borrow-lambda

Taran Dwivedula, Jeff Zhang

April 20, 2025

Abstract

Borrow checking and ownership is a solution to many of the problems that commonly arise in programming languages with manual memory management like C. Like the strategy of garbage collection implemented in several common general purpose programming languages, borrow checking eliminates the need for programmers to manually allocate and free the memory their program uses, which is the source of many human errors in programming. However, the need for high performance code makes garbage collection a sub-optimal solution, as garbage collection introduces performance overhead, non-determinism, and potentially memory fragmentation. Borrow checking and ownership is an alternative that takes a different approach - it relies on the strength of the programming language's type system. It is a set of rules that allows the compiler to automatically determine where to allocate and free memory, both eliminating the need of a garbage collector and the need for a programmer to manually allocate and free memory. In this paper, we discuss borrow checking as a solution to common memory errors, and provide an implementation on a small extension of the simply-typed lambda calculus.

1 Introduction

Systems-level programming mandates high performance code; that is, code that runs with minimal overhead, and uses memory efficiently. Minimizing fragmentation on the heap is essential, especially for servers and embedded systems. It's why a programming language like C has been very popular in this domain for decades. C is an abstraction directly from assembly that was designed to make code easier to read and write. As such, control of heap allocated memory remains fully with the programmer. The practice of writing good and safe code is left to the programmer - the idea being, a skilled programmer would be able to write memory safe code without mistakes, and just like assembly, mistakes would not result in errors - just undefined behaviour.

Over time, we have seen that relying on programmers to be skilled enough to avoid making memory errors in their code is not feasible. As codebases grow larger and larger, these errors become extremely hard to avoid. Computer viruses and exploits due to things like buffer overflows affect many people and are still present in a large amount of software today.

For general purpose programming, the concept of a garbage collector was able to solve this problem. A garbage collector finds areas of memory that are no longer referenced at some point in the program, and frees the memory. Different strategies exist, such as reference counting, stop and copy, and mark and sweep strategies. Usually, the garbage collector is required to run as a separate process, adding performance overhead. Some strategies of garbage collection allow the garbage collector to run concurrent to the program, but fragment the memory. Other strategies avoid memory fragmentation, but require the running program to be paused so the memory can be copied over in a non-fragmented fashion. In any case, garbage collection results in the overhead of another process, as well as sometimes non-determinism in the layout of memory, both of which are extremely sub-optimal for high performance systems code.

Work has been done to improve manually memory managed languages like C in terms of safety, by strengthening the typesystem itself, and attacking the semantics of the language. Cyclone, a Cornell lab project, developed a dialect of C that works to ensure memory safety at compile time. Among the features included were statically checked lifetimes of pointer variables, and restrictions on types of pointers which

work to ensure safety with regards to multi-aliasing [4]. Similarly, the concept of ownership types was conceived as a software design principle, particularly in research on concurrent programming, where an object governs an area of memory, and administers any modifications to it [1] [3]. In the modern day, many of these features were materialized in the Rust programming language, a popular systems-level programming language that is making the case for the assistance of borrow checking and ownership for high performance code.

In this paper, we implement borrow checking and ownership on a small extension of the simply typed lambda calculus. The paper is organized as follows. Section 2 provides an overview of the components of the borrow checking and ownership strategy. Section 3 discusses our implementation of a borrow checker in `borrow-lambda`. Section 4 describes some of the main challenges we faced when developing this implementation, and our solutions to them. Section 5 concludes the discussion of our implementation and discusses potential further improvements. Section 6 gives an overview of related work in the literature.

2 Overview

The borrow checking and ownership strategy can be divided into three separate components that work together - Ownership, Borrowing, and Lifetimes. This section will conceptually describe the three. As the Rust programming language is the most well known widespread implementation of these concepts, we will reference it for illustrative purposes.

2.1 Ownership

Rust implements ownership as described with three main rules. [6]

- 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.

