# TOWARDS A COMPLETE FORMAL SEMANTICS OF RUST

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Alexa White

March 2021

COMMITTEE MEMBERSHIP

|  |  |
|---|---|
| TITLE: | Towards a Complete Formal Semantics of Rust |
| AUTHOR: | Alexa White |
| DATE SUBMITTED: | March 2021 |
| COMMITTEE CHAIR: | Aaron Keen, Ph.D.<br>Professor of Computer Science |
| COMMITTEE MEMBER: | Maria Pantoja, Ph.D.<br>Professor of Computer Science |
| COMMITTEE MEMBER: | Theresa Migler, Ph.D.<br>Professor of Computer Science |

ABSTRACT

Towards a Complete Formal Semantics of Rust

Alexa White

Rust is a relatively new programming language with a unique memory model designed to provide the ease of use of a high-level language as well as the power and control of a low-level language while preserving memory safety. In order to prove the safety and correctness of Rust and to provide analysis tools for its use cases, it is necessary to construct a formal semantics of the language. Existing efforts to construct such a semantic model are limited in their scope and none to date have successfully captured the complete functionality of the language. This thesis focuses on the K-Rust implementation [9], which is implemented in a rewrite-based semantic framework called K, and expands it to include a larger subset of the Rust language. The K framework allows Rust programs to be executed by the defined semantic model, and the implementation is tested with several Rust programs by comparing the results of execution to the Rust compiler itself.

## ACKNOWLEDGMENTS

Thanks to:

- My advisor, Dr. Aaron Keen, and my committee members Dr. Maria Pantoja and Dr. Theresa Migler, for their time and assistance
- Rasmus the cat, for providing emotional support
- Andrew Guenther, for uploading this template

TABLE OF CONTENTS

Page

LIST OF TABLES . . . . . viii

LIST OF FIGURES . . . . . ix

CHAPTER

1 Introduction . . . . . 1

2 Background . . . . . 4

2.1 The Rust Programming Language . . . . . 4

2.1.1 Ownership . . . . . 4

2.1.2 Borrowing . . . . . 6

2.1.3 Mutable Variables and References . . . . . 6

2.1.4 Rules of Borrowing . . . . . 7

2.1.5 Lifetimes . . . . . 8

2.2 Semantic Formalization . . . . . 10

2.3 The K-Framework . . . . . 10

2.3.1 Configurations . . . . . 11

2.3.2 Computations . . . . . 11

2.3.3 Syntax and Rules . . . . . 12

3 Related Work . . . . . 15

3.1 Patina . . . . . 15

3.2 Oxide . . . . . 15

3.3 RustBelt . . . . . 16

3.4 KRust . . . . . 16

3.5 K-Rust . . . . . 17

4 K-Rust . . . . . 18

4.1 Operational Semantics . . . . . 18

4.2 Type System . . . . . 23

4.2.1 Configuration . . . . . 24

4.2.2 Rules . . . . . 25

4.2.2.1 Lifetime TC . . . . . 26

4.2.2.2 Parameter TC . . . . . . . . . . . . . . . . . . . . . . 27
4.2.2.3 Expressions TC . . . . . . . . . . . . . . . . . . . . . . 27
5 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.1 The Match Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.2 Type Checking Semantics . . . . . . . . . . . . . . . . . . . . . . . 34
5.2.1 Matching Literals . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.2.2 Matching Named Variables . . . . . . . . . . . . . . . . . . . . . 38
5.3 Operational Semantics . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.4 Type Checking and Evaluating a Match Expression . . . . . . . . . . 40
6 Testing and Validation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.1 Translating Rust Programs . . . . . . . . . . . . . . . . . . . . . . . 45
6.1.1 Rust to Surface-Rust . . . . . . . . . . . . . . . . . . . . . . . . 45
6.1.2 Surface-Rust to Core-Language . . . . . . . . . . . . . . . . . . . 46
6.2 Testing Benchmarks and Example Test Cases . . . . . . . . . . . . . 47
6.2.1 Match Test Example . . . . . . . . . . . . . . . . . . . . . . . . . 48
6.2.2 Compiler Rejection Example . . . . . . . . . . . . . . . . . . . . . 50
7 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
7.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
7.2 Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
BIBLIOGRAPHY . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

# LIST OF TABLES

Table Page

4.1 Compound Type Definitions . . . . . . . . . . . . . . . . . . . . . . . . 23

6.1 Translating Rust to Surface-Rust . . . . . . . . . . . . . . . . . . . . . 46

6.2 Translating Surface-Rust to Core-Language . . . . . . . . . . . . . . 47

# LIST OF FIGURES

Figure Page

2.1 Assignment Move Example . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Function Move Example . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 Function Borrow Example . . . . . . . . . . . . . . . . . . . . . . 7
2.4 Mutable Borrow Example . . . . . . . . . . . . . . . . . . . . . . 7
2.5 Lifetime Example . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.6 Lifetime Example Corrected . . . . . . . . . . . . . . . . . . . . . 9
2.7 K Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.8 LAMBDA Language . . . . . . . . . . . . . . . . . . . . . . . . . 13

4.1 Operational Semantics Configuration . . . . . . . . . . . . . . . . . 19
4.2 Rewrite Rules for Non-Atomic Read . . . . . . . . . . . . . . . . . 20
4.3 Core-language Example . . . . . . . . . . . . . . . . . . . . . . . 22
4.4 Type System Configuration . . . . . . . . . . . . . . . . . . . . . 25
4.5 Type Checking Architecture [9] . . . . . . . . . . . . . . . . . . . 26
4.6 Rewrite Rules for Lifetimes . . . . . . . . . . . . . . . . . . . . . 27
4.7 Rewrite Rules Parameter Binding . . . . . . . . . . . . . . . . . . 28
4.8 Rewrite Rules for Function Decomposition and Return TC . . . . . 28
4.9 Rewrite Rule for Function Call . . . . . . . . . . . . . . . . . . . 29
4.10 Rewrite Rules for Branch Decomposition . . . . . . . . . . . . . . 30
4.11 Rewrite Rules for Assignment Decomposition . . . . . . . . . . . . 31

