School of Electronic Engineering
and Computer Science

Final Report

Programme of study:
BSc Computer Science

Project Title:
Featherweight Muse: A Formalization and Interpreter Implementation

Supervisor:
Dr. Raymond Hu

Student Name:
Jorge Martín Albeníz

Final Year
Undergraduate Project 2023/24

Date: May 1, 2024

Abstract

Modern programming languages aim to provide a balance between safety, efficiency, and usability. Memory management is a crucial aspect of language design, as it directly impacts all three of these factors. As a first step towards developing a programming language that features a fast and safe memory management model, we present the formalization and implementation of Featherweight `muse` (FM), a statically-typed, interpreted programming language. The language is designed to provide memory safety and efficient resource management through its ownership system and borrow checking. This report outlines the syntax and reduction rules for small-step operational semantics of FM, as well a type system for managing memory at compile time. The report also covers the implementation, testing, and evaluation of an interpreter for FM and discusses future research directions and enhancements for `muse`.

Contents

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Manual memory management requires developers to explicitly allocate and deallocate memory which can be error-prone and lead to security vulnerabilities and application crashes (Nagarakatte *et al.* 2012; Khan *et al.* 2012). For example, a report on Google's Chromium project found that 70% of severe security vulnerabilities were due to memory safety issues (Google, 2021). Garbage collection, while automated, can introduce performance penalties and unpredictability in memory usage (Blackburn *et al.* 2004). The impact on real-time systems and applications, such as audio programs and video games that require low latency can be significant, as garbage collection can introduce unpredictable pauses and overhead (Pizlo and Vitek, 2006). Borrow checking and ownership systems, as seen in Rust, provide a solution to these problems by enforcing memory safety at compile time without the runtime costs associated with garbage collection (Zhang *et al.* 2023). By formalizing a featherweight language with these features, we can make the first step towards designing a safer, more efficient, and more user-friendly programming language that can be used in a wide range of applications.

1.2 Aims and Objectives

The aim of this project is to take the first step in the development of `muse` by formalizing and implementing Featherweight version, which is contains the features necessary to cover the core mathematics related to the type system. This featherweight version will serve as a medium for designing and structuring borrow checking and ownership rules which will be inspired and based on the systems presented by Pearce (2021), Payet *et al.* (2022), and Griesemer *et al.* (2020). To evaluate the effectiveness of these systems, a small-step interpreter will be implemented to execute programs written in Featherweight `muse`, and the properties of the language will be evaluated. Ideally this will provide a foundation for future research and development of the full `muse` language.

The project objectives are as follows:

1. **Formalization of FM:**
    - The syntax of the language will be formalized using a context-free grammar. This grammar will be the basis of formalization for the rest of the language.
    - The semantics will be formalized using a small-step operational semantics, presented as reduction rules.
    - The borrow checking and ownership rules will be formalized using typing rules.
    - The properties of the language will be defined, including progress, preservation, and type and borrow safety
2. **Interpreter Implementation:**
    - Create a lexer and parser for FM to transform source code into an abstract syntax tree (AST).
    - Implement the typing rules required for borrow checking and ownership.
    - Implement a small-step interpreter for FM that follows the reduction rules defined in the formalization.
3. **Testing and Evaluation:**
    - Define the core properties of FM and evaluate them using the interpreter.

- Perform rigorous testing of language semantics and their interactions with at least 80& code coverage.
- Evaluation of the compiler's effectiveness and adherence to the formalized language structure.

4. **Future Outlook for `muse`:**
    - Identifying potential areas for future research and development to enhance the usability and features of `muse`.

Chapter 2

Background and Literature Review

To supplement the work to be done in this report this literature review will compare and contrast between different memory management techniques used within modern programming languages. Particularly, we will look at type systems that leverage borrow checking and ownership types to statically analyse the memory usage of a program, asserting its correctness and safety. Finally, we that we will look at featherweight languages and how the core mathematics are formalized to model the type system and operational semantics of a language.

2.1 Memory Management Techniques in Programming Languages

Manual memory management, as seen in languages like C, requires programmers to explicitly allocate and free memory using functions such as *malloc* and *free*. The former allocates memory on the heap, while the latter deallocates it. This method offers maximum control over memory usage, which can lead to performance optimizations in systems where resources are limited. However, it also imposes a significant cognitive burden on developers and is prone to human error leading to issues such as memory leaks, double frees, and dangling pointers. Rezaei and Kavi (2000) explore these challenges, noting that while manual management can optimize performance, it often does so at the cost of program safety and developer overhead.

In contrast, garbage collection (GC), used by languages like Java and Python, automates memory management by periodically reclaiming memory that is no longer in use at runtime. This process enhances safety by reducing the risk of memory leaks and other related bugs. However, as noted by Jones *et al.* (2011), the automation of garbage collection can introduce performance unpredictability. GC typically involves pausing program execution, which can be a deal-breaker for real-time systems or applications that require low latency. The trade-offs between safety and performance are a central theme in the literature on garbage collection, with researchers exploring various algorithms and strategies to optimize memory management in different contexts.

Reference counting, employed by languages such as C++ and Objective-C, represents a middle ground between manual memory management and garbage collection. It involves keeping a count of references to each object, so memory can be freed when there are no references remaining. This technique can mitigate some performance issues associated with traditional garbage collection by deallocating objects incrementally. However, it can struggle with cyclic dependencies unless supplemented by additional algorithms to detect and break these cycles, as discussed by Wason and Kumar (2016). Furthermore, reference counting can introduce overhead due to the need to update reference counts frequently, impacting runtime performance.

2.2 Foundations of Type Systems and their use in memory management

As discussed by Pierce (2002), type systems are a fundamental aspect of programming language theory, that provide a structured approach to statically analyse programs and assert their correctness. Type systems achieve this by statically classifying and managing data into types and defining a set of logical rules upon those types, which further restrict the set of valid programs that a programmer can write. This approach helps catch and prevent runtime errors at compile time, thereby improving the safety and reliability of programs.

Some research within programming language theory focuses on using type systems to manage memory by restricting memory aliasing. Aliasing, in the context of memory management, refers to the ability to access the same memory location through different references. This can lead to unintended side effects and memory corruption, making it a critical issue in system programming. One such approach presented by Tofte and Talpin (1997) refers to *region-based* memory management, which places all values into stacked *regions*, using region calculus to track and infer the allocation and deallocation of memory.

Subsequently, extensions on their work has been successfully implemented in languages like Cyclone and Rust. Cyclone is a dialect of C designed to provide memory safety through static analysis by managing memory on a per-region basis (Jim *et al.* 2002). Originally, Cyclone structured their regions using a stack, in the same way that was presented by Tofte and Talpin (1997), however, this LIFO scheme proved to be restrictive as data cannot be deallocated early in certain scenarios. For example, when writing a program with infinite loops, such as servers and video games, data that is declared outside the loop will never be deallocated, even if the data is no longer being used. The core Cyclone team further extended the specification with linear references (Fluet *et al.* 2006; Swamy *et al.* 2006).

In contrast, Rust, introduces *non-lexical lifetimes*. Lifetimes are named regions, within the original model from Tofte and Talpin (1997), data allocated within *lexically-scoped* lifetimes are deallocated at the end of a regions scope. On the other hand, non-lexical lifetimes extend the lifetime to the end of the block which allows for more flexible memory management (RustLang, 2017; Weiss *et al.* 2019).

2.3 A deep dive into Rust's Ownership and Borrow Checking System

Rust, a systems programming language, introduces a novel approach to memory management that builds on top of decades of programming language research. Its notable use of region based memory management, ownership rules, and its borrow checking system, allows users to write memory safe code without garbage collection. The ownership system in Rust ensures that each value has a unique owner, and memory is automatically freed when the owner goes out of scope (Weiss *et al.* 2019). Rust's borrow checker ensures that references do not outlive the data they point to, preventing dangling pointers and ensuring that data races cannot occur. Pearce (2021) provides a detailed examination of how Rust's borrow checker operates to enforce memory safety

Borrow checking offers substantial practical benefits. For instance, the system allows for safe memory access patterns that are conducive to concurrent programming, significantly reducing the likelihood of race conditions without the overhead of runtime checks typically associated with other safe concurrency models. However, the complexity of Rust's borrow checker can also pose challenges, particularly for new users of the language. The strictness of the borrowing rules, while beneficial for preventing bugs, can also steepen the learning curve and may lead to frustration due to frequent compiler rejections on seemingly valid code. This aspect is critically analysed in works like "Understanding memory and thread safety practices and issues in real-world Rust programs" by Qin *et al.* (2020), which discusses how these challenges can impact the adoption and effective use of Rust in software development projects.

Further empirical analysis by Xu *et al.* (2020) explores the real-world effectiveness of Rust's memory safety features. They found that while Rust significantly reduces memory safety vulnerabilities, however issues still arise primarily due to misuse of unsafe code blocks, which bypass the borrow checker's protections (Klabnik and Nichols, 2023). Unsafe code blocks are typically used when Rust's borrow checker is too restrictive, highlights a critical area where Rust's safety mechanisms are not expressive enough, suggesting a potential avenue for further enhancement of the language's model to support more programs.

In conclusion, Rust's borrow checking system is a foundational element of its approach to memory safety, significantly influencing how safety and concurrency are managed in systems programming. The scholarly work reviewed here not only underscores the effectiveness of this system in enhancing safety but also points to the ongoing need for improvements to accommodate more flexible programming needs. As Rust continues to evolve, the balance between safety, usability, and flexibility will likely remain a

central focus of both theoretical exploration and practical implementation in the language's development trajectory.

2.4 Role of Featherweight Languages in Programming Language Research

Featherweight languages are used to study the theoretical foundations of programming languages. These languages strip down to the core grammatical structures, facilitating the analysis of specific language features without the complexities typically associated with full-fledged programming languages. A prime example of such a language is Featherweight Java, defined by Igarashi *et al.* (2001), which simplifies Java to its essence to investigate type safety and other fundamental properties.

Formalizations of Featherweight languages generally begin with a definition of syntax using context-free grammar. This grammar simplifies the language to its core components, making theoretical analysis more manageable. For example the paper, Featherweight Go, aims to expand the Go language with generics. It achieves this by defining a syntax that includes the core features of Go and then defines a new model that is extended to include generics. (Igarashi *et al.* 2001).

Typing rules in these languages are defined using concise proof trees that ensure type safety and consistency across operations. These rules are crucial for maintaining logical coherence in program execution. As seen in Griesemer *et al.* (2020) operational semantics are defined using reduction rules, which describe in detail how program statements are executed within the language's framework and providing insights into the runtime behaviour and interaction of language constructs.

Featherweight languages are also used to prove fundamental properties. Progress and preservation as seen in Pearce (2021), are essential properties that verify the soundness of programming languages. Progress ensures that computations do not get stuck unless a value is reached, and preservation confirms that once a program is well-typed, it remains so as it executes. These properties help demonstrate the robustness and reliability of the language's formal system (Harper, 2016).

The structured approach to formalization in Featherweight languages makes them particularly useful for exploring modifications and extensions to existing languages. For instance, Featherweight Java has been extended to include generics, proving type safety and demonstrating the adaptability of Featherweight models for advancing language design (Igarashi *et al.* 2001).

In conclusion, Featherweight languages like Featherweight Java provide powerful tools for theoretical research in computer science. By formalizing the syntax, typing rules, and operational semantics, they offer a clear and structured means of studying complex programming languages. These languages not only aid in understanding fundamental programming concepts but also serve as tests for developing new language features in a controlled and rigorous environment.