In reality, this is an instance of an *affine type system* - an instance of a substructural type system based on substructural logic where the rule of contraction is not permitted. In code, this corresponds to the language's move semantics - if a value isn't given the ability to be copied, it is instead *moved*. Ownership and scope of the value is transferred to the new binder. This allows the compiler to statically determine where the memory of the value should be freed, and where the value, as well as names tied to it, are allowed to be used, absolving this responsibility from the programmer. Figure 1 illustrates an example. We create

```
fn main() {
    let nums = vec![1, 2, 3];

    do_something_with_vec(nums);
    do_something_with_vec(nums);  // error: use of moved value
}

fn do_something_with_vec(v: Vec<u32>) {}
```

Figure 1: Example Rust code illustrating move semantics

a vector of unsigned integers - an object whose type does not permit copying - and bind it to the variable `nums`. We say the variable `nums` *owns* this vector and its associated memory. Up to this point, we should expect `nums` to go out of scope at the end of `main`, and the compiler would automatically free the memory at this time. However, we instead pass `nums` as a parameter into the function `do_something_with_vec`. Then,

as previously described, the vector gets *moved* into `do_something_with_vec`. The vector is no longer bound to the name `nums`, which causes the second call to `do_something_with_vec` to fail. Additionally, as the new owner, the function call administers the scope of this vector as well. As such, the vector's memory gets freed at the end of the function call. At this point, all access points to this memory are unusable and trigger compile time errors, which is what we want.

What if we did actually want to use the same vector twice inside of `main`? Currently, it is quite inconvenient that the vector gets detached from the `nums` variable after the first use of it. We may rewrite the code such that `do_something_with_vec` returns the vector, passing ownership of it back to the caller, allowing us to re-bind the vector to the `nums` variable. That is one possible solution, but what we really want is to provide some sort of reference, rather than temporarily giving ownership and taking it back. Crucially, it eliminates one nice guarantee we can make, which is the state of the value before and after we lend it. It would be nice to know, at a type level, whether the value we lent was mutated before we received it back. This is made possible with borrowing, which we describe in the next subsection.

2.2 Borrowing

Borrowing is a form of restricted pointers that allows us to give access of a value in memory, without relinquishing ownership of the value. Access is given through references to the value. These references are set to be either immutable or mutable by the programmer, meaning that references may only provide a view on the data without providing permission to mutate the data. Enforcement is an easy check during compile time.

The concept of mutable and immutable references is simple enough, but what makes them useful is the rules that determine when they are valid, and how they affect the access points of memory. Particularly, the following rules related to references are enforced at compile time.

- At any given time, a value can have either one mutable reference pointing to it, or any amount of immutable references pointing to it, but not both.
- The owner of a value cannot mutate or give up ownership of a value while any references to the value are in scope.

Again, both easy checks at compile time. It is easily seen that these two restrictions ensure two important conditions, that is, only one object is able to mutate a value at a time, and that values will not be mutated or freed while being read from. In other words, this eliminates the problem of mutable aliasing.

2.3 Lifetimes

It is important to know how long a value is valid for, before its associated memory is freed or repurposed. Being able to check this at compile time is a large component of creating a memory safe language. The aformentioned Cyclone first experimented with this by adding regions to C, and it is the basis of Rust's lifetimes feature. A potential, and extremely common error, is that we may have a pointer to some value that we continue to use even after the value's associated memory gets freed - a dangling pointer.

With the groundwork of ownership and borrowing laid out, the solution to this is simple - when we borrow a value, we must ensure that the value's associated memory does not get freed while a reference to it is still in scope. Or rather, we should not allow a reference to be created if its scope exceeds the window for which the value's memory remains intact. We formalize this with lifetimes.

