Rightarrow} v \dashv \Omega''}$	$\dfrac{\Omega \vdash \mathtt{M} \overset{(\cdot)}{\to} \mathsf{borrow}^m\, l\, v}{\Omega \vdash \mathtt{*M} \overset{(\cdot)}{\to} v}$	

Figure 3.7: readtable

	$\overset{(\cdot)}{\Leftarrow}$	$\overset{(\cdot)}{\leftarrow}$	$\leftsquigarrow$
x or X	$\dfrac{\Omega' = \Omega\left[\dfrac{\mathtt{x} \mapsto m}{\mathtt{x} \mapsto \bot}\right]}{\Omega \vdash \mathtt{x} \overset{(\cdot)}{\Leftarrow} m \dashv \Omega'}$	$\dfrac{\Omega' = \Omega\left[\dfrac{\mathtt{x} \mapsto m'}{\mathtt{x} \mapsto m}\right] \quad m' : m}{\Omega \vdash \mathtt{x} \overset{(\cdot)}{\leftarrow} m' \dashv \Omega'}$	$\dfrac{\Omega' = \Omega\left[\dfrac{\mathtt{X} \mapsto m'}{\mathtt{X} \mapsto m}\right] \quad m' : m}{\Omega \vdash \mathtt{X} \leftsquigarrow m' \dashv \Omega'}$
M,N	$\forall \diamond \in \{ \overset{(\cdot)}{\Leftarrow}, \overset{(\cdot)}{\leftarrow}, \leftsquigarrow \}. \left[\dfrac{\begin{array}{c}\Omega \vdash \mathtt{M} \diamond m \dashv \Omega' \\ \Omega' \vdash \mathtt{T} \leftsquigarrow m \dashv \Omega'' \\ \Omega' \vdash \mathtt{U} \rightsquigarrow n' \dashv \Omega' \\ \Omega' \vdash \mathtt{N} \diamond n \cap n' \dashv \Omega'''\end{array}}{\Omega \vdash \mathtt{M},\ \mathtt{N} \diamond (m, n, \mathtt{T} \to \mathtt{U}) \dashv \Omega'''}\right]$		
M.0	$\dfrac{\Omega \vdash \mathtt{M} \overset{(\cdot)}{\Leftarrow} (\bot, n, \_ \to \mathtt{U}) \dashv \Omega' \quad \Omega' \vdash \mathtt{M} \overset{(\cdot)}{\Leftarrow} (m, n, \mathtt{T} \to \mathtt{U}) \dashv \Omega''}{\Omega \vdash \mathtt{M.0} \overset{(\cdot)}{\Leftarrow} m \dashv \Omega''}$	$\forall \diamond \in \{ \overset{(\cdot)}{\leftarrow}, \leftsquigarrow \}. \left[\dfrac{\Omega \vdash \mathtt{M} \diamond (\bot, n, \_ \to \mathtt{U}) \dashv \Omega'}{\Omega \vdash \mathtt{M.0} \diamond m \dashv \Omega''}\right]$	
M.1	$\dfrac{\Omega \vdash \mathtt{M} \overset{(\cdot)}{\Leftarrow} (m, \bot, \mathtt{T} \to \_) \dashv \Omega' \quad \mathtt{U} = \mathsf{translate}\, n \quad \Omega' \vdash \mathtt{M} \overset{(\cdot)}{\Leftarrow} (m, n, \mathtt{T} \to \mathtt{U}) \dashv \Omega''}{\Omega \vdash \mathtt{M.1} \overset{(\cdot)}{\Leftarrow} n \dashv \Omega''}$		
_	$\dfrac{\Omega \vdash \mathsf{drop}\, v \dashv \Omega'}{\Omega \vdash \_ \overset{(\cdot)}{\Leftarrow} v \dashv \Omega'}$	$\forall \diamond \in \{ \overset{(\cdot)}{\leftarrow}, \leftsquigarrow \}. \left[\dfrac{}{\Omega \vdash \_ \overset{(\cdot)}{\rightarrow} \bot}\right]$	
*			
*M	$\dfrac{\Omega \vdash \mathtt{M} \overset{(\cdot)}{\Rightarrow} \mathsf{borrow}^m\, l \bot \dashv \Omega' \quad \Omega' \vdash \mathtt{M} \overset{(\cdot)}{\Leftarrow} \mathsf{borrow}^m\, l\, v \dashv \Omega''}{\Omega \vdash *\mathtt{M} \overset{(\cdot)}{\Leftarrow} v \dashv \Omega''}$	$\dfrac{\Omega \vdash \mathtt{M} \overset{(\cdot)}{\rightarrow} \mathsf{borrow}^s\, l\, v' \dashv \Omega' \quad \Omega'' = \Omega'[\mathsf{loan}^s\, l\, v'/\mathsf{loan}^s\, l\, v] \quad v' : v}{\Omega \vdash *\mathtt{M} \overset{(\cdot)}{\leftarrow} v' \dashv \Omega'}$	