5.1 Matching Literals Example . . . . . . . . . . . . . . . . . . . . . 33
5.2 Matching Named Variables Example . . . . . . . . . . . . . . . . . 33
5.3 Type Checking Syntax for Match Expressions . . . . . . . . . . . . 34
5.4 Match Branch Type Checking Syntax . . . . . . . . . . . . . . . . 35
5.5 Rewrite Rules for Match Type Checking . . . . . . . . . . . . . . . 36
5.6 Operational Syntax for Match Expressions . . . . . . . . . . . . . . 39
5.7 Rewrite Rules for Match Operations . . . . . . . . . . . . . . . . . 40
5.8 Match Example Rust Code . . . . . . . . . . . . . . . . . . . . . 40

5.9 Match Example Surface-Rust Code . . . . . . . . . . . . . . . . . . . 41
5.10 Match Example Core-Language Code . . . . . . . . . . . . . . . . . . 43

6.1 Match Test Example Rust Program . . . . . . . . . . . . . . . . . . . 48
6.2 Match Test Example Surface-Rust Program . . . . . . . . . . . . . . 49
6.3 Match Test Example Core-Language Program . . . . . . . . . . . . . 50
6.4 Compiler Rejection Example Rust Program . . . . . . . . . . . . . . 50
6.5 Compiler Rejection Example Surface-Rust Program . . . . . . . . . . 51
6.6 Compiler Rejection Example Results . . . . . . . . . . . . . . . . . . 52

# Chapter 1

# INTRODUCTION

Building a formal semantics of a programming language is a critical prerequisite to proving the safety, correctness, and reliability of programming tools implemented in that language. Semantic models serve as the basis for both analyzing and verifying the behavior of a given program as well as for bringing into focus the subtleties and ambiguities of the language itself. Rust, a relatively recently developed language with a syntax similar to C or C++, was designed with the intention of preserving memory safety and performance while still providing the power and control of a low-level language. One of the main features of Rust is its enforcement of ownership and borrowing of values, which allows for low overhead memory management without the need for a garbage collector. The language has seen wide success due to its unique ability to bridge the gap between low-level systems programming and high-level application programming. However, Rust's core principles of ownership and borrowing have not yet been thoroughly formalized which will be required to prove its safety guarantees and functional correctness.

Rust's claims of providing memory safety while preserving performance have significant implications for the field of computer science, but they have yet to be formally proven, and it is possible that not all of them hold. In order to verify the safety of the language and to construct tools for formal analysis of programs written in Rust, it is first necessary to construct a formalization of the semantics of the language. A set of formal semantics for a programming language is a way of modeling the computational meaning of a program written in that language with a precisely defined notation. This provides an abstract representation of the language that can be used to reason about specific use cases and ensure the correctness of the compiler.

In the few years since the release of Rust, multiple efforts have been made to formalize its semantics and prove the safety and correctness of the language. The first of these efforts, Patina [14], attempted to formalize of subset of an earlier version of Rust and provided some syntactic proofs for these semantics, but was made somewhat obsolete with Rust's first stable release, which drifted from Patina's language model. RustBelt [8], considered to be one of the best efforts so far to formalize the Rust language, focuses on a subset of the language and is then extended to include the libraries that use unsafe code. The paper reports the verification of what they consider the most important Rust libraries and leaves the extension to other libraries as an area of future work. Another implementation, Oxide [18], attempts to distill the complex language to make it easier to understand the semantic rules and provides a foundation for future research on the semantics of Rust. Oxide deals only with an abstract notion of memory and focuses on the safe portion of Rust, ignoring standard library abstractions implemented using unsafe code. Two implementations, KRust [17] and K-Rust [9], both make use of the K framework to define their semantics. K is a rewrite-based semantic framework that can be compiled into an executable interpreter and used to conduct formal analysis on the defined language. These works differ from previous efforts in that the use of K allows them to present concrete executables for their languages, as opposed to the formal proofs given in other works.

The main limitation of these and other similar efforts to formalize the semantics of the Rust language is that they are either limited to high-level abstractions of Rust or focus on only its core features with other features left to be explored in future work. In order to formalize the complete language, it is necessary to extend these implementations until their semantics encompass the entirety of Rust. In particular, extending and further validating the K-Rust implementation, which is publicly available, is the primary focus of this thesis.

The major contributions of this thesis are:

- Reworking the K-Rust implementation so that it is compatible with a newer version of the K framework
- Extending the functionality of the K-Rust implementation to include pattern matching semantics
- Testing and validating the functionality of the extended K-Rust implementation against the Rust execution environment

The rest of this thesis is structured as follows: Chapter 2 provides background information about the Rust programming language and the K framework. Chapter 3 discusses related works in more detail. Chapter 4 describes the K-Rust implementation in more detail and Chapter 5 describes my contributions to it. Chapter 6 discusses testing and validation of this implementation. Chapter 7 discusses possibilities for future work, and Chapters 8 concludes.

# Chapter 2

# BACKGROUND

## 2.1 The Rust Programming Language

Rust, originally released in 2015, was created with the fundamental goal of empowering users to more confidently program in a wider range of applications[10]. Low-level systems programming in languages like C and C++ traditionally require great caution and discipline to avoid the pitfalls common to applications like memory-management and concurrency. Rust claims to have the ease of use of a high-level programming language while at the same time providing the user with control of low-level details free of these pitfalls. In this way Rust challenges the status quo of programming language design in hopes to provide both speed and safety as well as ergonomics and productivity without making the usual trade-offs.