Generally, the *lifetime* of a value, or the window in which its memory can be used before it is freed, is taken from the scope of its owner. These lifetimes are embedded into the typesystem as any references are additionally tagged with the lifetime of the value it is pointing to. Then, these references are checked with a subtyping relation, where a reference with a larger scope is a subtype of a reference with a smaller scope. This allows us to check that all borrows, or references, go out of scope before the value that they are borrowing is freed.

Lastly, functions that accept references as input, and return references, need to ensure that these references have a scope that is larger than the function itself. However, the scope that these references have need not be fixed - they just need to be sufficiently large. As a result, functions are actually polymorphic over lifetime variables, provided that the instantiation lifetime variable meets the constraints we have discussed. Figure 2 shows an example. The function `borrower` requires that the reference passed in, and the reference

```
fn main() {
    let n = 0;
    let my_ref = borrower(&n);
}

fn borrower<'a>(r: &'a u32) -> &'a u32 {
    let b = 0;
    &b // error: variable 'b' does not live long enough
}
```

Figure 2: Example of a function with a polymorphic lifetime variable

returned out, has a lifetime `'a`. When the function is called, this variable `'a` is instantiated to the lifetime of the reference that is passed in, namely `&n`. Thus, the typechecker can determine that the implementation of the function is invalid, as the reference `&b` is being returned out of the function, but has a lifetime smaller than `'a`. This is because any possible instantiation of `'a` must must be strictly larger than the scope of the function since it is tied to a parameter.

3 Implementation

Our implementation consists of a small extension to the simply-typed lambda calculus, with constructs that are both necessary and illustrative to borrow checking and ownership. The implementation is written in OCaml, with each of the constructs encoded as OCaml types. The main relevant component is the typechecker, which runs in two phases that will be described in this section. Additionally, we have included a parser which collects source code into terms represented as nodes in an abstract syntax tree, and an evaluator in order to develop a fully functional REPL. The evaluator follows a simple call by value semantics.

3.1 Syntax

In addition to standard natural numbers and booleans and some simple operations for them, we add mutable and immutable references, the dereference operator, assignment and dereference-assignment, as well as a vector of nats for a more complex (non-copyable) data structure.

We add terms representing operations that create a vector given a list of terms, get a reference (mutable or immutable) of an element of a vector given a reference to the vector and an index, a similar method for pushing an element to a vector, and a method for popping an element (and returning the actual element as opposed to just a reference of it). Any term can also be given a type annotation. Note that in this implementation, all variables are mutable.

The syntax of terms is as follows:

$$
\begin{array}{rrcl}
\textsf{Term} & t & ::= & x \\
& & | & \lambda x.\, t \\
& & | & t_1\ t_2 \\
& & | & \&x \\
& & | & \&\text{mut}\ x \\
& & | & *x \\
& & | & \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \\
& & | & \text{let } x = t_1 \text{ in } t_2 \\
& & | & x := t \\
& & | & *x := t \\
& & | & \text{true} \\
& & | & \text{false} \\
& & | & \text{unit} \\
& & | & 0 \\
& & | & \text{pred } t \\
& & | & \text{succ } t \\
& & | & \text{iszero } t \\
& & | & \text{natvec}(t_1, t_2, \ldots t_n) \\
& & | & \text{get}(t_1, t_2) \\
& & | & \text{getmut}(t_1, t_2) \\
& & | & \text{push}(t_1, t_2) \\
& & | & \text{pop}(t_1) \\
& & | & t : T
\end{array}
$$

For the syntax of types, we first introduce a couple of constructs that will make them up. This syntax is based on a paper by Jung et al. (2018) [5].

$$
\begin{array}{rrcl}
\textsf{Lifetime} & \kappa & ::= & \alpha \\
\textsf{Mod} & \mu & ::= & \text{mut} \\
& & | & \text{shr}
\end{array}
$$

Lifetimes are represented by numeric values, corresponding to their scope depth. We can make this definition, as each binding site introduces a new scope. The subtyping relation works by then considering a reference with a larger lifetime (smaller scope depth) a subtype of a reference with a smaller lifetime (larger scope depth). The mod corresponds to the type of reference (mutable or shared, i.e. immutable). The types of references will be annotated with both of these constructs, in addition to a type and the variable that they borrow. Functions can be polymorphic over lifetimes, as denoted by the $\forall\alpha.$ quantifier. This is required for any functions that have a reference type as a parameter, since the concrete lifetime is not important (or even knowable).