Figure 3.8: writetable

	$\overset{(\cdot)}{\Rightarrow}$	$\rightsquigarrow$
`&M`	$\dfrac{\Omega \vdash \texttt{M} \overset{(\cdot)}{\to} m \quad \Omega \vdash \texttt{M} \overset{(\cdot)}{\Leftarrow} \mathsf{loan}^s\, l\, m \dashv \Omega'}{\Omega \vdash \texttt{\&M} \overset{(\cdot)}{\Rightarrow} \mathsf{borrow}^s\, l\, m \dashv \Omega'}$	
`&mut M`	$\dfrac{\Omega \vdash \texttt{M} \overset{(\cdot)}{\Rightarrow} m \dashv \Omega' \quad \Omega' \vdash \texttt{M} \overset{(\cdot)}{\Leftarrow} \mathsf{loan}^m\, l \dashv \Omega''}{\Omega \vdash \texttt{\&mut M} \overset{(\cdot)}{\Rightarrow} \mathsf{borrow}^m\, l\, m \dashv \Omega''}$	
`M N`	$\dfrac{\begin{array}{ll}\Omega \vdash \texttt{F} \overset{(\cdot)}{\to} (\texttt{T} \to \texttt{U}) & /\!/ \textit{ eval function} \\ \Omega \vdash \texttt{X} \overset{(\cdot)}{\Rightarrow} v \dashv \Omega' & /\!/ \textit{ eval argument} \\ \Omega' \vdash \texttt{T} \leftsquigarrow v \dashv \Omega'' & /\!/ \textit{ calculate return type} \\ \Omega'' \vdash \texttt{U} \rightsquigarrow w \dashv \Omega' & /\!/ \textit{ calculate return type} \\ \text{maximally widen } v \text{ to } v' \text{ such that } T \leftsquigarrow v' & \\ \Omega' \vdash \text{drop } v' \dashv \Omega''' & /\!/ \textit{ propogate side effects}\end{array}}{\Omega \vdash \texttt{F X} \overset{(\cdot)}{\Rightarrow} w \dashv \Omega'''}$	$\dfrac{\begin{array}{ll}\Omega \vdash \texttt{F} \rightsquigarrow (\texttt{T} \to \texttt{U}) & /\!/ \textit{ eval function} \\ \Omega \vdash \texttt{X} \rightsquigarrow v \dashv \Omega' & /\!/ \textit{ eval argument} \\ \Omega' \vdash \texttt{T} \leftsquigarrow v \dashv \Omega'' & /\!/ \textit{ calculate return} \\ \Omega'' \vdash \texttt{U} \rightsquigarrow w \dashv \Omega' & /\!/ \textit{ calculate return}\end{array}}{\Omega \vdash \texttt{F X} \rightsquigarrow w \dashv \Omega'}$
`M -> T {N}`	$\dfrac{\begin{array}{ll}\Omega \vdash \texttt{M} \leftsquigarrow m \dashv \Omega' & /\!/ \textit{ calculate widest input} \\ \Omega' \vdash \texttt{T} \rightsquigarrow t & /\!/ \textit{ calculate return type} \\ \Omega' \vdash \texttt{N} \overset{(\cdot)}{\Rightarrow} n \dashv \Omega & /\!/ \textit{ body must reset environment} \\ n : t & /\!/ \textit{ check body}\end{array}}{\Omega \vdash \texttt{M -> T \{ N \}} \overset{(\cdot)}{\Rightarrow} \texttt{T} \to \texttt{U}}$	$\dfrac{\begin{array}{ll}\Omega \vdash \texttt{T} \leftsquigarrow t \dashv \Omega' & /\!/ \textit{ get input type} \\ \Omega' \vdash \texttt{U} \rightsquigarrow u \dashv \Omega & /\!/ \textit{ check it works with body}\end{array}}{\Omega \vdash \texttt{T -> U} \rightsquigarrow \texttt{T} \to \texttt{U}}$