The feature unique to Rust which makes its safety guarantees possible is its system of ownership and borrowing. This strategy for memory-management does not require the programmer to explicitly allocate and free memory and also eliminates the need for a garbage collector, thereby preserving both speed and safety of the program. The following sections further detail the mechanics of this system and provide examples of its use.

### 2.1.1 Ownership

In the ownership system, variables are responsible for freeing their own resources. To ensure that each resource is freed exactly once, Rust enforces that every resource must have exactly one owner. This guarantees that no variable attempts to free a previously freed resource, and that no resources are left dangling with no owner to free them. Rust outlines the following three basic rules for the ownership system, which are checked at compile time [10]:

1. Each value has a variable referred to as the value's *owner*.

2. Each value can have only one owner at a time.

3. When the owner goes out of scope, the value is dropped.

While these rules are straightforward, they can make programming extremely inconvenient, as resources can be accessed by only one variable. To address this problem, Rust provides the ability to transfer ownership of a value from one variable to another. One way of doing this is with a *move*. Consider the code in Figure 2.1, which moves ownership with an assignment. On line 1, the variable x allocates memory for a new String, and becomes the owner of that allocation. On line 2, the String is moved from x to the new variable y with an assignment. When ownership is moved, y becomes the sole owner of the String and x can no longer access it. On line 3, the value of x is printed. Since x no longer owns any value, this code will result in a compiler error.

```
1  let x = String::from("Hello World");
2  let y = x;
3  println!("{}", x);
```

**Figure 2.1: Assignment Move Example**

Ownership transfer also occurs when a function is called. When a variable is passed as an argument to a function, the ownership is moved to the function parameter. Returning a value from a function will also move its ownership. Consider the code in Figure 2.2. As before, the variable x is the initial owner of the String value. When the function is called on line 5, the ownership of the string is moved to the variable s in the function, and x can no longer access it. The variable x no longer owns any value, so line 6 will once again result in a compiler error. If line 6 is removed, the code will compile and will print the value of the moved String (“Hello World”) from inside the function.

```
1  fn function(s: String) {
2      println!("{}", s);
3  }
4  let x = String::from("Hello World");
5  function(x);
6  println!("{}", x);
```

**Figure 2.2: Function Move Example**

### 2.1.2 Borrowing

Always moving values can be impractical, especially in the case of function calls as seen in the above examples. To avoid this, Rust also allows *borrowing* of values using references. Syntactically, the & symbol before a variable is used to reference the variable’s value, and the * symbol is used for dereferencing. A borrowed value can be temporarily used by another variable without moving ownership of the value, but instead moving a reference to the value. Since the variable does not own the value, it will not be de-allocated when the reference goes out of scope.

Consider the example code in Figure 2.3. This is the same code as in Figure 2.2, but now we make the function call with a reference to the variable x, and change the function parameter to be a reference to a String. Since the String is borrowed and not moved, the variable x keeps ownership of the value and the code will compile. When the code is run, the String will be printed once from inside the function and again on line 6.

### 2.1.3 Mutable Variables and References

In order to change the value a variable points to, the programmer must specify that the variable is mutable, as all variables are considered immutable by default. To

```
fn function(s: &String) {
    println!("{}", s);
}
let x = String::from("Hello World");
function(&x);
println!("{}", x);
```

**Figure 2.3: Function Borrow Example**

create a mutable variable, the keyword **mut** is used before the variable name when it is declared. A mutable reference can also be created by using **&mut** before the variable name.

Consider the example code in Figure 2.4. In this example, the function attempts to modify the value of the String. To allow this, the variable x, which owns the String, is declared as mutable on line 4. On line 6, we pass a mutable reference of the String to the function, which now takes a mutable reference to a String as an argument. The reference to the String is mutated on line 2, and then printed on line 6 with the owner, variable x. This code will compile and print the expected "Hello World".

```
1 fn function(s: &mut String) {
2     s.push_str(" World");
3 }
4 let mut x = String::from("Hello");
5 function(&mut x);
6 println!("{}", x);
```

**Figure 2.4: Mutable Borrow Example**

### 2.1.4 Rules of Borrowing

Allowing references is a major convenience for programming, but it introduces some issues into the ownership system, such as possible data races or dangling references. For this reason Rust places the following restrictions on references, which are checked at compile time [10]:

1. At any given time, only one of the following conditions can be true:

   - a value has one mutable reference
   - a value has any number of immutable references

2. The scope of a reference cannot outlive the scope of the original owner

### 2.1.5 Lifetimes

The previous sections refer to variables and references going "out of scope" to determine when a resource should be de-allocated to prevent dangling references. This section will provide a brief overview of how Rust determines that scope with the concept of *lifetimes*. Similar to types, lifetimes are often implicit and must be inferred by the compiler, so in cases of ambiguity they must be properly annotated by the programmer to ensure that references used at runtime are valid. The Rust compiler uses a *borrow checker* to compare lifetimes and check that the rules for borrowing are maintained. Syntactically, lifetimes are annotated with a ' symbol followed by an arbitrary name for the lifetime. These annotations are used to specify the relationships between multiple lifetimes.

Consider the code in Figure 2.5. In this example, two strings are created, then passed to a function which returns the longer of the two strings. The resulting string is then printed. This code will result in a compiler error due to the return type of the function longest_string. The function returns a reference, but the function signature does not indicate where the reference was borrowed from. Because of this, the lifetime of the output of the function is not clear to the borrow checker, so the compiler will reject the program.