We distinguish types that aren't themselves references as a subset of all types. This helps with reasoning about typing rules in certain cases where checking rules differ slightly based on this.

$$
\begin{array}{rrcl}
\textsf{Non-Reference Types} & T_* & ::= & \text{Nat} \\
& & | & \text{Bool} \\
& & | & \text{Unit} \\
& & | & \forall\alpha.T_1 \to T_2 \\
& & | & \text{NatVec} \\
\textsf{Types} & T & ::= & {}_x\&^{\kappa}_{\mu} T \\
& & | & T_*
\end{array}
$$

Lastly, when relevant, our typing rules make use of the following named set, denoting types that are copyable.

$$\text{Copy} = \{\text{Nat}, \text{Bool}, \text{Unit}, {}_x\&^{\kappa}_{\text{shr}} T\}$$

3.2 Typechecking Pass

We use bidirectional typechecking, as we felt that this was the simplest and most natural solution, especially for terms like conditionals whose type depends on that of the sub terms. As expected, the typechecker checks that terms are well-typed, and annotates the corresponding abstract syntax tree node with the determined type of the node, generating a new abstract syntax tree with nodes annotated with their types.

This pass also does all the necessary work for lifetimes; assigning lifetimes to each of the variables and their respective borrows (see Figure 3), checking the subtype relations for references (T-LftSubtype and T-MutShrSubtype, see Figure 6), and handling instantiation of functions polymorphic over lifetimes (Figure 4).

```
let a = 0 in          (* a has lifetime '0 *)
let b = succ 0 in     (* b has lifetime '1 *)
let d = &b in         (* d has type &'1 nat *)
let e = &mut a in     (* e has type &'0 mut nat *)
...
```

Figure 3: Example code illustrating lifetimes of variables

```
let y = (\x. x) : &'a nat -> &'a nat in
let k = succ (succ 0) in (* k has lifetime '1 *)
let z = y &k in          (* z has type &'1 nat*)
...
```

Figure 4: Example code illustrating lifetime polymorphic instantiation

3.3 Borrow Checking Pass

The borrow checking pass is responsible for checking rules related to ownership and borrowing. Most typing rules of interest are shown in Figure 6. Note that contexts are omitted from a rule if they are not used and don't change in the derivation. Rule T-Var checks the affine type property - that no value is used more than once. This is done by maintaining a context of variable usages which we call Δ. If the value is not trivially copyable, the variable gets added to this set (T-Var). For copyable types, we don't need to track its usage for the purpose of validating move semantics (T-VarCopy).

Additionally, we check that variables cannot be moved while being borrowed, by maintaining a context of the current borrows, which we call B. B is simply a list of variable/modifier pairs, which lets us determine if a variable is borrowed immutably or mutably.

A couple of interesting special cases include the handling of conditionals, and providing the ability to assign a value to a variable that has had a value moved out of it. In the former case (T-If), we will output the union of both Δ contexts generated by traversing each branch of the conditional. In the latter case (T-Assign), we remove the current variable from the Δ context when assigning it a value if it is present, so that it can be used again.

