Imperial College London

INTERIM REPORT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

A Type Checker With Support For Mutability And Dependent Types

Author:
Charlie Lidbury

Supervisor(s):
Steffen van Bakel
Nicolas Wu

January 24, 2024

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

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 2.

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, which is discussed in [ref].

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. .

Example of Rust doing t better

Because this translation has been shown to be behaviour preserving [cite: Aeneas], 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`.

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:

[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 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 complicated features like for loops and functions which return mutable references. So much so safe Rust isn't even entirely covered by the two main attempts at this translation Electrolysis [] and Aeneas [1]

2.2 Related Work

With a goal as broad as *make verification of systems code easier* OUTDATED, I'm operating in a relatively mature research space, with significant industry interest. In this section, I lay out the various approaches researchers have taken to get verified low-level code.

2.2.1 Systems Theorem Provers

There have already been a couple of attempts to make a programming language which is both a systems language and a theorem prover.

ATS

ATS [**?**] is the most mature systems programming language to date, with work dating back to 2002 [2]. As its website states, it is a *statically typed programming language that unifies implementation with formal specification* [3].

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 $\{l : addr\}(T@l|ptr(l)) \to (T@l|T)$ 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.

2.2.2 Extract-to-C

These approaches work by expressing your program in some theorem prover like F* or Coq, which outputs some efficient code that can be executed like C source code.

Low*

Low* is a subset of another language, F*, which can be extracted into C via a transpiler called KreMLin.

2.2.3 Modelling a Conventional Program

Rust Belt?

Chapter 3

Project Plan

Chapter 4

Evaluation Plan

Chapter 5

Ethical Issues

Bibliography

[1] Ho S, Protzenko J. Aeneas: Rust Verification by Functional Translation. Proceedings of the ACM on Programming Languages. 2022 Aug;6(ICFP):711-41. pages 11

[2] ATS-Implements;. https://www.cs.bu.edu/~hwxi/atslangweb/Implements.html. pages 11

[3] ATS-Home;. https://www.cs.bu.edu/~hwxi/atslangweb/Home.html. pages 11

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.