2.5 Conclusion

The analysis provided in this literature review showcases the different memory management techniques that are in play in modern programming languages, particularly in the domains of type systems and memory management. The exploration of type systems has revealed their important role in enhancing program reliability and resource management, with advanced systems that implement region-based memory aliasing. In particular, ownership rules and borrow checking in Rust are effective at managing memory safety at compile time. The literature not only confirms the effectiveness of these features in mitigating common programming errors but also critiques their complexity and the challenges they pose, particularly for new users. Despite these hurdles, ongoing research aims to refine these mechanisms to support more flexible and powerful programming paradigms.

Chapter 3

Syntax

The syntax of FM is defined as a context-free grammar as seen in Figure3.1. It is designed to provide all the syntactical elements that are necessary for demonstrating the core mathematical features of the language. Importantly we note that not all the syntax elements are directly accessible to the programmer, as some are used internally to represent the state of the program during evaluation. For example references are used to represent used internally to represent memory locations within partially reduced terms and cannot be written by the programmer. During this report we will refer to these elements as *internal syntax*.

Identifier	x	$:= a - zA - Z$
Numeric Literal	n	$:= 0 \mid 1 \mid 2 \mid \ldots$
Reference	ℓ	$:= \ell^\bullet \; \ell^\circ$
Value	v	$:= n \mid r$
Partial Value	$v_\perp$	$:= v \mid \perp$
LVal	w	$:= x \mid *w$
Term	t	$:= w \mid v \mid \textbf{move}\ w \mid \textbf{copy}\ w \mid \textbf{box}\ t \mid [\textbf{mut}]\ \textbf{ref}\ w \mid \textbf{let mut}\ w = t \mid w = t \mid \epsilon$
Program	P	$:= \bar{t}$
Type	T	:= int | [mut] ref w | box T
PartialType	$T_\perp$	$:= T \mid \lfloor T \rfloor$

Figure 3.1: Syntax of Featherweight `muse`

The only numbers supported in `muse` are positive integers. There are two types of references, *owned* and *borrowed* references, denoted by $\ell^\bullet$ and ℓ° respectively. Owned references recursively drop the location that they point to, while borrowed references, do not. Values can be either numeric literals, references, or the empty value ϵ. Partial values are used to represent moved values, they can be either a value or $\perp$. LVals are used to represent terms that can be assigned to, they can be either a variable x or a dereferenced variable $*w$. Programs are defined using the notation $\bar{t}$ to mean a possibly empty sequence of t_i. We implicitly drop ϵ from sequences. Types in FM are either integers, mutable references, references, or boxed types (for values that live on the heap). Note that the undefined type $\lfloor T \rfloor$ is also part of the *internal syntax*, as is used to represent a value that has been moved, while still retaining the original shape of that value in the case of reassignment. An example of this can be seen in Figure3.2.

It is important to note that in this specification of FM, terms like **move** w and **copy** w are used to explicitly denote their respective operations. However, in practice, these operations would be implicit, as the type system would determine whether a value should be moved or copied, as seen in Section 4.1.3. This data is then propagated to the later stages in the evaluation. However, for the sake of clarity and to maintain the focus on the main core and mathematics of the borrow-checking features, we decided to include these operations explicitly in the syntax.

```
let mut x = 0 // T(x) = int
let y = x // after move T(x) = undefined(int)
// any attempt to read x here would result in an error
x = 1 // T(x) = int
```

Figure 3.2: Example usage of undefined types

Chapter 4

Type System

The type system of `muse` implements borrow-checking and ownership types. The system expressed as a set of typing rules and showcase how a program is type checked to assert its correctness. Before delving into the formalization of FM, it is important to note that for the sake of brevity and to focus on the core mathematical features of borrow checking and ownership rules, we have omitted lifetimes from sections 5, 4 and 6. The full formalization of the language, including lifetimes, can be found in the appendix.

Our type system employs *flow typing*, where typing rules determine both the type for a given term and its effects. The type system is defined as a relation between a term t and a type T under the type environment Γ which through evaluation of t may produce Γ'.

$$\Gamma \vdash t : T \dashv \Gamma'$$

The *type environment* Γ is a map from variables w to partial types $T_\perp$.

4.1 Preliminaries

The following helper functions are used to define the type system for lightweight `muse`.

4.1.1 Root

Root is a function that returns the root of an LVal. Given a type environment Γ and an LVal w, the root of w is defined as:

$$\begin{aligned} root(\Gamma, x) &= x \\ root(\Gamma, *w) &= root(\Gamma, w) \end{aligned}$$

4.1.2 Contains

Let Γ be a type environment where $\Gamma(x) = T_\perp$, then $\Gamma \vdash x \rightsquigarrow T$ is defined as contains(Γ, $T_\perp$, T) where:

$$contains(\Gamma, T_\perp, T) = \begin{cases} \text{contains}(\Gamma,\, T'_\perp,\, T) & \text{if } T_\perp = \text{box } T' \\ \text{true} & \text{if } T_\perp = T \\ \text{false} & \text{otherwise} \end{cases} \tag{4.1}$$

4.1.3 IsCopy

IsCopy is a function that determines if a type T is considered to be *copyable*. A type is considered to be copyable if it is an integer or an immutable borrow.

$$\text{isCopy}(T) = \begin{cases} \text{true} & \text{if } T = \text{int or } T = \text{ref } T \\ \text{false} & \text{otherwise} \end{cases} \tag{4.2}$$

$$\frac{\Gamma(x) = T_\perp}{\Gamma \vdash x : T_\perp \dashv \Gamma} \text{ (T-Var)} \qquad \frac{\Gamma \vdash w : T_\perp \dashv \Gamma}{\Gamma \vdash *w : T_\perp \dashv \Gamma} \text{ (T-DerefBox)} \qquad \frac{\Gamma \vdash w : T \dashv \Gamma}{\Gamma \vdash *w : T_\perp \dashv \Gamma} \text{ (T-DerefRef)}$$

$$\Gamma \vdash n : int \dashv \Gamma \text{ (T-Val)} \qquad \frac{\Gamma \vdash t : T \dashv \Gamma'}{\Gamma \vdash \texttt{box}\ t : \text{box}\ T \dashv \Gamma'} \text{ (T-Box)}$$

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \neg\text{readProhibited}(\Gamma, w)}{\Gamma \vdash \textbf{ref}\ w : \text{ref}\ w \dashv \Gamma} \text{ (T-Bor)}$$

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \text{mut}(\Gamma, w) \qquad \neg\text{writeProhibited}(\Gamma, w)}{\Gamma \vdash \textbf{mut ref}\ w : \text{mut ref}\ w \dashv \Gamma} \text{ (T-MutBor)}$$

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \text{isCopy}(T) \qquad \neg\text{readProhibited}(\Gamma, w)}{\Gamma \vdash \texttt{copy w} : T \dashv \Gamma} \text{ (T-Copy)}$$

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \neg\text{writeProhibited}(\Gamma, w) \qquad \Gamma' = \text{move\_var}(\Gamma, w)}{\Gamma \vdash \texttt{move w} : T \dashv \Gamma'} \text{ (T-Move)}$$

$$\frac{x \notin \text{dom}(\Gamma) \qquad \Gamma \vdash t : T \dashv \Gamma' \qquad \Gamma'' = \Gamma'[x \mapsto T]}{\Gamma \vdash \texttt{let mut}\ x = t : \epsilon \dashv \Gamma''} \text{ (T-Declare)}$$

$$\frac{\begin{array}{c}\Gamma \vdash w : T_\perp \dashv \Gamma \qquad \Gamma \vdash t : T' \dashv \Gamma' \qquad \textit{shape\_compatible}(\Gamma', T_\perp, T') \\ \Gamma'' = \text{write}(\Gamma', w, T') \qquad \neg\text{writeProhibited}(\Gamma'', w)\end{array}}{\Gamma \vdash w = t : \epsilon \dashv \Gamma''} \text{ (T-Assign)}$$

$$\frac{T_1 \vdash t_1 : T_1 \dashv \Gamma_1 \qquad \cdots \qquad T_n \vdash t_n : T_n \dashv \Gamma_n + 1}{\Gamma \vdash \overline{t} : T \dashv \Gamma_{n+1}} \text{ (T-Seq)}$$

Figure 4.1: FM typing rules

4.1.4 ReadProhibited

Given type environment Γ and a set of all variables LV_Γ. Variable w, is considered to be *read prohibited* given that some variable $x \in LV_\Gamma$ exists where x *contains* a mutable borrow.

$$\text{readProhibited}(\Gamma, w) = \begin{cases} \text{true} & \text{if } \exists_{x \in LV_\Gamma} \cdot (x \rightsquigarrow \text{mut ref } w) \\ \text{false} & \text{otherwise} \end{cases} \tag{4.3}$$

4.1.5 writeProhibited

Given type environment Γ and a set of all variables LV_Γ, variable w is considered to be *write prohibited* given that some variable $x \in LV_\Gamma$ exists where x *contains* an immutable borrow or $readProhibited(\Gamma, w)$ holds.

$$\text{writeProhibited}(\Gamma, w) = \begin{cases} \text{true} & \text{if } \exists_{x \in LV_\Gamma} \cdot (x \rightsquigarrow \text{ref } w) \wedge \text{readProhibited}(\Gamma, w) \\ \text{false} & \text{otherwise} \end{cases} \tag{4.4}$$

4.1.6 Move_var

Given type environment Γ and a variable w, let $x = \text{root}(\Gamma, w)$, then move_var($\Gamma$, w) is a partial function defined as $\Gamma[x \mapsto T'']$ where $T'' = \text{undefine}(w, T)$.

$$\begin{aligned}\text{undefine}(x, T) &= \lfloor T \rfloor \\ \text{undefine}(*w, \text{box } T) &= \text{box } T' \textbf{ where } T' = \text{undefine}(w, T)\end{aligned}$$

4.1.7 Mut

Mut is a function that determines whether the variable contains an immutable borrow. Given type environment Γ and an LVal w, let T be the type of w under Γ. Then mut(Γ, w) is a partial function defined as mutable(Γ, w, T) where:

$$\begin{aligned}\text{mutable}(\Gamma, *w, \text{mut ref } u) &= \text{mut}(\Gamma, u) \\ \text{mutable}(\Gamma, *w, \text{box } T) &= \text{mut}(\Gamma, w) \\ \text{mutable}(\Gamma, w, T) &= \text{true} \wedge T \neq \text{ref } u\end{aligned}$$

4.1.8 Shape_compatible

Given type environment Γ, and types $T_\perp$, $T'_\perp$. Then shape_compatible(Γ, $T_\perp$, $T'_\perp$) is a partial function defined as:

$$\begin{aligned}\text{shape\_compatible}(\Gamma, \text{int}, \text{int}) &= \text{true} \\ \text{shape\_compatible}(\Gamma, \text{ref } w, \text{ref } u) &= \text{true} \\ \text{shape\_compatible}(\Gamma, \text{mut ref } w, \text{mut ref } u) &= \text{true} \\ \text{shape\_compatible}(\Gamma, \text{box } T_\perp, \text{box } T'_\perp) &= \text{shape\_compatible}(\Gamma, T, T') \\ \text{shape\_compatible}(\Gamma, \lfloor T \rfloor, T') &= \text{shape\_compatible}(\Gamma, T, T') \\ \text{shape\_compatible}(\Gamma, T, \lfloor T' \rfloor) &= \text{shape\_compatible}(\Gamma, T, T')\end{aligned}$$

4.1.9 Write

Let Γ be a type environment, w be an LVal, and $T_\perp$ be a type. Then write(Γ, w, $T_\perp$) is a partial function defined as $\Gamma[x \mapsto T_\perp]$ where $(\Gamma', T'_\perp) = \text{update}(\Gamma, w, T_\perp, T)$.

$$\begin{aligned}\text{update}(\Gamma, x, T_\perp, T) &= (\Gamma, T) \\ \text{update}(\Gamma, *w, \text{box } T_\perp, T) &= (\Gamma', \text{box } T') \textbf{ where } (\Gamma', T') = \text{update}(\Gamma, w, T_\perp, T) \\ \text{update}(\Gamma, *w, \text{mut ref } u, T) &= (\Gamma', \text{mut ref } u) \textbf{ where } \Gamma' = \text{write}(\Gamma, u, T)\end{aligned}$$

4.2 Typing Rules

The following rules define the type system for lightweight `muse`. The rules are defined in terms of the type of a term t under the type environment Γ.