`M = N`	$\dfrac{\Omega \vdash \texttt{N} \overset{(\cdot)}{\Rightarrow} v \dashv \Omega' \quad \Omega' \vdash \texttt{M} \overset{(\cdot)}{\Leftarrow} v \dashv \Omega''}{\Omega \vdash \texttt{M = N} \overset{(\cdot)}{\Rightarrow} 'unit \dashv \Omega''}$	$\dfrac{\Omega \vdash \texttt{N} \rightsquigarrow v \dashv \Omega' \quad \Omega' \vdash \texttt{M} \leftsquigarrow v \dashv \Omega''}{\Omega \vdash \texttt{M = N} \overset{(\cdot)}{\Rightarrow} 'unit \dashv \Omega''}$
`M;N`	$\dfrac{\Omega \vdash \texttt{M} \overset{(\cdot)}{\Rightarrow} 'unit \dashv \Omega' \quad \Omega' \vdash \texttt{N} \overset{(\cdot)}{\Rightarrow} n \dashv \Omega''}{\Omega \vdash \texttt{M;N} \overset{(\cdot)}{\Rightarrow} n \dashv \Omega''}$	
`case M { 'a`$_0$ `=> N`$_0$`, ... 'a`$_k$ `=> N`$_k$`, }`	$\forall \diamond \in \{\overset{(\cdot)}{\Rightarrow}, \rightsquigarrow\}. \left[\dfrac{\begin{array}{ll}\Omega \vdash \texttt{M} \diamond \{'a_1, ..., 'a_k\} \dashv \Omega' & /\!/ \textit{ eval interogant} \\ \forall i. [\Omega' \vdash \texttt{M} \overset{(\cdot)}{\Leftarrow} 'a_i \dashv \Omega_i'] & /\!/ \textit{ narrow type in branch} \\ \forall i. [\Omega_i' \vdash \texttt{N}_i \diamond n_i \dashv \Omega_i''] & /\!/ \textit{ eval branch} \\ n = n_0 \cup ... \cup n_k & /\!/ \textit{ combine branch values} \\ \Omega'' = \Omega_0'' \cup ... \cup \Omega_k'' & /\!/ \textit{ combine branch side effects}\end{array}}{\Omega \vdash \texttt{case M \{ 'a}_0 \texttt{ => N}_0\texttt{, ... \}} \diamond n \dashv \Omega''}\right]$	
`T | U`		$\dfrac{\Omega \vdash \texttt{T} \rightsquigarrow t \dashv \Omega' \quad \Omega' \vdash \texttt{U} \rightsquigarrow u \dashv \Omega''}{\Omega \vdash \texttt{T | U} \rightsquigarrow t \cup u \dashv \Omega''}$

Figure 3.9: readonlytable

3.4.3 Reading from syntax

The $\overset{(\cdot)}{\to}$ defines what it means to read from a piece of syntax. Judgments of this form turn into memory reads at runtime, they can follow references and look into pairs.