To fix this code, we must add lifetime annotations to the function signature. Note that lifetime annotations do not change the actual length of any lifetimes, but rather

```
fn longest_string(s1: &str, s2: &str) -> &str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}
let x = String::from("a string");
let y = String::from("a longer string");
let result = longest_string(x.as_str(), y.as_str());
println!("{}", result);
```

**Figure 2.5: Lifetime Example**

inform the compiler how ambiguous lifetimes are related to each other. In this case we must tell the compiler how the lifetime of the output of the function relates to the lifetimes of its input. The annotations are shown in Figure 2.6. The lifetime annotation **‘a** is used to tell the compiler that the function takes two inputs, each with a lifetime at least as long as **‘a**, and has an output with a lifetime at least as long as **‘a**. At runtime, the lifetime substituted for **‘a** will be the overlap of the lifetimes of s1 and s2, and since lifetimes are nested, this will always be the smaller of the two lifetimes. This code compiles and prints the longer of the two strings, as expected.

```
fn longest_string<‘a>(s1: &‘a str, s2: &‘a str) -> &‘a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}
let x = String::from("a string");
let y = String::from("a longer string");
let result = longest_string(x.as_str(), y.as_str());
println!("{}", result);
```

**Figure 2.6: Lifetime Example Corrected**

As previously mentioned, the Rust compiler is sometimes able to infer lifetimes, so annotation is not always necessary. Excluding explicit annotation of lifetimes is referred to as *lifetime elision*. The compiler will follow a defined set of rules to infer a lifetime, and will give an error if any references remain for which the lifetime could not be determined. Determining the lifetime of all references in a program and verifying that they adhere to the borrowing rules theoretically allows Rust to guarantee the memory safety of the program at compile time.

## 2.2 Semantic Formalization

A *semantics* for a programming language is a model for the computational meaning of a program written in that language [12]. Semantics are used as an abstraction of the real execution of a program, which would be too complex to describe directly. They instead represent the most relevant aspects of possible executions and ignore details not pertaining to the correctness of the program. Correctness is typically defined by the input and output of the program and whether or not it terminates. Semantics are called *formal* when they are written in a notation with a precisely defined meaning. Two different levels of semantics are *static semantics* and *dynamic semantics*. Static semantics include only the checks that can be performed at compile-time before actually running a program, such as type checking and borrow checking in the case of Rust. Dynamic semantics are a model of the run-time behavior of a program and include the observable behaviors of a program when it is executed.

## 2.3 The K-Framework

K [16] is a rewrite-based semantic framework that can be used to define the semantics of a programming language that can then be compiled into an executable interpreter for the defined language. K provides the user with several tools for conducting formal analysis of the defined language, including the ability to execute programs with the compiled interpreter. A programming language is defined in K with three main components: configurations, computations, and rules. These components are further detailed in the following sections. K has been used to formalize a number of large programming languages, including C [5], Java [3], JavaScript [13], and PHP [7].

### 2.3.1 Configurations

Configurations represent the program state with a set of labeled *cells*. Cells are defined by the user to capture the necessary components of the language being defined. They can be nested, and can contain lists, sets, maps, and multisets. Some examples of cells in a configuration are an environment mapping variables to their memory addresses, a store mapping addresses to values, a stack for function calls, or a multiset of concurrent threads.

Figure 2.7 shows the syntax for declaring a configuration. This is the configuration for a simple language defined in the K tutorial [2]. The entire configuration is typically nested in a top-level cell labeled *T*. Line 3 defines the k cell, which initially contains the contents of the program from the input file. This is accomplished with the $PGM keyword. It is common practice to label the cell holding the contents of the program *k*. Line 4 defines an environment cell labeled *env*, which is initialized to an empty map. Similarly, line 5 defines the *store* cell which contains a map.

```
1  configuration
2  <T>
3      <k> $PGM:Exp </k>
4      <env> .Map </env>
5      <store> .Map </store>
6  </T>
```

**Figure 2.7: K Configuration**

### 2.3.2 Computations

Computations in K are the list of computational tasks to be performed in the program being executed, where each task is a term over the defined syntax of the language, and are typically stored in the k cell of the configuration. Syntactically, these tasks are separated by the ~> ("followed by") operator, and **.** denotes the empty computation. The initial state of the computations will be the entire text of the input program. Computations are performed by rewriting the terms using the defined rewrite rules for the language. The execution of a program is finished when no more rewrite rules can be applied to any of the terms in the computations.

### 2.3.3 Syntax and Rules

The syntax of the language is defined with the keyword **syntax** followed by the name of the syntax object and a list of terms that are considered that type of object. Rules are the defined rewrite instructions that match to terms in computations. They are defined with the keyword **rule** and use the rewrite symbol =>. Rules manipulate the contents of the configuration cells to change the state of the program.

Consider the example in Figure 2.8 (from the K tutorial), which defines a very simple language called LAMBDA. This subset of the language defines the syntax objects Val and Exp. A Val defined on lines 4-5, can be either a variable (using K's builtin KVar syntax), or a lambda expression (written lambda x . y, where x is the variable and y is the expression). An Exp, defined on lines 6-7, can be a Val, or one Exp followed by another. On line 7, the attributes *strict* and *left* are used to specify the evaluation strategy for the term. Strict specifies that both terms must be evaluated to a KResult object before rewrite rules can be applied. Here a KResult is defined to be a Val. The left attribute specifies that the term should be left associative. The *binder* attribute on line 5 invokes K's builtin substitution module, which handles how variables are bound to the lambda expressions. Several other attributes exist in the framework for specifying the evaluation strategies that should be used for a given term.