4.2.1 LVal

Variables are directly typed according to the type environment Γ. Dereferences are typed differently according to the type of the variable it refers to. A de-reference to a box type is trivially typed according to the nested type. A de-reference to a borrow type is typed according to the type of the borrowed variable.

$$\frac{\Gamma(x) = T_\perp}{\Gamma \vdash x : T_\perp \dashv \Gamma} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash w : \text{box } T_\perp \dashv \Gamma}{\Gamma \vdash *w : T_\perp \dashv \Gamma} \quad \text{(T-DerefBox)}$$

$$\frac{\Gamma \vdash w : [\text{mut}]\ \text{ref u} \dashv \Gamma \qquad \Gamma \vdash u : T \dashv \Gamma}{\Gamma \vdash *w : T \dashv \Gamma} \quad \text{(T-DerefRef)}$$

4.2.2 Value

Featherweight muse only supports *int* as a *source-level* type for values, therefore all values are typed as integers, leaving the context unchanged.

$$\frac{}{\Gamma \vdash n : \mathit{int} \dashv \Gamma} \quad \text{(T-Val)}$$

4.2.3 Box

Encapsulation of terms is reflected in their type, represented as boxed types.

$$\frac{\Gamma \vdash t : T \dashv \Gamma'}{\Gamma \vdash \texttt{box}\ t : \text{box } T \dashv \Gamma'} \quad \text{(T-Box)}$$

4.2.4 Borrow

Immutable borrows require that the variable is not *readProhibited*. This is to ensure that we are not creating an immutable borrow to a variable that is already borrowed mutably.

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \neg\text{readProhibited}(\Gamma, w)}{\Gamma \vdash \textbf{ref}\ w : \text{ref } w \dashv \Gamma} \quad \text{(T-Bor)}$$

4.2.5 Mutable Borrow

Mutable borrows require that the variable is not *writeProhibited*. This prevents mutable access to immutable borrows. Furthermore, mut(Γ, w) must hold, as it prevents gaining mutable access from an immutable borrow.

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \text{mut}(\Gamma, w) \qquad \neg\text{writeProhibited}(\Gamma, w)}{\Gamma \vdash \textbf{mut ref}\ w : \text{mut ref } w \dashv \Gamma} \quad \text{(T-MutBor)}$$

4.2.6 Copy

Copying operations are governed by the type's ability to be copied and whether readProhibited() holds.

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \text{isCopy}(T) \qquad \neg\text{readProhibited}(\Gamma, w)}{\Gamma \vdash \texttt{copy w} : T \dashv \Gamma} \quad \text{(T-Copy)}$$

4.2.7 Move

The movement of a variable is a destructive operation, which makes the type associated to the variable *undefined*.

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \neg\text{writeProhibited}(\Gamma, w) \qquad \Gamma' = \text{move\_var}(\Gamma, w)}{\Gamma \vdash \texttt{move w} : T \dashv \Gamma'} \quad \text{(T-Move)}$$

4.2.8 Declare

Declarations introduce new variables into the type environment, updating it accordingly.

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash t : T \dashv \Gamma' \qquad \Gamma'' = \Gamma'[x \mapsto T]}{\Gamma \vdash \textbf{let } [\textbf{mut}]\ x = t : \epsilon \dashv \Gamma''} \quad \text{(T-Declare)}$$

4.2.9 Assign

Assignments update the given type of a variable within the type environment, and may drop the previous value.

$$\frac{\begin{array}{c}\Gamma \vdash w : T_\perp \dashv \Gamma \qquad \Gamma \vdash t : T' \dashv \Gamma' \qquad \mathit{shape\_compatible}(\Gamma', T_\perp, T') \\ \Gamma'' = \mathrm{write}(\Gamma', w, T') \qquad \neg\mathrm{writeProhibited}(\Gamma'', w)\end{array}}{\Gamma \vdash w = t : \epsilon \dashv \Gamma''} \quad \text{(T-Assign)}$$

4.2.10 Program

Programs are typed based on the sequence of their constituent terms and the resulting context changes.

$$\frac{T_1 \vdash t_1 : T_1 \dashv \Gamma_1 \qquad \cdots \qquad T_n \vdash t_n : T_n \dashv \Gamma_n + 1}{\Gamma \vdash \overline{t} : T \dashv \Gamma_{n+1}} \quad \text{(T-Seq)}$$

Chapter 5

Operational semantics

The execution of `muse` is defined by small-step operational semantics. These are expressed as a set of reduction rules and showcase how a program is reduced to a simpler form. As stated before, to focus on the core mathematical features of borrow checking and ownership rules, we have omitted lifetimes from sections 5, 4 and 6. The full formalization of the language, including lifetimes, can be found in the appendix.

5.1 Preliminaries

Before defining the reduction rules, we need to define some helper functions that will be necessary for accessing and manipulating the program state.

5.1.1 Loc

Loc is a partial function that returns the location of a variable in the program state. Given a program state S and a variable w, the location of w is defined as:

$$loc(S, x) = \ell_x$$
$$loc(S, *w) = \ell \text{ where } loc(S, w) = \ell \text{ and } S(\ell_w) = \ell^*$$

5.1.2 Read

Read is a partial function that returns the value of a variable in the program state. Given a program state S and a variable w, the value of w is defined as:

$$read(S, w) = S(\ell_w) \text{ where } loc(S, w) = \ell_w$$

5.1.3 Write

Write is a partial function that updates the value of a variable in the program state. Given a program state S, a variable w, and a value v.

$$write(S, w, v) = S[\ell_w \mapsto v] \text{ where } loc(S, w) = \ell_w$$

5.1.4 Drop

Drop is a function that removes a value from the program state. Given a program state S and a value v, *Drop* is defined as:

$$drop(S, \theta) = S$$
$$drop(S, v) = drop(S \setminus \{\ell \mapsto v\}, v) \text{ where } S(\ell) = v$$

$$\frac{|\bar{t}| > 1 \qquad \bar{t} = t, \bar{t}' \qquad S \cdot t \to S' \cdot t'}{S \cdot \bar{t} \to S' \cdot t', \bar{t}'} \quad \text{(R-Program)} \qquad \frac{S \cdot t \to S' \cdot t'}{S \cdot E[t] \to S' \cdot E[t']} \quad \text{(R-Sub)}$$

$$\frac{v = read(S, w) \qquad S' = write(S, w, \bot)}{S \cdot \texttt{move x} \to S' \cdot v} \quad \text{(R-Move)} \qquad \frac{v = read(S, w)}{S \cdot \texttt{copy w} \to S \cdot v} \quad \text{(R-Copy)}$$

$$\frac{\ell_w = loc(S, w)}{S \cdot \texttt{[mut] ref w} \to S \cdot \ell_w} \quad \text{(R-Bor)} \qquad \frac{\ell_n^\bullet \notin dom(S) \qquad S' = S[\ell_n^\bullet \mapsto v]}{S \cdot \texttt{box v} \to S' \cdot \ell_n^\bullet} \quad \text{(R-Box)}$$

$$\frac{S' = S[\ell_x \mapsto v]}{S \cdot \texttt{let mut x = v} \to S' \cdot \epsilon} \quad \text{(R-Declare)}$$

$$\frac{v' = read(S, w) \qquad S' = drop(S, v') \qquad S'' = write(S', w, v)}{S \cdot w = v \to S'' \cdot \epsilon} \quad \text{(R-Assign)}$$

Figure 5.1: FM reduction rules

5.2 Program

Program Reduction is a relation on pairs of states and programs $S \cdot P \to S' \cdot P'$, where *State* S is a map from $r \mapsto v$.

$$\frac{|\bar{t}| > 1 \qquad \bar{t} = t, \bar{t}' \qquad S \cdot t \to S' \cdot t'}{S \cdot \bar{t} \to S' \cdot t', \bar{t}'} \quad \text{(R-Program)}$$

5.3 Term

Term Reduction is a relation on pairs of states and terms $S \cdot t \to S' \cdot t'$.

Sub-term reduction employs a single rule *R-Sub* to reduce the sub term within an *Evaluation Context*.

$$\frac{S \cdot t \to S' \cdot t'}{S \cdot E[t] \to S' \cdot E[t']} \quad \text{(R-Sub)}$$

Evaluation contexts are defined as follows:

$$E \quad := \square \mid \textbf{move } E \mid \textbf{copy } E \mid \textbf{[mut] ref } E \mid \textbf{box } E \mid \textbf{let mut } w = E \mid w = E$$

5.3.1 Move

Move semantics are used to transfer ownership of a value from one variable to another. The value is read from the source variable and written to the destination variable. The source variable is then dropped from the program state.

$$\frac{v = read(S, w) \qquad S' = write(S, w, \bot)}{S \cdot \texttt{move x} \to S' \cdot v} \quad \text{(R-Move)}$$

5.3.2 Copy

Copy semantics are used to duplicate a value. The value is read from the source variable and written to the destination variable. The source variable is not dropped from the program state.

$$\frac{v = read(S, w)}{S \cdot \texttt{copy w} \to S \cdot v} \quad \text{(R-Copy)}$$

5.3.3 Borrow

A Borrow is a controlled reference to a value. The R-Bor rule returns a fresh reference to the location of the variable in the program state.

$$\frac{\ell_w^\circ = loc(S, w)}{S \cdot \texttt{[mut] ref w} \rightarrow S \cdot \ell_w^\circ} \quad \text{(R-Bor)}$$

5.3.4 Box

A Box is type that holds a heap allocated value. The R-Box rule creates a fresh *owned reference* of the value in the program state.

$$\frac{\ell_n^\bullet \notin dom(S) \qquad S' = S[\ell_n^\bullet \mapsto v]}{S \cdot \texttt{box v} \rightarrow S' \cdot \ell_n^\bullet} \quad \text{(R-Box)}$$

5.3.5 Declare

A declaration introduces a new variable into the environment. Assigning the value to that variable in the program state.

$$\frac{S' = S[\ell_x \mapsto v]}{S \cdot \texttt{let mut x = v} \rightarrow S' \cdot \epsilon} \quad \text{(R-Declare)}$$

5.3.6 Assign

An assignment updates the value of a variable in the environment. Notice that the previous value assigned to the variable is dropped from the program state entirely, it will no longer live in memory.

$$\frac{v' = read(S, w) \qquad S' = drop(S, v') \qquad S'' = write(S', w, v)}{S \cdot w = v \rightarrow S'' \cdot \epsilon} \quad \text{(R-Assign)}$$

5.4 Worked example

The following example showcases an erroneous program. The program attempts to assign a value to a variable that has already been moved.

