Imperial College London

MASTERS THESIS

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

May 27, 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.

Contents

Chapter 1

Introduction

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 **??**.

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 **??** 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 **??**.

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:

[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.

```
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.

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 $\rightarrow$ 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 **??**.

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 [**?**] [3]. In this research I don't intend to support any constructs not already supported by either of these prior works, so I can use the translation algorithms they have already developed.

2.2 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 [**?**]. As its website states, it is a *statically typed programming language that unifies implementation with formal specification* [**?**].

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*.

[2] See [**?**] 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 [**?**] 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.

For instance, the `ptr_get0` function has the type $\{l : addr\}(T@l|ptr(l)) \to (T@l|T)$ 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[**?**] 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

λ_{Ochre}

At its core, Ochre is an imperative language with functions, mutation, pointers, and user-defined types. Every value in Ochre at runtime is either an atom (more on this below), a dependent pair, a dependent function, or a pointer to one of the aforementioned. Algebraic data types like you would find in Haskell or Rust are defined using a composition of the above features.

This chapter introduces λ_{Ochre}'s features one by one with examples, skip to Figure **??** if you just want the syntax.

Atoms

Pairs

Variables

References (Pointers)

Functions

yadayada

λ_{Ochre} was originally based upon Aeneas's [**?**] LLBC, however over time so many features have been added and removed it has diverged

In order to achieve the desired properties, λ_{Ochre} must have dependent functions, dependent pairs, Rust-like ownership semantics, user-defined data types, and in-place mutation. I took Aeneas's LLBC as a starting point, then gradually added in concepts from $\Pi\Sigma$[**?**]. Where possible, I simplified the language down,

3.1 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 [**?**][**?**]. 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.

3.2 Syntax

λ_{Ochre} has an extremely permissive syntax, and heavily relies on the typing rules to reject ill-formed programs. This leads to some usual results like `if 'a {'b} else {'c} = 'hello` being syntactically well-formed. In practice, this doesn't cause issues, because the typing rules are strict enough. Figure **??**

$T, U, M, N ::=$	
$x \mid y \mid z$	runtime variable identifier
$X \mid Y \mid Z$	comptime variable identifier
$*M$	dereference
MN	function application
$M \to T\,(\{\, N \,\})$	(dependent) function definition (optional runtime body)
M, N	pair construction
$M = N$	assignment
$'a$	atom construction
$T \mid U$	type union
$M; N$	sequence
$\mathsf{case}\, M\, \{\, \overrightarrow{'a \Rightarrow N} \,\}$	case statement
$\&M \mid \&\mathsf{mut}\, M$	borrow constructor
$M : T$	type constraint
$*$	top
$\bot$	uninitialised

Figure 3.1: λ_{Ochre} syntax

3.3 Type System

3.4 Concrete Semantics

3.5 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	$\dot{\Rightarrow}$	$\dot{\Leftarrow}$
	move	*write*
Non-destructive	$\dot{\to}$	$\dot{\leftarrow}$
	read	*type narrow*
Compile time only	$\rightsquigarrow$	$\leftsquigarrow$
	erased read	*erased write*

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

VAR-MOVE

$$\frac{\Omega' = \Omega[\mathtt{x} \mapsto \bot / \mathtt{x} \mapsto m]}{\Omega \vdash \mathtt{x} \Rightarrow m \dashv \Omega'}$$

PAIR-LEFT-MOVE