While most of the time this is used for reading, it can also be used to do something called type *widening*. $\mathtt{x} \overset{(\cdot)}{\to} 5$ means 5 can be read from x, but if 5 can be read from x, so can $\mathbb{N}$, because 5 is a subtype of $\mathbb{N}$.

It's called type widening because it widens the type in the environment, instead of just reading a wider type while leaving the original alone. So $\{\mathtt{x} \mapsto 5\} \vdash \mathtt{x} \overset{(\cdot)}{\to} \mathbb{N} \dashv \{\mathtt{x} \mapsto \mathbb{N}\}$.

If you encounter a judgment like $\Omega \vdash \mathtt{x} \overset{(\cdot)}{\to} 5$, it means the environment hasn't changed (i.e. $\Omega \vdash \mathtt{x} \overset{(\cdot)}{\to} 5 \dashv \Omega$), so you know the value of x has just been read, instead of being widened in place.

The definition of $\overset{(\cdot)}{\to}$ for all pieces of syntax is shown in Figure **??**.

3.4.4 Type Narrowing

The $\overset{(\cdot)}{\leftarrow}$ constrains the type of syntax down to a smaller type. This is primarily useful when type-checking case expressions. When type-checking a particular branch of a case expression, you can modify the environment to reflect the fact that you know which branch you're in and therefore what the value of the case expression is. This is particularly useful when the type of the right-hand side of a pair depends on the left-hand value, because you can case the left-hand value, and in each branch you will know the type of the right. This is how ADTs are implemented in Ochre. It's definition is given in Figure **??**.

3.5 Scope and Feature Set

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

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

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

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

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

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

Reasoning about function side effects/strong updates - In Ochre, if a function takes a mutable reference to a value of type T, the value is guaranteed to still be of type T after the function return. You may want this not to be the case if the type encodes some property of your data structure, for instance, if you have a type for lists and another for sorted lists you may want an in-place sorting algorithm to change the type of the referenced list into a sorted list. I choose to not support this for a few reasons:

1. I predict that it will be idiomatic in Ochre to separate data structures from proofs about their structure. If this is the case, you could return a proof about one of your inputs, which immutably borrows that input, causing it to be invalidated if the data structure is ever mutated. This would not involve strong updates.

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

3. People can still do strong updates by moving the data structure in and out of a function instead of giving it a borrow. This is even possible if the caller only has a mutable reference to the data because strong updates are allowed locally.

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

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 4

Evaluation

If this project works, I should be able to type-check a program with both mutability and dependent types and reject it if the mutation is done incorrectly. Which examples exactly I will test is yet to be determined, but it will be something along the following lines, probably with a Σ type.

```
ResizeableArray: Type = (n : Nat, Vec(Int, n))
v: ResizeableArray = (0, ());

push(&mut v, 42); // v == (1, (42))
push(&mut v, 42); // v == (2, (42, 42))

double(&mut v); // v == (4, (42, 42, 42, 42))
```

This would only type check if `push` and `double` are implemented correctly, which involves them correctly keeping the length variable up to date (left-hand element of pair).

Chapter 5

Ethical Issues

I do not foresee any ethical issues arising from this project.

Bibliography

ATS-Home, a. URL https://www.cs.bu.edu/~hwxi/atslangweb/Home.html. pages 11

ATS-Implements, b. URL https://www.cs.bu.edu/~hwxi/atslangweb/Implements.html. pages 11

The Rust Programming Language - The Rust Programming Language. URL https://doc.rust-lang.org/book/title-page.html. pages 17

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

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 15

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

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

Son Ho and Jonathan Protzenko. Aeneas: Rust Verification by Functional Translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP):711–741, August 2022. ISSN 2475-1421. doi: 10.1145/3547647. URL http://arxiv.org/abs/2206.07185. pages 10, 13, 41

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

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

Sebastian Ullrich. Kha/electrolysis, January 2024. URL https://github.com/Kha/electrolysis. pages 41

Appendix A

Formal Verification using (Dependent) Types

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

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

Boolean Tautologies in Haskell

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

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

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

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

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

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

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

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

Dependent Types are Quantifiers

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

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