The initial environment is defined as a set of terms P_1 and the *Program State*. Note that $\emptyset$ has been used to denote an empty map.

$$\textit{Program State} = \emptyset,\ P_1 = \{\texttt{let mut x = box 5, let mut y = move x, let mut z = move x}\}$$

The reduction process starts by using R-Program to begin reducing the first term.

$$\frac{|P_1| > 1 \qquad P_1 = \texttt{let mut x = box 5}, P_2 \qquad (5.2)}{\emptyset \cdot P_1 \rightarrow \{\ell_1 \mapsto 5\} \cdot \texttt{let x =}\ \ell_1, P_2} \quad \text{(R-Program)} \qquad (5.1)$$

The first term is initially reduced by using R-Sub, to reduce the sub term `Box(5)` within the declaration of x.

$$\frac{(5.3)}{\emptyset \cdot \texttt{let mut x = [box 5]} \rightarrow \{\ell_1 \mapsto 5\} \cdot \texttt{let mut x =}\ [\ell_1]} \quad \text{(R-Sub)} \qquad (5.2)$$

Which is then reduced to a reference to 5 in the heap by R-Box:

$$\frac{\ell_1 \not\subset dom(\emptyset) \qquad \{\ell_1 \mapsto 5\} = \emptyset[\ell_1 \mapsto 5]}{\emptyset \cdot \texttt{box 5} \rightarrow \{\ell_1 \mapsto 5\} \cdot \ell_1} \quad \text{(R-Box)} \qquad (5.3)$$

We continue the reduction by using R-Program again to reduce the current environment:

$$\frac{|\texttt{let mut x = } \ell_1, P_2| > 1 \qquad (5.5)}{\{\ell_1 \mapsto 5\} \cdot \texttt{let mut x = } \ell_1, P_2 \rightarrow \{\ell_1 \mapsto 5, \ell_x \mapsto \ell_1\} \cdot \epsilon, P_2} \quad \text{(R-Program)} \qquad (5.4)$$

The first term is then finally reduced using R-Declare.

$$\frac{\{\ell_1 \mapsto 5, \ell_x \mapsto \ell_1\} = \{\ell_1 \mapsto 5\}[\ell_x \mapsto \ell_1]}{\{\ell_1 \mapsto 5\} \cdot \texttt{let mut x = } \ell_1 \rightarrow \{\ell_1 \mapsto 5, \ell_x \mapsto \ell_1\} \cdot \epsilon} \quad (R - Declare) \qquad (5.5)$$

The environment after:

$$\textit{Program State} = \{\ell_1 \mapsto 5, \ell_x \mapsto \ell_1\},\ P_2 = \{\texttt{let mut y = move x, let mut z = move x}\}$$

Moving on to the second term, we again start by using R-Program.

$$\frac{|P_2| > 1 \qquad P_2 = \texttt{let mut y = move x}, P_3 \qquad (5.7)}{\{\ell_1 \mapsto 5, \ell_x \mapsto \ell_1\} \cdot P_2 \rightarrow \{\ell_1 \mapsto 5\} \cdot \texttt{let y = } \ell_1, P_3} \quad \text{(R-Program)} \qquad (5.6)$$

The second term is initially reduced by using R-Sub, to reduce the sub term `x` within the declaration of y.

$$\frac{(5.8)}{\{\ell_1 \mapsto 5, \ell_x \mapsto \ell_1\} \cdot \texttt{let mut y = move x} \rightarrow \{\ell_1 \mapsto 5\} \cdot \texttt{let mut y = } [\ell_1]} \quad \text{(R-Sub)} \qquad (5.7)$$

Here `x` is a box type, it does not implement the `Copy` trait, therefore we use R-move to reduce it to a value using move semantics.

$$\frac{\ell_1 = \textit{read}(\{\ell_1 \mapsto 5, \ell_x \mapsto \ell_1\}, x) \qquad \{\ell_1 \mapsto 5\} = \textit{write}(\{\ell_1 \mapsto 5, \ell_x \mapsto \ell_1\}, x, \bot)}{\{\ell_1 \mapsto 5, \ell_x \mapsto \ell_1\} \cdot \texttt{move x} \rightarrow \{\ell_1 \mapsto 5\} \cdot \ell_1} \quad \text{(R-move)} \qquad (5.8)$$

We continue the reduction of the second term by using R-Program again.

$$\frac{|\texttt{let mut y = } \ell_1, P_3| > 1 \qquad (5.10)}{\{\ell_1 \mapsto 5\} \cdot \texttt{let mut y = } \ell_1, P_3 \rightarrow \{\ell_1 \mapsto 5, \ell_y \mapsto \ell_1\} \cdot \epsilon, P_3} \quad \text{(R-Program)} \qquad (5.9)$$

The second term is then reduced to its final step by using R-Declare.

$$\frac{\{\ell_1 \mapsto 5, \ell_y \mapsto \ell_1\} = \{\ell_1 \mapsto 5\}[\ell_y \mapsto \ell_1]}{\{\ell_1 \mapsto 5\} \cdot \texttt{let mut y = } \ell_1 \rightarrow \{\ell_1 \mapsto 5, \ell_y \mapsto \ell_1\} \cdot \epsilon} \quad \text{(R-Declare)} \qquad (5.10)$$

The environment after:

$$\textit{Program State} = \{\ell_1 \mapsto 5, \ell_y \mapsto \ell_1\},\ P_3 = \{\texttt{let mut z = move x}\}$$

Notice how the variable `x` has now been dropped from the program state, therefore we cannot assign it to z in the next term. We can continue the example to show this.

$$\frac{|P_3| > 1 \qquad P_3 = \texttt{let mut z = move x}, P_4 \qquad (5.12)}{\{\ell_1 \mapsto 5, \ell_y \mapsto \ell_1\} \cdot P_3 \rightarrow \{\ell_1 \mapsto 5\} \cdot \texttt{let z = } \ell_1, P_4} \quad \text{(R-Program)} \qquad (5.11)$$

The third term is initially reduced by using R-Sub, to reduce the sub term `x` within the declaration of z.

$$\frac{(5.13)}{\{\ell_1 \mapsto 5, \ell_y \mapsto \ell_1\} \cdot \texttt{let mut z = } [x] \rightarrow \{\ell_1 \mapsto 5\} \cdot \texttt{let mut z = } [\ell_1]} \quad \text{(R-Sub)} \qquad (5.12)$$

At this stage of the reduction process, the R-Move rule cannot be applied, as $read(S, x)$ would result in an error, as x is no longer in the program state. Therefore, the reduction process would terminate here.

$$\frac{v = read(S, w) \qquad S' = write(S, w, \bot)}{\{\ell_1 \mapsto 5, \ell_y \mapsto \ell_1\} \cdot \texttt{move x} \to \text{ERROR}} \quad \text{(R-Move)} \tag{5.13}$$

Chapter 6

Properties

6.1 Valid States

Before establishing the main lemmas progress and preservation, we must first define what it means for a program state to be valid. A program state is considered valid if it satisfies the following definition:

Definition 1 (Valid State)**.** Let $S \cdot t$ be a program state where $\overline{v} \in t$ is the sequence of distinct values contained in any $\ell^\bullet \in dom(S)$. Then S is valid when $\neg\exists_{i,j} \cdot (i \neq j \wedge \exists_{\ell^\bullet} \cdot (v_i = v_j = \ell^\bullet))$.

This definition ensures that a valid store cannot hold owning references to the same value. This is to prevent the creation of multiple owning references to the same value, which would violate the ownership semantics of the language.

6.2 Safe Abstraction

To ensure that the type environment given for a program store is a *safe abstraction*, we must define the following properties:

Definition 2 (Valid Type)**.** Let S be a program store, $v^\perp$ a partial value and $T_\perp$ a partial type, then v is abstracted by T in S, denoted as $S \vdash v \sim T$, according to the following rules:

$$\frac{}{S \vdash \epsilon \sim \epsilon} \text{ (V-Epsilon)} \qquad \frac{}{S \vdash n \sim int} \text{ (V-Int)} \qquad \frac{}{S \vdash \perp \sim \lfloor T \rfloor} \text{ (V-Undef)}$$

$$\frac{loc(S, w) = \ell}{S \vdash \ell^\circ \sim \text{[mut] ref w}} \text{ (V-Bor)} \qquad \frac{S(\ell^\cdot) = v^\perp \qquad S \vdash v \sim T}{S \vdash \ell^\cdot \sim \text{box T}} \text{ (V-Box)}$$

Definition 3 (Safe Abstraction)**.** Let S be a program store and Γ a type environment then S is *safely abstracted* by Γ, denoted $S \sim \Gamma$ if $dom(S) = dom(\Gamma)$ and for all x $\in$ dom(Γ), $S \vdash v^\perp \sim T_\perp$ where $S(\ell_x) = v^\perp$ and $\Gamma(x) = T_\perp$.

This definition ensures that the type environment is a safe abstraction of the program store, meaning that the type of each value in the store is correctly abstracted by the type in the type environment.

6.3 Borrow Checking

Borrow checking is a key feature of the type system that ensures that references are used correctly, and at any point in the runtime every borrow is valid.

Definition 4 (Well-formed Type Environment)**.** A type environment Γ is considered well-formed if for all $x \in dom(\Gamma)$ and w where $\Gamma \vdash x \rightsquigarrow$ [mut] ref w and $\Gamma(x) = v^\perp$, $w \in dom(\Gamma)$ we have $\Gamma \vdash w : T$.

This establishes that every variable that contains a reference, either mutable or immutable, is pointing to a valid location in the type environment. This is to ensure that there are no dangling references in the type environment.

6.4 Progress and Preservation

We can now define the progress lemma, which determines that a well-typed program that has not terminated will always be able to execute another step.

Lemma 1 (Progress)**.** Let $S_1 \cdot t_1$ be a *valid state*; let Γ_1 be a *well-formed typing environment* where $S_1 \sim \Gamma_1$; Let Γ_2 be a typing environment and let T be a type. If $\Gamma_1 \vdash t_1 : T \dashv \Gamma_2$ then either $S_1 \cdot t_1 \to S_2 \cdot t_2$ for some $S_2 \cdot t_2$ or $S_1 \cdot t_1$ is a terminal state (i.e a value).

We can also define the preservation lemma, which asserts that a well-typed program will remain well-typed program after any number of steps of execution.

Lemma 2 (Preservation)**.** Let $S_1 \cdot t$ be a valid program state and $S_2 \cdot v$ be a terminal state. Let Γ_1 be a well-formed typing environment where $S_1 \sim \Gamma_1$; Let Γ_2 be a typing environment. If $\Gamma_1 \vdash t : T \dashv \Gamma_2$ and $S_1 \cdot t \rightsquigarrow S_2 \cdot v$ then S_2 remains valid where $S_2 \sim \Gamma_2$ and $S_2 \vdash v \sim T$.

Here it is important to note that the preservation lemma considers the reduction of an entire term at a time (denoted by $\rightsquigarrow$) rather than a single step of reduction. This is due to the flow typing nature of the language, where $\Gamma_1 \vdash t : T \dashv \Gamma_2$ produces Γ_2 which may not be a *safe abstraction* during intermediate steps.

6.5 Type and Borrow Safety

Finally, we can establish a type and borrow safety theorem which states that a well-typed program is guaranteed to continue executing until a terminal state is reached.

Lemma 3 (Type and Borrow Saftey)**.** Let $S_1 \cdot t_1$ be a valid program state; let Γ_1 be a well-formed typing environment where $S_1 \sim \Gamma_1$; let Γ_2 be a typing environment and let T be a type. If $\Gamma_1 \vdash t : T \dashv \Gamma_2$ then $S_1 \cdot t \rightsquigarrow S_2 \cdot v$ for some terminal state $S_2 \cdot v$.

Chapter 7

Implementation

In this section we will cover an implementation of an FM interpreter. The interpreter is divided into four main stages: tokenizing, parsing, type-checking and evaluation. The type-checking and evaluation stages are implemented in a way that aligns with the type system and operational semantics defined in the previous sections.

It is important to note that the implementation covers the entire specification of FM, including the extension for functions defined in Section 8.1, and the subsequent updated rules containing lifetimes that can be found in appendices B and C. As previously discussed in Section 3, there is a slight difference between the syntax defined in this paper and the syntax used in the implementation. The interpreter is able to infer whether a variable ought to be moved or copied in the type checking step, and is able to propagate that information to the evaluation step. This is done by adding a `copyable` field to the `Variable` struct, which is set during the type checking step by calling the `copyable()` on a `Type`. Therefore, there is no need to explicitly declare whether a variable should be moved or copied in the syntax.