On line 10, the rewrite rule is defined. This rule rewrites a lambda expression followed by a Val in the k cell by binding the Val V to the KVar X in the Exp E (written E[V/X]). The syntax category (also called *sort*) can be specified as a tag on the terms. For example, here X must be of sort KVar, E must be of sort Exp, and V must of sort Val. The **...** at the end of the rule specifies that the rest of the contents in the k cell should be ignored and not modified, and only the matching term is rewritten.

```
1  module LAMBDA
2    imports KVAR-SYNTAX
3
4    syntax Val ::= KVar
5                 | "lambda" KVar "." Exp      [binder]
6    syntax Exp ::= Val
7                 | Exp Exp                    [strict, left]
8    syntax KResult ::= Val
9
10   rule <k> (lambda X:KVar . E:Exp) V:Val => E[V / X] ... </k>
11
12 endmodule
```

**Figure 2.8: LAMBDA Language**

Consider the following sample code that could be executed by this semantics:

```
    (lambda x . x) (lambda y . (y z) ) a
```

The configuration for this language contains only the k cell. Initially, the computations in the k cell will be the entire line of code, along with ~> . ("followed by the empty computation") at the end to signify the end of the program. Because the computation is initially in the form Exp Exp, (where the first Exp is (lambda x . x) (lambda y . (y z) ) and the second is a), the rewrite rule cannot yet apply. This is because rules do not just apply anywhere they match, but only where they have been given permission to apply by means of evaluation strategies specified for language constructs. In this case, the strict attribute allows the arguments to be evaluated before the semantic rules are applied. Both Exp will be evaluated until they reach a KResult (in this language a Val) or until they cannot be evaluated further. The first Exp is also of the form Exp Exp, where its first Exp is the Val (lambda x . x) and its second is the Val (lambda y . (y z)). Since this construct is left associative, the configuration will now look like this (note that no rewrite rules have been applied yet, but the evaluation strategies have changed K's internal structure for storing the terms from (Exp Exp) to ((Val Val) Exp)):

```
<k>
    ( (lambda x . x ) (lambda y . (y z) ) ) a ~>  .
</k>
```

The rewrite rule can now be applied, since we have a term containing a lambda expression followed by a Val. Here X is the KVar x, E is the Exp x, and V is Val (lambda y . (y z) ). Applying the rule results in the following configuration:

```
<k>
    (lambda y . (y z) ) a ~>  .
</k>
```

Since the first Exp has now reached a KResult, the second Exp can be evaluated. Since the Exp a is already of sort Val, the rewrite rule can be applied again, where X is the KVar y, E is the Exp (y z), and V is KVar a. After the second rewrite the remaining computation in the k cell does not match any of the defined rules, so execution stops. The final output will be the resulting configuration, as expected:

```
<k>
    a z ~>  .
</k>
```

Chapter 3

# RELATED WORK

## 3.1 Patina

Patina [14] is considered to be the first major research effort to semantically formalize Rust. Patina formalizes the Rust type system to capture the core features of memory safety and provides partial syntactic proofs. This work focused on an earlier version of the Rust language which at the time was not yet stable, and the newer versions released have since drifted from the language model of Patina. The memory layout and validity decisions made in this work also cause problems, as these decisions have not yet been concretely specified by Rust itself. Despite its issues, Patina provided a starting point for following research into formalizing Rust's ownership and borrowing system.

## 3.2 Oxide

Oxide [18] is a work focused on formalizing a language that represents a distilled version of the Rust language. This language, called Oxide, has syntax very similar to Rust but deals only with an abstract notion of memory. This allows Oxide to sidestep the requirement of specific memory layouts for its types, motivated by the fact that Rust does not have a formal specification for memory layout and validity guarantees, especially in regard to its unsafe code base. Oxide also requires fully annotated type bindings in order to make the semantics easier to follow and avoid the complexities that come with a type inference system.

Oxide is focused primarily on formalizing the ownership system and so the language includes only the safe portions of Rust and ignores its libraries implemented with unsafe code. The main contributions of this work are proving the first syntactic type safety result for the Rust language and providing the first inductive definition for Rust's borrow checker. The authors intended for the Oxide language to provide a basis for future work on formalizing Rust by extending the language to include Rust libraries implemented with unsafe code.

## 3.3 RustBelt

In RustBelt [8], the authors claim to give the first machine-checked formal proof of safety for a language that represents a core subset of the Rust language they call $\lambda_{Rust}$. Like Oxide, $\lambda_{Rust}$ focuses on the subset of the Rust language containing its safe libraries. Unlike Oxide, however, the work provides a method for including unsafe Rust libraries into the proof by describing the verification conditions that the unsafe features must satisfy to be considered a safe extension of the language. This verification is carried out on some important libraries commonly used in Rust, such as Cell, RefCell, and Mutex. The ability of $\lambda_{Rust}$ to support unsafe libraries in a way proven to be safe within the language makes this work widely considered to be one of the most significant efforts so far to fully formalize Rust.

## 3.4 KRust

By utilizing the K framework, KRust [17] presents a formal operational semantics on a subset of Rust. The distinguishing feature of this work is that the use of K to define the language yields a formal interpreter and builtin verification tools that can be used to execute and test real Rust programs. This work claims to present the first formal executable semantics for Rust. The semantics defined with KRust were thoroughly tested on hundreds of programs, including the official Rust test suite. Executable semantics in K also have the advantage of providing formal analysis tools for additional research, such as state-space explorers, which can be used in applications like debuggers and verification of functional correctness. One of the main limitations of this work is that, similar to other related works, it does not cover any of Rust's unsafe libraries. It also leaves out some significant features of safe Rust, including the type system and lifetime annotations. These and other extensions to the KRust language are left for future work.

## 3.5 K-Rust

Not to be confused with KRust, K-Rust [9] is another executable formal semantics for Rust implemented in the K framework. K-Rust encompasses all of Rust's safe libraries and, unlike KRust, includes the totality of Rust's type system. This implementation is separated into operational semantics, which can be thought of as the run-time execution of a Rust program, and a type checking semantics, which can be thought of as the compile-time type and borrow checking for a program. The K configuration also includes a memory model with memory operations. This implementation was tested with several Rust programs by comparing the results with those of the Rust execution environment. Similar to its contemporaries, K-Rust is an incomplete subset of the Rust language and leaves the extension of the semantics for future work.