The borrow checking pass also uses the B context to check that borrows adhere to the mutability rule, that is, mutable borrows can't exist while any other borrow to the same value is in scope (T-BorrowMut), and immutable borrows can't exist while a mutable borrow to the same value is in scope (T-BorrowShr).

$$\Gamma = \cdot \mid \Gamma, x : T; \alpha \quad \Delta = \cdot \mid \Delta, x \quad F = \cdot \mid F, x \quad B = \cdot \mid B, x_\mu$$

Figure 5: Definitions of typing contexts.

$$\frac{\Gamma(x) = T \quad x \notin \Delta \quad T \notin \text{Copy} \quad x_\mu \notin B}{\Gamma \mid B \mid \Delta \vdash x : T \mid \Delta, x} \text{ (T-Var)} \qquad \frac{\Gamma(x) = T \quad T \in \text{Copy}}{\Gamma \mid \Delta \vdash x : T \mid \Delta} \text{ (T-VarCopy)}$$

$$\frac{\Gamma \vdash t : T_1 \quad T_1 <: T_2}{\Gamma \vdash t : T_2} \text{ (T-Subtype)}$$

$$\frac{\alpha \leq \beta}{\Gamma \vdash {}_x\&^\alpha_\mu\ T <: {}_y\&^\beta_\mu\ T} \text{ (T-LftSubtype)} \qquad \frac{\alpha \leq \beta}{\Gamma \vdash {}_x\&^\alpha_{\text{mut}}\ T <: {}_y\&^\beta_{\text{shr}}\ T} \text{ (T-MutShrSubtype)}$$

$$\frac{\Gamma(x) = T; \alpha \quad x \notin \Delta \quad x \in F \quad x_{\text{mut}} \notin B}{\Gamma \mid F \mid B \mid \Delta \vdash \&x : {}_x\&^\alpha_{\text{shr}}\ T \mid \Delta \mid B, x_{\text{shr}}} \text{ (T-BorrowShr)}$$

$$\frac{\Gamma(x) = T; \alpha \quad x \notin \Delta \quad x \in F \quad x_\mu \notin B}{\Gamma \mid F \mid B \mid \Delta \vdash \&\text{mut } x : {}_x\&^\alpha_{\text{mut}}\ T \mid \Delta \mid B, x_{\text{mut}}} \text{ (T-BorrowMut)}$$

$$\frac{\Gamma \mid F \mid \Delta \vdash x : {}_y\&^\alpha_\mu\ T \mid \Delta' \quad T \in \text{Copy}}{\Gamma \mid F \mid \Delta \vdash\ ^*x : T \mid \Delta'} \text{ (T-Deref)}$$

$$\frac{\Gamma \mid \Delta \mid B \vdash t_1 : \text{Bool} \mid \Delta_1 \mid B_1 \quad \Gamma \mid \Delta_1 \mid B_1 \vdash t_2 : T \mid \Delta_2 \quad \Gamma \mid \Delta_1 \mid B_1 \vdash t_3 : T \mid \Delta_3}{\Gamma \mid \Delta \mid B \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \mid \Delta_2 \cup \Delta_3 \mid B} \text{ (T-If)}$$

$$\frac{\Gamma \mid \Delta \mid B \vdash t : T \mid \Delta' \quad \Gamma(x) = T \quad x_\mu \notin B}{\Gamma \mid \Delta \mid B \vdash x := t : \text{Unit} \mid \Delta' \setminus x} \text{ (T-Assign)}$$

$$\frac{\Gamma \mid F \mid \Delta \mid B \vdash t_1 : T_1 \mid \Delta_1 \mid B_1 \quad \Gamma, x : T_1; |\Gamma| \mid \mid F, x \mid \Delta_1 \mid B_1 \vdash t_2 : {}_x\&^\alpha_\mu T_2 \mid \Delta_2 \mid B_2 \quad \alpha < |\Gamma|}{\Gamma \mid F \mid \Delta \mid B \vdash \text{let } x = t_1 \text{ in } t_2 : {}_x\&^\alpha_\mu\ T_2 \mid \Delta_2 \mid B, x_\mu} \text{ (T-LetRef)}$$