7.1 Stack Overview

It was decided to implement the interpreter in `rust` as I am confident with the language, and it is a systems programming language that is both safe and efficient. It also has a flexible and expressive type system and a powerful macro system that can be used to significantly simplify the implementation of the interpreter. The entire interpreter is implemented from scratch, with no external libraries used.

7.2 Lexer

The lexer is responsible for converting the source code into an array of tokens. As shown in FigureD.1, tokens are implemented as an enum where each variant represents a type of token. Some token variants contain additional data, for example the variant `Token::NumericLiteral(i64)` contains an integer holding the value of the numeric literal.

The lexer is implemented as a struct that contains the source code, the current position in the source code, and the current line and column. It implements a method called `tokenize` which returns a vector of tokens with the given source code. The tokenization process is implemented as a loop that iterates over the source code, matching string slices, consuming characters and producing tokens. The structure of the lexer is shown in FigureD.2.

7.3 Parser

The parser is responsible for converting the array of tokens into an abstract syntax tree (AST). Within the interpreter an AST is defined as a composite trait of the `TypeCheck` and `Evaluate` traits. These sub-traits define the methods that are required to type check and evaluate a term. The `Term` enum is defined as a recursive data structure that represents the syntax of the language. Each variant of the enum represents a different kind of term, such as a variable, a value, a box, a reference, or a let binding. Other structs that implement the `AST` trait are `Value`, and `LVal`. `Reference` is a struct that represents a reference to a location in memory.

The parser is implemented as a struct that contains the array of tokens, the current position in the array, and the current token. It implements a method called `parse` which returns an AST generated

from the given array of tokens. The parser is implemented as a recursive descent parser, with each non-terminal in the grammar being implemented as a function. The parser also implements a number of helper functions that are used to consume tokens and check for errors.

7.4 Type Checking

The type checker is responsible for ensuring that the program is well-typed and asserting borrow safety and ownership. Types are trivially defined as an enum, implementing a multitude of methods that are used for enforcing the borrow checker rules. The Slot struct contains a value and a lifetime, and is used for both the `Type` and `Value` types, as seen in FigureD.8.

We also define the type environment as a struct that contains a `HashMap` from variables to tuples of type and lifetime. It implements a number of methods that are used to update the type environment and check for errors. We also implement a series of helper methods, following the definition of the type system. Type checking is implemented for each struct that implements the `AST` trait. The type checking process is implemented as a series of recursive functions that traverse the AST and update the type environment. The type checking process also checks for errors and returns an error message if the program is not well-typed.

7.5 Execution

Before we can implement the execution process, we need to define the state of the program. The state of a program is defined as a struct that contains a vector of stack-frames and the store. The `StackFrame` contains a map from variables to locations where each frame represents a unique lifetime. The `Store` contains a `HaskMap` from locations to values, this represents the runtime memory of the program. `State` also implements a number of methods that are used to update the program state and check for errors.

The Execution process is responsible for evaluating the AST and reducing terms step by step to a final result. It is implemented as a series of recursive functions that traverse the AST and produce a final result. This is performed for each struct that implements the `AST` trait.

We finally have the pieces to implement the interpreter. It starts by performing an initial type check on the program, then iterates over each term in the program, asserting progress and preservation, type checking and evaluating each term. The interpreter returns an error if any of the assertions fail, otherwise it returns a final value.

Chapter 8

Extensions

8.1 Functions

Featherweight `muse` can be extended to support functions. To do this, it is necessary to extend the syntax of FM by defining and modifying new terms and types.

$$
\begin{array}{lll}
\text{Term} & t & := \ldots \mid x(\overline{v}) \\
\text{Function} & F & := \textbf{fn}\ x'(\overline{x : T})\ [\ : T'\ ]\ \{\overline{t}\}^{l} \\
\text{Program} & P & := \overline{t|F} \\
\text{Type} & T & := \ldots \mid (\overline{T}) : T
\end{array}
$$

Figure 8.1: Extended *muse* Syntax

Arguments are defined as a name, type pair. Function declarations are defined as a term that contains a set of arguments, a return type, and a body, a body is defined as a set of terms paired with a lifetime. Note that this is the first time we have introduced lifetimes into the syntax. Lifetimes are used to track scopes and ensure that no references outlive their scope. Programs are now defined as a sequence of terms and functions. Types are extended to include function types, which are defined as a set of types for arguments and a return type. To simplify the breadth of this extension we will focus solely on defining the operational semantics and type system for functions and function calls, and not on the rest of the core language. The fully extended syntax, typing rules and reduction rules can be found in the appendices A, B and C respectively.

Before we go on to define the operational semantics and type system extensions for functions we must first trivially modify the program state to include a map of function declarations D, where $x \mapsto \lambda$ and λ is defined as a pair consisting of a set of arguments $\overline{x}$ and a set of terms with a lifetime $\{\overline{t}\}^{l}$.

$$\frac{D' = D[x' \mapsto (\overline{x}, \{\overline{t}\}^{m})]}{\langle D, S \cdot \textbf{fn}\ x'(\overline{x : T})\ [\ : T'\ ]\ \{\overline{t}\}^{m} \rightarrow D', S \cdot \epsilon\rangle^{l}} \quad \text{(R-Function)}$$

$$\frac{D(x) = (\overline{x}, \{\overline{t}\}^{m}) \qquad S' = S[\ell_{n::x} \mapsto \langle v_n\rangle^{m}] \qquad \delta\mathcal{L} = 1}{\langle D, S \cdot x(\overline{v}) \rightarrow D, S' \cdot \{\overline{t}\}^{m}\rangle^{l}} \quad \text{(R-Call)}$$

$$\frac{\langle D, S \cdot \overline{t} \rightarrow D', S' \cdot v, \emptyset\rangle^{l} \qquad S'' = \text{drop}(S', m)}{\langle D, S \cdot \{\overline{t}\}^{m} \rightarrow D, S'' \cdot v\rangle^{l}} \quad \text{(R-Block)}$$

Figure 8.2: Extended reduction rules

The reduction rule for function declarations is simple, it adds the function to the function declaration map. The reduction rule for function calls is more complex, it requires looking up the function in the function declaration map, adding each argument to the state and then evaluating the body of the function with the arguments passed to the function call. The lifetime context $\mathcal{L}$ of the function call is incremented by one. R block is a new reduction rule that is used to evaluate the body of a function. It evaluates the body of the function and then drops the lifetime context of the function from the state.

```
// comments use double slash
let x: int = 4

let y = 4.0 //  <- type is infered here

// all variables in muse are immutable by default
y -= 1 // <- ERROR y is not mutable

let mut z = 1
z += 1
```

Figure 8.4: Basic syntax

$$\frac{\Gamma' = \Gamma[x' \mapsto (\overline{T}) : T'] \qquad \Gamma_{block} = \Gamma'[\overline{x \mapsto T}] \qquad \Gamma_{block} \vdash \overline{t} : T'' \dashv \Gamma'_{block} \qquad T = T''}{\langle \Gamma \vdash \textbf{fn}\ x'(\overline{x : T})\ [\ : T'\ ]\ \{\overline{t}\}^l : (\overline{T}) : \epsilon \dashv \Gamma' \rangle^l} \quad \text{(T-Function)}$$

$$\frac{\Gamma(x) = (\overline{T'}) : T' \qquad \forall i.(\text{shape\_compatible}(T_i, T'_i))}{\langle \Gamma \vdash x(\overline{T}) : T' \dashv \Gamma \rangle^l} \quad \text{(T-Call)}$$

Figure 8.3: Extended type rules

The typing rule for function declarations does the following: It adds the function to the type environment, and then creates a new type environment for type checking the body of the function which contains the arguments of the function. The typing rule for function calls is simpler, it looks up the function in the type environment and checks that the arguments passed to the function call are *shape comparable* with the arguments of the function.

8.2 Extending Featherweight muse to full muse

Given that we have a well-defined outlook into the syntax, semantics and type system of featherweight muse we can now try exploring what a full version of `muse` would look like. The following section will do just that, utilizing the insights gained during this project to extend muse to include basic core features found in other languages such as arithmetic operations, control flow and loops, structs, enums, and pattern matching. In addition, the type system is extended to include type inference, generics and structural typing.

8.2.1 Syntax

`muse`'s syntax should be designed to be simple and easy to learn, providing syntactical sugar without straying too far away from classic programming standards. It should be obvious to a developer what `muse` code is doing at a glance. We do this by following pre-established standards within other languages and focusing on readability over zero-cost abstractions.

Figures 8.4, 8.6 and 8.5 showcase basic features of `muse`. Variables are declared using the `let` keyword, and types are inferred by the compiler. All variables are immutable by default, and must be declared as mutable using the `mut` keyword. Control flow and loops are similar to most C based languages, with the addition of pattern matching, similar to `Rust`. Pattern matching in muse is exhaustive, meaning all cases must be covered. Other syntactical sugar such as list comprehensions, similar to `python`, and slices are also supported.

Functions in `muse` are defined using the `fn` keyword, and can take any number of arguments. muse implements *local type inference*, requiring all function parameters to be properly typed and provide a return type when necessary. Functions are *first class members* in `muse`, meaning they can be passed as parameters and returned from functions.

```
fn add(x: int, y: int): int {
    return x + y
}

let three = add(1, 2)

// functions are first class members
let adder = add
let four = adder(2, 2)

// functions can be passed as paramaters
fn apply_math(f: (int, int): int, x: int, y: int): int {
    return f(x, y)
}

// here we pass the add function as a paramater
// adder is structurally typed as a binary_op
let five = apply_math(adder, 2, 3)
```

Figure 8.5: Function syntax

```
let x = 1

if x == 1 {
    print("x is 1")
} else if x == 2 {
    print("x is 2")
} else {
    print("x is not 1 or 2")
}

// slices can be used to generate simple iterators
let evens = [0, 2..10]

// for loops
for i in evens {
    print(i)
}

// list comprehensions are also supported
let squares = [x * x for x in [0..10]]

// while loops
while x < 10 {
    x += 1
}

// match statements, similar to rust
// all cases must be covered
match x {
    1 => print("x is 1"),
    2 => print("x is 2"),
    _ => print("x is not 1 or 2")
}
```

Figure 8.6: Control flow syntax

```
struct Point {
    x: int,
    y: int
}

let origin = Point { x: 0, y: 0 }

// structs can be nested
struct Rectangle {
    top_left: Point,
    bottom_right: Point
}

let mut rect = Rectangle {
    top_left: origin,
    bottom_right: Point { x: 10, y: 10 }
}

// structs can be destructured
let { top_left, bottom_right } = rect

// structs can be updated
rect.top_left.x = 1
print(rect.top_left.x) // 1

// structs can have methods
struct Point {
    x: int,
    y: int
    fn new(x: int, y: int) -> Point {
        return Point { x: x, y: y }
    }
}
```

Figure 8.7: Struct syntax

8.2.2 Type system

`muse` implements a strong, static type system. The type system is designed to be simple, yet expressive. It should be easy to reason about the types of variables and functions in `muse`. The type system supports generics, where types can be parameterized, and structural typing, where types are inferred at compile-time based on the operations that are performed on them, similar to `Go`.

The primary way to create types in `muse` is using a `struct`. A struct is a collection of named fields, each with a type. Structs can be nested, and can have methods. Enums are also supported, providing a way to define a type that can be one of several variants. Interfaces are used to define a contract for a type. A type that implements an interface must provide an implementation for all the methods defined in the interface. Interfaces are similar to traits in `Rust`. Examples of these features can be seen in Figures 8.7, 8.8, and 8.9.