As it constitutes the basis for this work, the implementation details of K-Rust will be described in more detail in the following chapter.

# Chapter 4

# K-RUST

The K-Rust implementation is made up of two separate components: a type checking semantics that models Rust's ownership system, and an operational semantics for a core subset of the Rust language referred to as *core-language*. The operational semantics consists of the *memory level*, which contains a memory model together with memory operations, and the *core-language level*, which contains the operational semantic rules for the core-language based on the memory model. The type checking semantics is referred to as the *surface-rust level*. To run a Rust program with the K-Rust implementation, it must first be translated into surface-rust. If it is correct on the surface-rust level type system semantics, it is translated into core-language and run on the operational semantics. Core-language is a purely functional language with syntax that captures the behavior of Rust's safe constructors. Surface-rust is an extension of core-language which is syntactically similar to Rust, and includes variable modifiers like mutabilities and lifetimes as well as an annotated type system. The entirety of the original K-Rust implementation can be found at [1].

## 4.1 Operational Semantics

This section discusses the operational semantics of K-Rust. For reference, the K configuration for these semantics is shown in Figure 4.1. The three main components of this configuration are threads, closures, and memory.

The memory model of K-Rust is the basis on which the rest of the operational semantics are built. As shown on line 21 of Figure 4.1, the memory cell of the configuration contains a *memaddress* cell, a *blocks* cell, and a *memorystatus* cell.

```
<T>
    <threads>
        <thread multiplicity="*">
            <k> $PGM:Exp </k>
            <env> .Map </env>
            <clstack> .List </clstack>
        </thread>
    </threads>

    <closures>
        <closureCnt> 0:Int </closureCnt>
        <funclosure> .Map </funclosure>
        <closure multiplicity = "*">
            <crId> 0:Int </crId>
            <crContext> .Map </crContext>
            <crParams> .FnParams </crParams>
            <crBody> .K </crBody>
        </closure>
    </closures>

    <memory>
        <memaddress> 0:Int </memaddress>
        <blocks>
            <block multiplicity="*">
                <baddress> .K </baddress>
                <bnum> 0:Int </bnum>
                <bstore> .Map   </bstore>
            </block>
        </blocks>
        <memorystatus> .Map </memorystatus>
    </memory>
</T>
```

**Figure 4.1: Operational Semantics Configuration**

Memaddress is an integer counter for the number of blocks that have been allocated. Memorystatus is a map from memory locations to their statuses. A memory status is an integer pair, where the first integer is the number of operations that write the memory location and the second is the number of operations that read the memory location. Finally, the blocks cell can contain any number of *block* cells, indicated by the "multiplicity="*"" tag on line 24. The block cell represents an allocated block of memory. This includes the address of the block (*baddress*), the number of data items stored in the block (*bnum*), and a block store mapping the indices of the block to the data itself (*bstore*). The memory blocks contain units of primitive data types, and the block structure allows for the creation of compound data types like structs and arrays. The memory statuses are used to keep track of variable ownership.

The memory operational semantics rules include allocating a new block, atomic reads and writes, non-atomic reads and writes, appending a block, freeing a block, and compare and swap. As an example of these operations, Figure 4.2 shows the non-atomic read rewrite rules.

```
rule <k>
        readna(address(N:Int), I:Int) => readnac(address(N),I:Int) ...
     </k>
     <memorystatus>
        ... (N |-> memstatus(_,(K:Int => K+Int1))) ...
     </memorystatus>

rule <k>
        readnac(address(N:Int), I:Int) => V ...
     </k>
     <memorystatus>
        ... (N |-> memstatus(_,(K:Int => K-Int1))) ...
     </memorystatus>
     <baddress> address(N:Int) </baddress>
```

**Figure 4.2: Rewrite Rules for Non-Atomic Read**

The call to a non-atomic read *readna* takes an address and an Int as arguments. An address is a wrapper for an Int, which here represents the memory location being read and is labeled N. The second Int, labeled I, is an offset into this memory location. The first rewrite rule increments the number of operations reading from block N by 1 in the memorystatus cell on line 5. It also rewrites this computation in the k cell on line 2 to a *readnac* with the same address and offset as arguments to finish the read.

The second rule handles the read itself. Similar to the first rule, the memorystatus for reads is now decremented by 1 on line 12 since after this rewrite rule is applied the read will be finished. To read the value, the block with address N is located on line 14. Line 15 then locates the value V associated with the offset I in this block. The computation is then rewritten to the value V in the k cell on line 9, and the read is completed.

The two other components of the operational semantics configuration shown in Figure 4.1 are *threads*, which hold the threads of execution of the program, and *closures*, which handle functions. As with memory blocks, the multiplicity tag on the thread and closure cells allow for any number of each to exist at once. Each *thread* cell contains the k cell which holds the computations of the program, along with the *env* cell mapping variables to their values, and the *clstack* cell, which holds environments in the function call stack and is used to restore the environment on a function call return. The closures cell contains the closures themselves as well as the total number of closures (*closureCnt*), and a map from function names to their respective closure ID number (*functionclosure*). Each *closure* cell has a unique integer ID (*crId*), an environment map to be used in evaluating the function body (*crContext*), the function parameters (*crParams*), and the function body (*crBody*).

The operational semantics of K-Rust include rules for function definitions and calls, arithmetic operations, branching statements, forking, and the memory operations described previously.