$$\frac{\Omega \vdash \mathtt{M} \Rightarrow (m, n, \mathtt{T} \to \mathtt{U}) \dashv \Omega' \qquad \Omega' \vdash \mathtt{M} \Leftarrow (\bot, n, \_ \to \mathtt{U}) \dashv \Omega''}{\Omega \vdash \mathtt{M.0} \Rightarrow m \dashv \Omega''}$$

PAIR-RIGHT-MOVE

$$\frac{\begin{array}{rl} \Omega \vdash \mathtt{M} \Rightarrow (m, n, \mathtt{T} \to \mathtt{U}) \dashv \Omega' & \textit{get pair} \\ \Omega' \vdash \mathtt{T} \twoheadleftarrow m \dashv \Omega'' & \textit{calculate right restriction} \\ \Omega' \vdash \mathtt{U} \rightsquigarrow u \dashv \Omega'' & \textit{calculate right restriction} \\ \Omega'' \vdash \mathtt{M} \Leftarrow (m, \bot, \mathtt{T} \to \_) \dashv \Omega''' & \textit{return pair} \end{array}}{\Omega \vdash \mathtt{M.1} \Rightarrow n \cap u \dashv \Omega'''}$$

MUT-REF-MOVE

$$\frac{\Omega \vdash \mathtt{M} \Rightarrow \mathsf{borrow}^m\, l\, v \dashv \Omega' \qquad \Omega' \vdash \mathtt{M} \Leftarrow \mathsf{borrow}^m\, l\, \bot \dashv \Omega''}{\Omega \vdash *\mathtt{M} \Rightarrow v \dashv \Omega''}$$

FUNCTION-CALL

$$\frac{\begin{array}{rl} \Omega \vdash \mathtt{F} \dot{\to} (\mathtt{T} \to \mathtt{U}) & \textit{eval function} \\ \Omega \vdash \mathtt{X} \Rightarrow v \dashv \Omega' & \textit{eval argument} \\ \Omega' \vdash \mathtt{T} \twoheadleftarrow v \dashv \Omega'' & \textit{calculate return type} \\ \Omega'' \vdash \mathtt{U} \rightsquigarrow w \dashv \Omega' & \textit{calculate return type} \\ \text{maximally widen } v \text{ to } v' \text{ such that } T \twoheadleftarrow v' & \\ \Omega' \vdash \text{drop}\, v' \dashv \Omega''' & \textit{propogate side effects} \end{array}}{\Omega \vdash \mathtt{F\ X} \Rightarrow w \dashv \Omega'''}$$

FUNCTION-DEFINITION

$$\frac{\begin{array}{rl} \Omega \vdash \mathtt{M} \twoheadleftarrow m \vdash \Omega' & \textit{calculate widest input} \\ \Omega' \vdash \mathtt{T} \rightsquigarrow t & \textit{calculate return type} \\ \Omega' \vdash \mathtt{N} \Rightarrow n \dashv \Omega & \textit{body must reset environment} \\ n : t & \textit{body must be a subtype of return type} \end{array}}{\Omega \vdash \mathtt{M\ ->\ T\ \{\ N\ \}} \Rightarrow \mathtt{T} \to \mathtt{U}}$$

PAIR-CONSTRUCTION

$$\frac{\Omega \vdash \mathtt{M} \Rightarrow m \dashv \Omega' \qquad \Omega' \vdash \mathtt{N} \Rightarrow n \dashv \Omega''}{\Omega \vdash \mathtt{M,\ N} \Rightarrow (m, n, \_ \to \_) \dashv \Omega''}$$

ASSIGNMENT

$$\frac{\Omega \vdash \mathtt{N} \Rightarrow v \dashv \Omega' \qquad \Omega' \vdash \mathtt{M} \Leftarrow v \dashv \Omega''}{\Omega \vdash \mathtt{M\ =\ N} \Rightarrow {}'unit \dashv \Omega''}$$

ATOM-CONSTRUCTION

$$\frac{}{\Omega \vdash \mathtt{'a} \Rightarrow {}'a}$$

SEQUENCE

$$\frac{\Omega \vdash \mathtt{M} \Rightarrow m \dashv \Omega' \qquad \Omega' \vdash \text{drop}\, m \dashv \Omega'' \qquad \Omega'' \vdash \mathtt{N} \Rightarrow n \dashv \Omega'''}{\Omega \vdash \mathtt{M;\ N} \Rightarrow n \dashv \Omega'''}$$

CASE

$$\frac{\begin{array}{rl} \Omega \vdash \mathtt{M} \Rightarrow m \dashv \Omega' & \textit{eval interogant} \\ \forall i. [\Omega' \vdash \mathtt{M} \dot{\leftarrow} {}'a_i \dashv \Omega'_i] & \textit{narrow type in branch} \\ \forall i. [\Omega'_i \vdash \mathtt{N} \Rightarrow n_i \dashv \Omega''_i] & \textit{eval branch} \\ n = n_0 \cup ... \cup n_k & \textit{combine branch values} \\ \Omega'' = \Omega''_0 \cup ... \cup \Omega''_k & \textit{combine branch side effects} \end{array}}{\Omega \vdash \mathtt{case\ M\ \{\ 'a0\ =>\ N0,\ ...\ \}} \Rightarrow n \dashv \Omega''}$$

IMMUTABLE-BORROW

$$\frac{\Omega \vdash \mathtt{M} \dot{\to} m \qquad \Omega \vdash \mathtt{M} \dot{\leftarrow} \mathsf{loan}^s\, l\, m \dashv \Omega'}{\Omega \vdash \mathtt{\&M} \Rightarrow \mathsf{borrow}^s\, l\, m \dashv \Omega'}$$

MUTABLE-BORROW

$$\frac{\Omega \vdash \mathtt{M} \Rightarrow m \dashv \Omega' \qquad \Omega' \vdash \mathtt{M} \Leftarrow \mathsf{borrow}^m\, l \dashv \Omega''}{\Omega \vdash \mathtt{\&mut\ M} \Rightarrow \mathsf{borrow}^m\, l\, m \dashv \Omega''}$$

$$\Omega \vdash \mathtt{M\ :\ T} \Rightarrow m \dashv \Omega \text{TYPE-CONSTRAINT-MOVE}$$

Figure 3.2: Definition of $\Rightarrow$/move/destructive read

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.

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.