`muse` also provides a simple generic syntax, similar to most modern languages. Generics allow for the definition of functions and structs that can work with any type, as long as the type meets certain constraints. In `muse`, generics are structurally typed, meaning that as long as the type meets the constraints, it can be used with the generic function or struct.

```
// enum variants can be defined in 3 different ways: Named, Struct, Tuple
enum Event {
    Quit,
    Move { x: int, y: int },
    Write(String),
    ChangeColor(int, int, int),
}

let e = Event::Write("Hello World")

// match expressions can be used to destructure enums
match e {
    Event::Quit => print("Quit"),
    Event::Move { x, y } => print(f"Moved to {x} {y}"),
    Event::Write(s) => print(f"Written: {s}"),
    Event::ChangeColor(r, g, b) => print(f"Changed Color to {r}, {g}, {b}")
}
```

Figure 8.8: Enum syntax

```
interface Drawable {
    fn draw(self)
}

struct Circle: Drawable {
    radius: int
    fn draw(self) {
        print(f"Drawing Circle with radius {self.radius}")
    }
}

let c = Circle { radius: 10 }
c.draw() // Drawing Circle with radius 10
```

Figure 8.9: Interface syntax

```
// generic structs can be defined
struct Point<T impl Add> {
    x: T,
    y: T
    // in this case Add is a type that is
    // bound to the built-in + operator
    fn +(self, other: Point<T>) -> Point<T> {
        return Point { x: self.x + other.x, y: self.y + other.y }
    }
}

fn add<T impl Add>(x: T, y: T): T {
    return x + y
}

let three = add(1, 2)
let four = add(2.0, 2.0)

// generic functions can be passed as paramaters
fn apply_math<T>(f: binary_op<T>, x: T, y: T): T {
    return f(x, y)
}

// here we pass the add function as a paramater
let five = apply_math(add, 2, 3)
```

Figure 8.10: Generic syntax

```
let mut x = Point { x: 0, y: 0 } // x is the owner of Point { x: 0, y: 0 }
let mut y = x // x is moved to y, y is the owner of Point { x: 0, y: 0 }
let mut z = x // y is moved to z, z is the owner of Point { x: 0, y: 0 }

let a = x // ERROR x is being used after it has been moved
```

Figure 8.11: Ownership example

8.2.3 Memory Management

As explored within *featherweight muse*, `muse` provides a memory management model that utilizes its type system to statically analyse memory aliasing, similar to `Rust`. `muse` uses a combination of ownership, borrowing, and lifetimes to ensure memory safety without the need for garbage collection. Ownership within `muse` requires that every value has a unique owner. When the owner goes out of scope, the associated value is dropped. This ensures that memory is always freed when it is no longer needed. Borrows within `muse` act as controlled references. Borrow checking allows for multiple immutable borrows to a value, but only one mutable borrow. This ensures that data cannot be mutated in one part of the program while it is being accessed elsewhere. Examples of ownership and borrowing can be seen in Figures 8.11 and 8.12.

8.2.4 Argument passing:

`muse` leverages different argument passing techniques to ensure memory safety and efficiency. The rules for argument passing are as follows:

```
let mut a = Point { x: 0, y: 0 };
let b =  ref a;
// b.x = 1; // ERROR b is a immutable reference to a
let mut c = a.clone();
{
    let d = mut ref c;
    // c.x = 2 // ERROR c is borrowed as mutable, cannot be accessed;
    d.x = 1;
}
c.y = 1;
```

Figure 8.12: Reference example

```
// Primitive types, and types that implement
// the Copy trait are passed by value
fn f(x: int) { ... }

// All other types are implicitlyn passed by
// an immutable reference, ownership is borrowed
fn g(x: Point) { ... }

// parameters can be explicitly passed by reference
fn h(x: ref int) { ... }

// here x is a mutable reference to an int
//ownership is borrowed
fn i(x: mut int) { ... }

//here x takes ownership of the int
// the caller is no longer the owner
fn j(x: owned int) { ... }
```

Figure 8.13: Argument passing example

Chapter 9

Evaluation

In this section, we evaluate the formalization and implementation of Featherweight `muse` by testing the properties defined in Section 6. We also evaluate the interpreter by testing the language with various programs to ensure that the interpreter is able to catch and provide the user with appropriate errors for programs that violate the type system. We also evaluate the code coverage of the interpreter to ensure that the majority of the code has been tested.

9.1 Properties

The properties of `muse` are designed to ensure that the language is memory safe, type safe, and efficient. To evaluate the properties defined in Section 6.4 , we can utilize the interpreter to assert the correctness of these properties on each step of the execution process. This was done by implementing the dynamic checks corresponding to each lemma and passing through the necessary states and environments to ensure that the properties hold within each step of the interpreter loop.

As you can see in Figure9.1, the `assert_progress` function closely follows the progress lemma, first checking the validity of the state, then checking the well-formedness of the type environment, and finally checking the safety of the abstraction. The function then type checks the term and evaluates it. If the term is a value, the function returns, otherwise it evaluates the term.

9.2 Testing

The programs written to test these properties were chosen to test the semantics and properties of FM. As the syntax of FM is stripped down to simple terms that only allow the most basic operations, the extent to which these properties could be evaluated was limited. However, the interpreter was able to assert that these properties hold for the given terms. One of the more complex programs that we were able to test can be seen in Figure9.2. This program showcases a function that performs an in-place swap of two integers. It displays the declaration of functions and variables, the passing of references to functions, the reassignment of variables, and the dereferencing of references. The program was able to be type checked and evaluated by the interpreter, and the properties of progress and preservation held during each step of execution.

To test the safety of the language, several "bad" programs, were written to test the interpreter. These programs attempt to perform operations that are not allowed by the type system. In these sort of cases, the interpreter should provide the user with an appropriate error when type checking the program and terminate. For example, the erroneous program `worked_example.mu` shown in Figure9.4, attempts to declare a variable using a value that has already been moved. The interpreter catches this, and correctly provides the user with an error: "Type error: Type of Box Numeric is undefined, indicating that it was moved". A set of these programs were written to test various violations of borrowing and ownership rules and the interpreter was able to catch and provide the user with the appropriate error message for all tests.

A full set of test cases can be found in the `tests` directory of the repository, and a table showing the objective of the test and intended results can be found in Figure9.3 and 9.4, every test passed.

9.3 Code Coverage

Using the Rust tool `cargo-llvm-cov` we were able to analyse the code coverage of the interpreter. The interpreter achieved 93.5% function code coverage, with the majority of the untested code being error handling code. The code coverage report can be seen in Figure9.5.

```
pub fn assert_progess(
    s1: State,
    t1: Term,
    g1: Typeenvironment,
    lifetime: usize
) -> Result<(), String> {
    if !valid_state(s1.clone(), t1.clone())? {
        return Err(
            "Invalid state".to_string()
        )
    }

    if !well_formed(g1.clone())? {
        return Err(
            "Type environment is not well formed".to_string()
        )
    }

    if !safe_abstraction(s1.clone(), g1.clone())? {
        return Err(
            "Type environment is not a safe abstraction of current state".to_string()
        )
    }

    t1.clone().type_check(g1, lifetime)?;

    match t1 {
        Term::Value(_) => {
            return Ok(());
        },
        _ => {
            t1.clone().evaluate(s1, lifetime)?;
            return Ok(());
        }
    }

}
```

Figure 9.1: Assertion funtion for progress

```
fn swap(mut ref a : int, mut ref b : int) {
    let mut c = *a
    *a = *b
    *b = c
}

let mut x = 0
let mut y = 1

swap(x , y)
```

Figure 9.2: swap.mu

Test Name	Objective	Expected Result
boxed_variable.mu	Test the creation of a boxed variable from a variable	{ x: 0, y: ref 0 }
double_box_deref.mu	Test the dereferencing of a double boxed variable	{ x: ref Undefined, y: ref 1 }
fn_borrow.mu	Test the borrowing of a variable within a function	{ x: ref 0, y: ref 0 }
fn_inplace.mu	Test the in-place mutation of a variable within a function	{ x: 5 }
fn_lifetime_transfer.mu	Test the transfer of a lifetime from a variable within a function, to a variable outside the function	{ x: 5 }
fn_transfer.mu	Test the passing through of a box from a variable to a function, and the transfer of the box to a new variable	{ x: Undefined, z: ref 5 }
immut_after_mut.mu	Test the creation of an immutable borrow to a mutable borrow	{ x: 0, y: ref 0, z: ref 0 }
multiple_move.mu	Testing the moving of a variable to multiple locations	{ res: ref5 }
reassign_after_move.mu	Test the reassignment of a variable after it has been moved	{ x: 1, y: 0 }
reassign_deref.mu	Testing the reassignment of a dereferenced variable	{ x: 4, y: ref 4 }
reassign_in_diff_scope.mu	Testing the reassignment of a variable in a different scope	{ x: 0, y: 1, w: 1 }
reassign_ref.mu	Testing the reassignment of a reference	{ x: 0, y: 1, z: ref 1 }
swap.mu	Test mutable references as function arguments, mutating them within the function scope and observing the changes outside the function scope	{ x: 1, y: 0 }

Figure 9.3: "Good" Test Cases

Test Name	Objective	Expected Result
double_mut_ref.mu	Test the creation of two mutable references to the same variable	Type error: Cannot create a mutable reference to x as it's already borrowed immutably
assign_borrowed.mu	Test the reassignment of an immutably borrowed variable	Type error: Cannot assign to borrowed reference: x
assign_mut_borrowed.mu	Test the reassignment of a mutably borrowed variable	Type error: Cannot assign to borrowed reference: x
bad_typing.mu	Test the incorrect reassignment to a variable	Type error: Incompatible types: box int and ref z
dec_after_partial_move.mu	Test the use of a partial move after dereferencing a double box int once	Type error: Type of box int is undefined, indicating that it was moved
function_incorrect_arg_count.mu	Test passing the incorrect number of arguments to a function	Type error: Incompatible argument count: expected 2, got 1
function_incorrect_arg_type.mu	Test passing the incorrect argument types to a function	Type error: Incompatible argument type: expected int, got box int
function_incorrect_return_type.mu	Testing mismatch between declared return type and function body return type	Type error: Unexpected return type: expected int, got box int
mut_after_immut.mu	Test the creation of a mutable borrow after creating an immutable borrow	Type error: Cannot create a mutable reference to x as it's already borrowed immutably
mut_from_immut.mu	Test gaining mutable access from an immutable borrow	Type error: Mutable reference cannot be created from immutable reference: y
worked_example.mu	Testing multiple the usage of a variable after moving	Type error: Type of box int is undefined, indicating that it was moved
func_not_def.mu	Test calling a function that is not defined	Type error: Function not defined: g()
func_dup_arg.mu	Test passing the same argument twice to a function	Type error: Duplicate argument in function declaration: x

Figure 9.4: "Bad" Test Cases

Filename	Function Coverage	Line Coverage
ast.rs	85.71% (18/21)	76.09% (35/46)
constants.rs	100.00% (1/1)	68.42% (13/19)
interpreter.rs	100.00% (2/2)	97.30% (36/37)
lexer.rs	100.00% (4/4)	97.83% (45/46)
parser.rs	100.00% (11/11)	93.45% (214/229)
properties.rs	100.00% (8/8)	81.25% (143/176)
reduction.rs	25.00% (1/4)	73.13% (98/134)
run_tests.rs	100.00% (3/3)	97.92% (94/96)
state.rs	96.67% (29/30)	83.72% (144/172)
token.rs	66.67% (2/3)	66.67% (2/3)
typecheck.rs	100.00% (6/6)	92.15% (223/242)
typing.rs	100.00% (30/30)	81.20% (190/234)
Totals	**93.50% (115/123)**	**86.26% (1237/1434)**

Figure 9.5: Code Coverage Report

9.4 Future Work

9.4.1 Extending support for non-termination

Currently, the *Type and borrow safety* property of the language requires termination. While this is fine for the calculus as it stands, adding support for loops and recursion would require extending the properties to support non-termination. Work has already been done regarding this in Payet *et al.* (2022), and it would be interesting to see how these ideas could be applied to FM.