As an example of core-language, Figure 4.3 shows a simple Rust program followed by its core-language translation. This program defines the function *sum_from_one*, which takes an i32 n and recursively sums every number from 1 to n. It then calls the function with the variable x = 100. In core-language, a statement "if E then E1 else E2" is rewritten to "case E of {E2, E1}". The reordering of the branch expressions is due to the structure of the case rewrite rules. In the K-Rust implementation, branching case statements can only be used on Int types (where booleans are rewritten to either 0 or 1). Each branch expression correlates to an Int (starting at 0 and incrementing by 1 for each branch) which is matched to the Int in the test expression. For if-then statements, the second branch comes first as it correlates to a false test, which would be rewritten to 0. These case semantic rules are modified in the following chapter to include other types and more freedom to define the case expression values being matched. After running the second program with the operational semantics, the k cell contains the expected result of 5050, matching the output of the Rust program.

```
fn sum_from_one(n: i32) -> i32 {
    if n == 0 {
        return 0
    }
    else {
        return n + sum_from_one(n - 1)
    }
}
let x = 100;
sum_from_one(x)
```

```
fn sum_from_one(n) {
    case n == 0 of {
        n + sum_from_one(n - 1),
        0
    }
};
let x = 100 in (sum_from_one(x))
```

**Figure 4.3: Core-language Example**

## 4.2 Type System

The type system semantics are defined on the surface-rust language. Surface-rust is an extension of core-language which includes variable modifiers for verifying ownership and type-checking correctness of a program. Types are given the syntactic sort *RType*. The RTypes included in the type system are the primitive types *i32* and *bool*, as well as pointer types and compound types. Pointer types can be either an own type or a reference type. An own type, denoted *own(RType)*, represents the owner of a value of type RType. A reference type, denoted *ref(Lifetime, Mutability, RType)* is a reference to a value of type RType, where Lifetime refers to the lifetime of the owner of the value and Mutability indicates whether the reference is mutable or immutable (denoted *mut* and *imm*). Lifetimes are either a wrapped integer indicating the lifetime or a lifetime annotation. Compound types can be function types, sum types, or product types. A function type is denoted *fnTy(Lifetimes; RTypes; RType)*, where the fields represent lifetime variables for the function, a list of parameter types, and the return type. Sum types are equivalent to Rust's enum types, and product types are equivalent to Rust's struct types. They are denoted *sumTy(RTypes)* and *prodTy(RTypes)*, where the RTypes are a list of types for all the fields of the compound type. Table 4.1 shows examples of the compound types and their Rust equivalents.

**Table 4.1: Compound Type Definitions**

| Rust | Surface-Rust |
|---|---|
| fn main() {...} | main :=: fnTy(;;void)<br>fun main() ... |
| fn putX<'a>(x: i32, p: &<'a>mut Point) ->bool {...} | putX :=: fnTy('a; i32, ref('a,mut,own(ty(Point))); bool)<br>fun put(x, p) ... |
| struct Point (i32, i32)<br>or<br>struct Point {x: i32, y: i32} | Point :=: prodTy(i32,i32) |
| enum PointOption {<br>  TwoD(i32, i32),<br>  ThreeD(i32, i32, i32),<br>} | TwoD :=: prodTy(i32, i32)<br>ThreeD :=: prodTy(i32, i32, i32)<br>PointOption :=: sumTy(ty(TwoD), ty(ThreeD)) |

### 4.2.1 Configuration

The configuration for the type system semantics is shown in Figure 4.4. The k cell holds the contents of the input program. The *varCtx* cell holds the total number of variables that currently exist in the *varCnt* cell, and a map from the variables to their type information in the *varInfo* cell. When a variable is created, the integer in the varCnt cell is used to uniquely index it in the varInfo map, allowing variables of the same name to shadow one another without losing their original values. The *env* cell holds a map from variable names to their integer indexes in the varInfo map. The *stackEnv* contains a list of lifetimes treated as a stack. Whenever a new lifetime starts, the current environment is saved to the stackEnv cell and restored when that lifetime ends. The *depGraph* cell holds a map representing the Reference Dependence Graph (RDG), which is used to model the reference relations between variables and is used in borrow checking. The *currentLft* cell holds an integer representing the current lifetime. This number is incremented when a new lifetime starts and decremented when a lifetime ends. The *comtypes* cell stores any number of *comtype* cells. These cells represent the definition of either a sum type or a product type. They contain an integer ID (*ctyId*), an integer representing the kind of compound type, where 0 is a product type and 1 is a sum type (*ctyKind*), a map containing all the field types of the compound type (*ctyElem*), and the total number of fields in the type (*cntElem*). Note that for simplicity, this implementation does not allow for the fields of a struct to have names, and instead maps integer indexes to field types in the ctyElem cell. Finally, the *ctyCnt* cell stores the total number of compound types that have been defined, and is used to set the unique integer ID of a compound type when it is created.

```
configuration
<T>
    <k> $PGM:Rust </k>
    <varCtx>
        <varCnt> 0:Int </varCnt>
        <varInfo> .Map  </varInfo>
    </varCtx>
    <typeCtx> .Map </typeCtx>
    <env> .Map </env>
    <stackEnv> .List </stackEnv>
    <depGraph> .Map </depGraph>
    <currentLft> 0:Int </currentLft>
    <comtypes>
        <comtype multiplicity="*" >
            <ctyId> -1:Int </ctyId>
            <ctyKind> 0:Int </ctyKind>
            <ctyElem> .Map  </ctyElem>
            <cntElem> 0:Int </cntElem>
        </comtype>
    </comtypes>
    <ctyCnt> 0:Int </ctyCnt>
</T>
```

**Figure 4.4: Type System Configuration**

### 4.2.2 Rules

Figure 4.5 shows the process of evaluating a program with the type checking semantics. “TC” is short for Type Checking. The arrows represent how each computational item is decomposed using the rewrite rules. Since all code aside from type declarations must be wrapped in a function, Function TC is the starting point for all type checking decomposition. This is divided into four main parts: Lifetimes TC, Parameters TC, Expressions TC, and Return TC. The following sections detail these four components and provide examples of the primary rewrite rules responsible for the type checking decomposition.