$$\frac{\Gamma \mid F \mid \Delta \mid B \vdash t_1 : T_1 \mid \Delta_1 \mid B_1 \quad \Gamma, x : T_1; |\Gamma| \mid \mid F, x \mid \Delta_1 \mid B_1 \vdash t_2 : T_* \mid \Delta_2 \mid B_2}{\Gamma \mid F \mid \Delta \vdash \text{let } x = t_1 \text{ in } t_2 : T_* \mid \Delta_2 \mid B} \text{ (T-Let)}$$

$$\frac{\Gamma \mid \Delta \vdash t_1 : \text{Nat} \mid \Delta_1 \quad \Gamma \mid \Delta_1 \vdash t_2 : \text{Nat} \mid \Delta_2 \quad \ldots \quad \Gamma \mid \Delta_{k-1} \vdash t_k : \text{Nat} \mid \Delta_k}{\Gamma \mid \Delta \vdash \text{natvec}(t_1, t_2, \ldots t_k) : \text{NatVec} \mid \Delta_k} \text{ (T-NatVec)}$$

$$\frac{\Gamma \mid \Delta \mid B \vdash t_1 : {}_x\&^\alpha_\mu\ \text{NatVec} \mid \Delta_1 \mid B_1 \quad \Gamma \mid \Delta_1 \mid B_1 \vdash t_2 : \text{Nat} \mid \Delta_2 \mid B_2}{\Gamma \mid \Delta \mid B_2 \vdash \text{get}(t_1, t_2) : {}_x\&^\alpha_{\text{shr}}\ \text{Nat} \mid \Delta_2 \mid B, x_{\text{shr}}} \text{ (T-NatVecGet)}$$

$$\frac{\Gamma \mid \Delta \mid B \vdash t_1 : {}_x\&^\alpha_{\text{mut}}\ \text{NatVec} \mid \Delta_1 \mid B_1 \quad \Gamma \mid \Delta_1 \mid B_1 \vdash t_2 : \text{Nat} \mid \Delta_2 \mid B_2}{\Gamma \mid \Delta \vdash \text{getmut}(t_1, t_2) : {}_x\&^\alpha_{\text{mut}}\ \text{Nat} \mid \Delta_2 \mid B, x_{\text{mut}}} \text{ (T-NatVecGetMut)}$$

Figure 6: Selected typing rules for our language.

Finally, this pass also checks that references themselves are valid with respect to their lifetimes. This could potentially arise if an escaping reference is returned out of a let in or lambda statement. We make use of the lifetime annotations from the previous pass, and ensure that these references being returned out have larger lifetime than the scope they are being returned to. The rule T-LetRef, though quite complex, encodes this.

We introduce the fourth context here - F, which tracks variables bound in the current scope of the current lambda term. We do not allow captured borrows in our language, since allowing captured borrows would further complicate the type system as closures would need lifetimes attached to them.

Additionally, we use the semicolon as a shorthand to denote both a variable's type (on the left) and lifetime (on the right) in the Γ (variables) context together. The rule states that when a reference is returned out with lifetime α, the program typechecks if the index α is strictly smaller than the current scope. This means that the reference that is being returned out of the scope outlives the scope itself, which is what we want. Additionally, when checking the inner term t_2, we add the bound variable x with type T_1 and assign it a lifetime index corresponding to the current scope depth, which is equivalent to the size of the set Γ.

4 Challenges

Ending borrows. The most difficult aspect of this type system is knowing when to end borrows. Consider the following code example:

```
let x = 0 in
let y = 0 in
let z = (let a = &y in let b = &x in b) in
let d = &mut y in
...
```

In the expression that is bound to `z`, we borrow both `x` and `y` immutably. However, the borrow of `x` ends as `b` is only bound in the inner scope, but the borrow of `x` continues since `b` is returned as the result of the expression. As such, `y` is able to be mutably borrowed again in the outer scope. However, we would not be able to do the same with `x` as it is immutably borrowed and bound to `z`.