9.4.2 Implementing non-lexical lifetimes

Lifetimes in FM are lexical, meaning that they tied to the scope in which they are declared. This is a limitation of the current implementation and non-lexical lifetimes, as discussed in Section 2.2, would allow for more flexible borrowing and ownership rules, and would allow for more complex programs to be written in FM.

Chapter 10

Conclusion

The project successfully presented a formalization of featherweight muse, covering the mathematics that define core features of the language, including the borrow checking based type system, operational semantics, and properties. Following this the core of the language was further extended to support functions and therefore scope-managed lifetimes. An interpreter was also implemented for the language, which was used to evaluate the properties and memory management analysis defined in the formalization. Given a suite of tests designed to evaluate the memory safety of the language, the interpreter was able to assert that the properties hold for each small step of the given test, and was able to catch and provide the user with appropriate errors for programs that violate the type system. Using the insights gained from the project, we moved forward to provide a deeper look into a fully-fledged version of muse.

In future work, we plan to extend the properties of muse to support non-termination, and re-formalize the memory management model to support non-lexical lifetimes. These extensions will provide a more comprehensive view of the language and its capabilities, and will allow for a more detailed analysis of the memory management techniques used in muse. In addition to this, we plan to expand Featherweight muse to support more advanced features, such as algebraic data types, structural typing and generics. These extensions will provide a more complete view of the language and its capabilities, and will serve as a stepping stone for further development of the muse language.

Chapter 11

Bibliography

Ancona, Davide *et al.* (July 2016). "Behavioral Types in Programming Languages". In: *Foundations and Trends® in Programming Languages* 3, pp. 95–230. DOI: 10.1561/2500000031.

Blackburn, Stephen M., Perry Cheng, and Kathryn S. McKinley (2004). "Myths and realities: the performance impact of garbage collection". In: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '04/Performance '04. New York, NY, USA: Association for Computing Machinery, pp. 25–36. ISBN: 1581138733. DOI: 10.1145/1005686.1005693. URL: https://doi.org/10.1145/1005686.1005693.

Crichton, Will, Gavin Gray, and Shriram Krishnamurthi (Oct. 2023). "A Grounded Conceptual Model for Ownership Types in Rust". In: *Proc. ACM Program. Lang.* 7.OOPSLA2. DOI: 10.1145/3622841. URL: https://doi.org/10.1145/3622841.

Fluet, Matthew, Greg Morrisett, and Amal Ahmed (2006). "Linear regions are all you need". In: *Programming Languages and Systems: 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006. Proceedings 15*. Springer, pp. 7–21.

Fu, Qiancheng and Hongwei Xi (2023). "Two-level Linear Dependent Type Theory". In: *ArXiv* abs/2301.08540.

Google (2021). *Memory Safety*. Accessed: 2024-04-22. URL: https://www.chromium.org/Home/chromium-security/memory-safety/.

Griesemer, Robert *et al.* (Nov. 2020). "Featherweight go". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA, pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3428217. URL: http://dx.doi.org/10.1145/3428217.

Hainry, Emmanuel and Romain Péchoux (2018). "Type-based complexity analysis of object-oriented programs". In: *ArXiv* abs/1802.03685.

Harper, Robert (2016). *Practical Foundations for Programming Languages*. 2nd. USA: Cambridge University Press. ISBN: 1107150302. DOI: 10.5555/3002812.

Igarashi, Atsushi, Benjamin C. Pierce, and Philip Wadler (May 2001). "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3, pp. 396–450. ISSN: 0164-0925. DOI: 10.1145/503502.503505. URL: https://doi.org/10.1145/503502.503505.

Jim, Trevor *et al.* (2002). "Cyclone: a safe dialect of C." In: *USENIX Annual Technical Conference, General Track*, pp. 275–288.

Jones, Richard, Antony Hosking, and Eliot Moss (2011). *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC.

Khan, Khaled *et al.* (Jan. 2012). "Improving Memory Management Security for C and C++". In: pp. 190–216. ISBN: 9781466615816. DOI: 10.4018/978-1-4666-1580-9.ch011.

Klabnik, Steve and Carol Nichols (2023). *The Rust programming language*. No Starch Press.

Nagarakatte, Santosh, Milo M. K. Martin, and Steve Zdancewic (2012). "Watchdog: Hardware for safe and secure manual memory management and full memory safety". In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 189–200. DOI: 10.1109/ISCA.2012.6237017.

Payet, Étienne, David J. Pearce, and Fausto Spoto (2022). "On the Termination of Borrow Checking in Featherweight Rust". In: *NASA Formal Methods*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Cham: Springer International Publishing, pp. 411–430. ISBN: 978-3-031-06773-0. DOI: 10.1007/978-3-031-06773-0_22.

Pearce, David J. (Mar. 2021). "A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust". In: *ACM Transactions on Programming Languages and Systems* 43.1, pp. 1–73. ISSN: 1558-4593. DOI: 10.1145/3443420. URL: http://dx.doi.org/10.1145/3443420.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. 1st. The MIT Press. ISBN: 0262162091.

Pizlo, Filip and Jan Vitek (2006). "An Empirical Evaluation of Memory Management Alternatives for Real-Time Java". In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, pp. 35–46.

Qin, Boqin *et al.* (2020). "Understanding memory and thread safety practices and issues in real-world Rust programs". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Rezaei, Mehran and Krishna M Kavi (2000). "Implementation technique of memory management". In: *Proceedings of the International Conference on Software Engineering Theory and Practice*.

RustLang (Aug. 2017). *The rust RFC book*. URL: https://rust-lang.github.io/rfcs/2094-nll.html.

Swamy, Nikhil *et al.* (2006). "Safe manual memory management in Cyclone". In: *Science of Computer Programming* 62.2, pp. 122–144.

Tofte, Mads and Jean-Pierre Talpin (1997). "Region-Based Memory Management". In: *Information and Computation* 132.2, pp. 109–176. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1996.2613. URL: https://www.sciencedirect.com/science/article/pii/S0890540196926139.

VanHattum, Alexa *et al.* (2022). "Verifying Dynamic Trait Objects in Rust". In: *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

Wason, R and P Kumar (2016). "A Novel Simulation of Memory Management in Computer Systems". In: *Journal of Computer Engineering* 18.1, pp. 22–29.

Weiss, Aaron *et al.* (2019). "Oxide: The Essence of Rust". In: *ArXiv* abs/1903.00982.

Williams, Christian and Michael Stay (2021). "Type Theory and its Meaning Explanations". In: *ArXiv* abs/2103.15312.

Xu, Hui *et al.* (2020). "Memory-Safety Challenge Considered Solved? An Empirical Study with All Rust CVEs". In: *ArXiv* abs/2003.03296.

Zhang, Yuchen *et al.* (2023). "Towards Understanding the Runtime Performance of Rust". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. ¡conf-loc¿, ¡city¿Rochester¡/city¿, ¡state¿MI¡/state¿, ¡country¿USA¡/country¿, ¡/conf-loc¿: Association for Computing Machinery. ISBN: 9781450394758. DOI: 10.1145/3551349.3559494. URL: https://doi.org/10.1145/3551349.3559494.

Appendices

Appendix A

Extended Syntax

Numeric Literal	n	$:= 0 \mid 1 \mid 2 \mid \ldots$
Reference	ℓ	$:= \ell^{\bullet}\ \ell^{\circ}$
Value	v	$:= n \mid r$
Partial Value	$v_{\perp}$	$:= v \mid \perp$
LVal	w	$:= x \mid *w$
Term	t	$:= w$
		$\mid v$
		$\mid$ **move** w
		$\mid$ **copy** w
		$\mid$ **box** t
		$\mid$ **[mut] ref** w
		$\mid$ **let mut** $w = t$
		$\mid w = t$
		$\mid \epsilon$
		$\mid x(\overline{v})$
Function	F	$:=$ **fn** $x'(\overline{x : T})$ $[\ : T'\ ]$ $\{\overline{t}\}^{l}$
Program	P	$:= \overline{t \mid F}$
Type	T	:= int | [mut] ref w | box T | $(\overline{T}) : T$
PartialType	$T_{\perp}$	$:= T \mid \lfloor T \rfloor$
Term	t	:= ... | $x(\overline{v})$

Figure A.1: Syntax of Featherweight `muse`

Appendix B

Extended Type Rules

$$\frac{\Gamma(x) = T_\perp}{\Gamma \vdash x : T_\perp \dashv \Gamma} \text{ (T-Var)} \qquad \frac{\Gamma \vdash w : T_\perp \dashv \Gamma}{\Gamma \vdash *w : T_\perp \dashv \Gamma} \text{ (T-DerefBox)} \qquad \frac{\Gamma \vdash w : T \dashv \Gamma}{\Gamma \vdash *w : T_\perp \dashv \Gamma} \text{ (T-DerefRef)}$$

$$\Gamma \vdash n : int \dashv \Gamma \text{ (T-Val)} \qquad \frac{\Gamma \vdash t : T \dashv \Gamma'}{\Gamma \vdash \texttt{box}\ t : \text{box}\ T \dashv \Gamma'} \text{ (T-Box)}$$

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \neg\text{readProhibited}(\Gamma, w)}{\Gamma \vdash \textbf{ref}\ w : \text{ref}\ w \dashv \Gamma} \text{ (T-Bor)}$$

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \text{mut}(\Gamma, w) \qquad \neg\text{writeProhibited}(\Gamma, w)}{\Gamma \vdash \textbf{mut ref}\ w : \text{mut ref}\ w \dashv \Gamma} \text{ (T-MutBor)}$$

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \text{isCopy}(T) \qquad \neg\text{readProhibited}(\Gamma, w)}{\Gamma \vdash \texttt{copy w} : T \dashv \Gamma} \text{ (T-Copy)}$$

$$\frac{\Gamma \vdash w : T \dashv \Gamma \qquad \neg\text{writeProhibited}(\Gamma, w) \qquad \Gamma' = \text{move\_var}(\Gamma, w)}{\Gamma \vdash \texttt{move w} : T \dashv \Gamma'} \text{ (T-Move)}$$

$$\frac{x \notin \text{dom}(\Gamma) \qquad \Gamma \vdash t : T \dashv \Gamma' \qquad \Gamma'' = \Gamma'[x \mapsto T]}{\Gamma \vdash \texttt{let mut}\ w = t : \epsilon \dashv \Gamma''} \text{ (T-Declare)}$$

$$\frac{\begin{array}{c}\Gamma \vdash w : T_\perp \dashv \Gamma \qquad \Gamma \vdash t : T' \dashv \Gamma' \qquad \mathit{shape\_compatible}(\Gamma', T_\perp, T') \\ \Gamma'' = \text{write}(\Gamma', w, T') \qquad \neg\text{writeProhibited}(\Gamma'', w)\end{array}}{\Gamma \vdash w = t : \epsilon \dashv \Gamma''} \text{ (T-Assign)}$$

$$\frac{T_1 \vdash t_1 : T_1 \dashv \Gamma_1 \qquad \cdots \qquad T_n \vdash t_n : T_n \dashv \Gamma_n + 1}{\Gamma \vdash \overline{t} : T \dashv \Gamma_{n+1}} \text{ (T-Seq)}$$

$$\frac{\Gamma' = \Gamma[x' \mapsto (\overline{T}) : T'] \qquad \Gamma_{block} = \Gamma'[\overline{x \mapsto T}] \qquad \Gamma_{block} \vdash \overline{t} : T'' \dashv \Gamma'_{block}}{\Gamma \vdash \textbf{fn}\ x'(\overline{x : T})\ [\ : T'\ ]\ \{\overline{t}\}^l : (\overline{T}) : \epsilon \dashv \Gamma'} \text{ (T-Function)}$$

$$\frac{\Gamma(x) = (\overline{T'}) : T' \qquad \forall i.(\text{shape\_compatible}(T_i, T'_i))}{\Gamma \vdash x(\overline{T}) : T' \dashv \Gamma} \text{ (T-Call)}$$

Figure B.1: FM typing rules

Appendix C

Extended Reduction Rules

$$\frac{|\overline{t}| > 1 \qquad \overline{t} = t, \overline{t}' \qquad D, S \cdot t \to D', S' \cdot t'}{\langle D, S \cdot \overline{t} \to D', S' \cdot t', \overline{t}' \rangle^l} \text{ (R-Program)} \qquad \frac{D, S \cdot t \to D', S' \cdot t'}{\langle D, S \cdot E[t] \to D', S' \cdot E[t'] \rangle^l} \text{ (R-Sub)}$$

$$\frac{v = read(S, w) \qquad S' = write(S, w, \bot)}{\langle D, S \cdot \texttt{move x} \to DS' \cdot v \rangle^l} \text{ (R-Move)} \qquad \frac{v = read(S, w)}{\langle D, S \cdot \texttt{copy w} \to D, S \cdot v \rangle^l} \text{ (R-Copy)}$$

$$\frac{\ell^\circ_w = loc(S, w)}{\langle D, S \cdot \texttt{[mut] ref w} \to D, S \cdot \ell^\circ_w \rangle^l} \text{ (R-Bor)} \qquad \frac{\ell^\bullet_n \notin dom(S) \qquad S' = S[\ell^\bullet_n \mapsto v]}{\langle D, S \cdot \texttt{box v} \to D, S' \cdot \ell^\bullet_n \rangle^l} \text{ (R-Box)}$$

$$\frac{S' = S[\ell_x \mapsto v]}{\langle D, S \cdot \texttt{let mut x = v} \to D, S' \cdot \epsilon \rangle^l} \text{ (R-Declare)}$$

$$\frac{v' = read(S, w) \qquad S' = drop(S, v') \qquad S'' = write(S', w, v)}{\langle D, S \cdot w = v \to D, S'' \cdot \epsilon \rangle^l} \text{ (R-Assign)}$$

$$\frac{D' = D[x' \mapsto (\overline{x}, \{\overline{t}\}^m)]}{\langle D, S \cdot \textbf{fn } x'(\overline{x : T}) \ [: T' \ ] \ \{\overline{t}\}^m \to D', S \cdot v \rangle^l} \text{ (R-Function)}$$

$$\frac{D(x) = (\overline{x}, \{\overline{t}\}^m) \qquad S' = S[\ell_{n::x} \mapsto \langle v_n \rangle^m] \qquad \delta\mathcal{L} = 1}{\langle D, S \cdot x(\overline{v}) \to D, S' \cdot \{\overline{t}\}^m \rangle^l} \text{ (R-Call)}$$

$$\frac{\langle D, S \cdot \overline{t} \to D', S' \cdot v, \emptyset \rangle^l \qquad S'' = \text{drop}(S', m)}{\langle D, S \cdot \{\overline{t}\}^m \to D, S'' \cdot v \rangle^l} \text{ (R-Block)}$$

Figure C.1: FM reduction rules

Appendix D

Implementation Figures

```
pub enum Token {
    NumericLiteral(i64),
    Identifier(String),
    Box,
    Ref,
    Deref,
    Let,
    Mut,
    Assign,
    Fn,
    // ...
}
```

Figure D.1: Token implementation

```
pub struct Lexer {
    input: Vec<String>,
    current_position: usize,
}

impl Lexer {
    pub fn new(input: &str) -> Lexer { ... }

    fn next_token(&mut self) -> TokenKind{
        // if token is valid, return token, else return error
        match self.input.get(self.current_position) {
            Some(token) => {
                match token.as_str() {
                    "box" => TokenKind::Box,
                    "ref" => TokenKind::Ref,
                    "let" => TokenKind::Let,
                    "mut" => TokenKind::Mut,
                    //..
                }
            },
            None => TokenKind::EOF
        }
    }

}
```

Figure D.2: Lexer implementation

```rust
pub trait AST : Evaluate + TypeCheck {}

pub struct Reference {
    pub location: String,
    pub path: Vec<usize>,
    pub owned: bool, // owned references are dropped recursively
}

pub enum Value {
    NumericLiteral(i64),
    Reference(Reference),
    Epsilon,
    Undefined
}

 pub enum LVal {
    Variable { name: String, copyable: Option<bool> },
    Deref { var: Box<LVal> },
}

pub enum Term {
    Variable(LVal),
    Value(Value),
    Box { term: Box<Term> },
    Ref { mutable: bool, var: LVal },
    Let { mutable: bool, variable: LVal, term: Box<Term> },
    // ...
}
```

Figure D.3: Abstract Syntax Tree (AST) implementation

```rust
pub struct Parser
{
    tokens: Vec<TokenKind>,
    current_position: usize,
}

impl Parser {

    fn check_consume(&mut self, token: TokenKind)  { ...}

    fn parse_type(&mut self) -> AtomicType { ... }

    fn parse_variable(&mut self) -> Variable { ... }

    // ... other parsing functions

    fn parse_term(&mut self) -> Term {
        match self.tokens.get(self.current_position) {
            Some(token) => {
                match token {
                    TokenKind::NumericLiteral(n) => {
                        self.current_position += 1;
                        Term::Value(Value::NumericLiteral(*n))
                    },
                    TokenKind::Identifier(s) => { ... },
                    TokenKind::Box => { ... },
                    TokenKind::Mut => { ... },
                    TokenKind::Ref => { ... },
                    TokenKind::Let => { ... },
                    _ => panic!("Invalid token: {:?}", token)
                }
            },
            None => panic!("Unexpected EOF")
        }
    }
}
```

Figure D.4: Parser implementation

```rust
pub struct Slot<T> {
    pub value: T,
    pub lifetime: usize,
}

pub enum Type {
    Epsilon,
    Numeric,
    Reference { var: LVal, mutable: bool },
    Box(Box<Type>),
    Undefined(Box<Type>),
    Function { args: Vec<Type>, ret: Option<Box<Type>>}
}

impl Type {
    pub fn copyable(&self) -> bool {}
    fn prohibits_reading(&self, variable: LVal) -> bool {}
    fn prohibits_writing(&self, variable: LVal) -> bool {}
    // ...
}
```

Figure D.5: Type Implementation

```rust
pub struct Typeenvironment {
    gamma: HashMap<Variable, Slot<Type>>,
}

impl Typeenvironment {
    pub fn new() -> Typeenvironment { ... }
    pub fn get(&self, key: &Variable) -> Result<Slot<Type>, String> { ... }
    pub fn insert(&mut self, key: Variable, value: Type, lifetime: Lifetime) { ... }
    // ...
}

pub fn write_prohibited(gamma: &Typeenvironment, variable: LVal) -> bool { ... }
pub fn read_prohibited(gamma: &Typeenvironment, variable: LVal) -> bool { ... }
pub fn move_var(
    mut gamma: Typeenvironment,
    variable: LVal,
    lifetime: Lifetime
) -> Result<Typeenvironment, String> { ... }

// ... other helper functions
```

Figure D.6: Type Environment Implementation

```rust
pub trait TypeCheck {
    fn type_check(
        &mut self,
        gamma: Typeenvironment,
        lifetime: usize,
    ) -> Result<(Typeenvironment, Type), String>;
}

impl TypeCheck for Value { ... }

impl TypeCheck for LVal { ...}

impl TypeCheck for Program { ... }

impl TypeCheck for Term {
    fn type_check(
        &mut self,
        gamma: Typeenvironment,
        lifetime: usize,
    ) -> Result<(Typeenvironment, Type), String> {
        match self {
            Term::FunctionCall { name, params } => { ... }
            Term::FunctionDeclaration {
                name: fn_name,
                args,
                body,
                ty,
            } => { ... }
            Term::Variable(ref mut var) => { ... }
            Term::Value(val) => { ... }
            Term::Box { term } => {
                let (g, t) = term.type_check(gamma, lifetime)?;
                return Ok((g, Type::Box(Box::new(t))));
            }
            Term::Ref { mutable, var } => { ... }
            Term::Let { variable, term, .. } => { ... }
            Term::Assign { variable, term } => { ... }
        }
    }
}
```

Figure D.7: Type Checking Implementation

```rust
pub struct StackFrame {
    pub locations: HashMap<String, Reference>,
    pub functions: HashMap<String, (Vec<Argument>, Vec<Term>)>,
}

pub struct Store {
    pub cells: HashMap<Location, Slot<Value>>,
}

impl Store {
    pub fn new() -> Store { ... }
    pub fn allocate(&mut self, value: Value, lifetime: usize) -> Reference { ... }
    pub fn read(&self, reference: Reference) -> Result<Value, String> { ... }
    pub fn drop_lifetime(&mut self, lifetime: usize) { ... }
    pub fn drop(&mut self, value: &Value) -> Result<(), String> { ... }
    pub fn get(&self, reference: Reference) -> Option<&Slot<Value>> { ... }
    // ...
}

pub struct State {
    pub stack: Vec<StackFrame>,
    pub store: Store,
}

impl State {
    pub fn new(stack: Vec<StackFrame>, store: Store) -> State { ... }
    pub fn locate(&self, name: String) -> Result<Reference, String> { ... }
    pub fn dom(&self) -> Vec<String> { ... }
    pub fn top(&self) -> &StackFrame { ... }
    pub fn top_mut(&mut self) -> &mut StackFrame { ... }
    // ...
}
```

Figure D.8: State Implementation

```rust
pub trait Evaluate {
    fn evaluate(
        &mut self,
        s: State,
        lifetime: usize
    ) -> Result<(State, Self), String>
    where
        Self: Sized;
}

impl Evaluate for Value { ... }

impl Evaluate for Variable { ... }

impl Evaluate for Program { ... }

impl Evaluate for Term {
    fn evaluate(
        &mut self,
        s: State,
        lifetime: usize
    ) -> Result<(State, Term), String> {
        match self {
            Term::Let { variable, term, .. } => { ... }
            Term::Assign { variable, term } => { ... }
            Term::Box { term } => { ... }
            Term::Ref { mutable: _, var } => { ... }
            // ...
        }
    }
}
```

Figure D.9: Execution Implementation

```rust
pub struct Interpreter {
    pub program_state: State,
    typing_environment: Typeenvironment,
}

impl Interpreter {
    pub fn new() -> Interpreter { ... }

    pub fn run(&mut self, mut ast: Program) -> Result<Value, String> {

        // initial type check
        ast.type_check(self.typing_environment.clone(), 0)?;

        for mut term in ast.terms {
            assert_progess(
                self.program_state.clone(),
                term.clone(),
                self.typing_environment.clone(),
                0,
            )?;

            assert_preservation(
                self.program_state.clone(),
                term.clone(),
                self.typing_environment.clone(),
                0,
            )?;

            let (s, _) = term.evaluate(self.program_state.clone(), 0)?;
            let (gamma2, _) = term.type_check(self.typing_environment.clone(), 0)?;

            self.typing_environment = gamma2;
            self.program_state = s;
        }

        return Ok(Value::Epsilon);
    }
}
```

Figure D.10: Interpreter Implementation