To know which borrows to keep alive, we decided to tag each reference with the variable that it borrows. Then, in the rule T-LetRef we simply include the returned reference's variable in the borrow context B. This is also useful for "partial borrows", like with NatVec, as we can mark the borrows of individual elements as borrowing the entire container to prevent aliasing. This can be seen in the rules T-NatVecGet and T-NatVecGetMut.

```
let x = natvec_make(0, succ 0, succ (succ 0)) in
let elem = natvec_get_mut(&mut x, succ 0) in
natvec_push(&mut x, 0) (* error: cannot mutably borrow 'x' *)
```

Shadowing. Since many of the contexts store the names of variables, variables with the same name cause issues. However, forcing the programmer to come up with unique names for every variable is extremely inconvenient. We resolve this issue by adding an additional pass after parsing that assigns every variable a unique ID which is used internally in the contexts. The original name is still used for error messages.

5 Conclusion and Future Work

The language we implemented captures the basic ideas of ownership and borrow checking like in Rust, however it is quite limited in scope. It is useful from a theoretical standpoint to formally understand the semantics of borrowing, but it is certainly far from practical programming. There are many areas in which we could make big improvements in terms of usability:

Recursion. The language as it stands currently is not even Turing complete as it lacks a looping construct. A simple way to implement this would be to add a Fix term to bind a function in itself. Alternatively, we could add recursive types so that the fixed-point combinator can be typed.

Captured borrows. We disallowed capturing by borrow in this language to simplify the implementation, however allowing captured borrows could be implemented by attaching a lifetime parameter to functions (taking the minimum lifetime of all captured borrows) and treating them similarly to references in the borrow checking rules.

Reusable functions. Lambda abstractions in our language are not Copy so they can only be called once. This is because lambdas can capture variables by move, and those values are subject to the affine typing rules so they should only be able to be used once. However, this is overly restrictive since functions that do not capture anything should be able to be used multiple times. To implement functions that can be called multiple times, we could introduce a notion of "top-level functions" which cannot capture anything by definition and could be added as an exception to the affine typing rules.

A more general approach would be to check whether or not a lambda abstraction captures any variables and in turn assign it a different type depending on whether or not it can be called multiple times. This is essentially what Rust does with its `FnOnce`, `Fn`, and `FnMut` traits.

Product types. We introduced the NatVec type to have some form of "container" type which is not trivially copyable to illustrate more interesting cases of ownership and borrowing semantics. However, its use is limited as it is not general purpose. The type system becomes much richer and more powerful with general purpose product types, but this adds significant complexity; types that hold references must be polymorphic over lifetimes, handling disjoint borrows of individual fields of a product type and partial moves of individual fields, to name a few challenges.

6 Related Work

Substructural and uniqueness type systems for safer memory management have been a popular research topic in the literature. As stated before, the idea of region-based memory management has its origins with the Cyclone language.

Regarding Rust specifically, the RustBelt project (2016-2021) aimed to formalize the foundations of the language. Jung et al. (2018) [5] model the Rust language with a lambda calculus they call λ_{Rust}, which is quite close to the Middle Intermediate Representation (MIR) used by the actual Rust compiler. Their model includes a rich type system that matches many of Rust's features (general sum types, product types, recursive types, recursion, concurrency). Their model of lifetimes differs from ours as they use "lifetime tokens", which is less restrictive as it allows for reborrowing and splitting borrows on product types. They encode the semantics of λ_{Rust} in the Coq proof assistant and prove soundness of a few types in Rust's standard library, such as `Arc` and `Mutex`.

Dang et al. (2019) [2] extend this work with a memory model that includes atomics and memory ordering which provides a more accurate model of concurrency.

References

[1] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, 2002.

[2] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rustbelt meets relaxed memory. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[3] Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. *ACM SIGPLAN Notices*, 47(10):21–40, 2012.

[4] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293, 2002.

[5] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[6] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, San Francisco, CA, 2018.