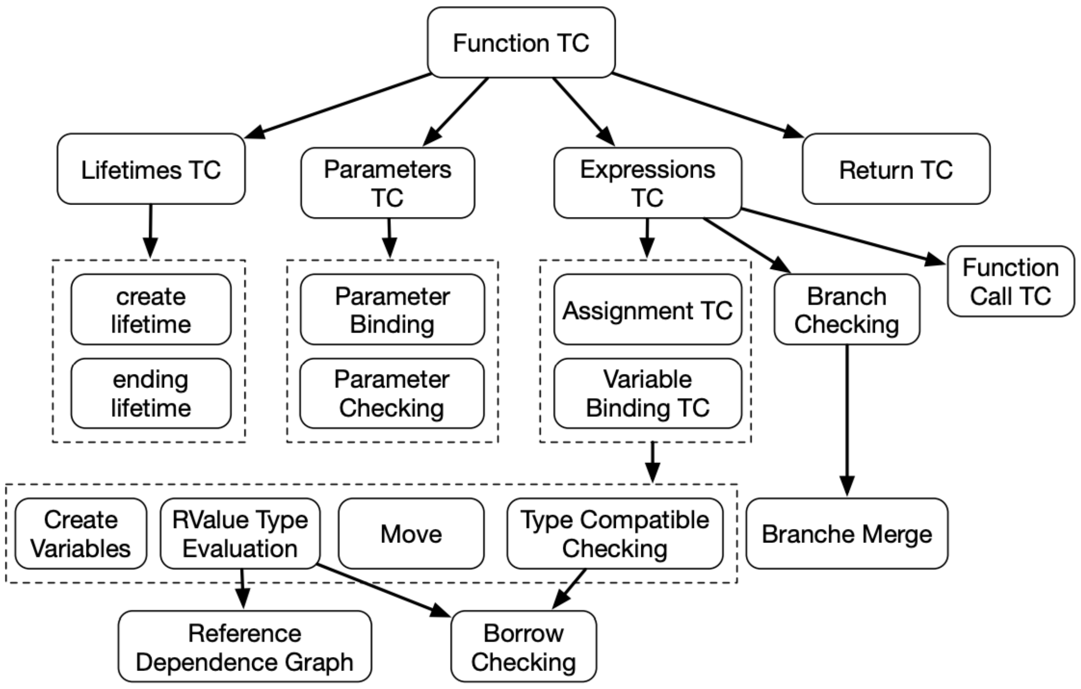

**Figure 4.5: Type Checking Architecture [9]**

#### 4.2.2.1 Lifetime TC

In surface-Rust, the keywords **newlft** and **endlft** are used to begin and end lifetimes. The rules for these terms are shown in Figure 4.6. The rewrite rule for newlft increments the number in the currentLft cell and saves the current environment to the stackEnv cell so it can be restored when this lifetime ends. The rewrite rule for endlft decrements the currentLft number, sets the env cell to the environment on the top of the stack in the stackEnv cell, and removes it from the stack. It then rewrites the endlft computation to a *removeLifetime* computation, which takes as arguments the currentLft before it was decremented and the map stored in the varInfo cell. The rewrite rules for removeLifetime are responsible for removing all the variables that have gone out of scope with the end of the current lifetime. This process involves checking the lifetime information in varInfo for each variable and removing dead variables from the RDG, which is discussed further in the Expression TC section.

```
rule <k> newlft => . ... </k>
     <currentLft> C:Int => C +Int 1 </currentLft>
     <stackEnv> .List => ListItem(Rho) ...  </stackEnv>
     <env> Rho:Map </env>

rule <k> endlft =>  removeLifetime(L,VR,.Map,.Set) ... </k>
     <currentLft> L:Int => L -Int 1 </currentLft>
     <stackEnv> ListItem(Rho) => .List ...  </stackEnv>
     <env> _ => Rho:Map </env>
```

**Figure 4.6: Rewrite Rules for Lifetimes**

#### 4.2.2.2 Parameter TC

The *bindParamTypes* term handles the parameter checking and binding. This set of rules, shown in Figure 4.7, iterates through all the parameters of a function and for each creates a new variable which is bound to the type specified in the fnTy definition. This rule will get stuck if the number of function parameters does not match the number of specified types. To maintain the necessary type information for variables, K-Rust stores them as a 6-tuple in the form $(L, M, T, L_1, L_2, B)$. The first three fields are the static variable information. $L$ is the integer lifetime of the variable, $M$ is the mutability (mut or imm), and $T$ is the type. The second three fields are dynamic information, which may be modified during the type checking process. $L_1$ is the highest lifetime for which this variable has been immutably borrowed, and $L_2$ is the highest lifetime for which this variable has been mutably borrowed. $B$ is a boolean indicating whether or not the variable has been initialized. This 6-tuple is referred to as a variable's *varInfo*, and is what a variable's ID is mapped to in the varInfo cell.

#### 4.2.2.3 Expressions TC

The *rtTyCK* term handles the return type checking. This rule, shown in Figure 4.8 along with the rule for function decomposition, simply compares the return type of

```
rule bindParamTys((P:Ident,Ps:CIdents); (T:RType, Ts:RTypes); Ls)
    => createVar(P,imm) ~> bindParamTy(P,T) ~> bindParamTys(Ps;Ts;Ls)

rule bindParamTys(.CIdents; .RTypes; _) => .

rule <k> bindParamTy(P,T) => . ... </k>
     <env> ... P |-> I:Int ... </env>
     <varInfo>
        ... var(I) |-> varInfo(_,_,(_ => T),_,_,(_ => true)) ...
     </varInfo>
```

**Figure 4.7: Rewrite Rules Parameter Binding**

the function body to the return type specified in the fnTy definition. To get the return type of the function body, which is initially represented by a single term of sort Exp, the rule has a strict attribute to force the evaluation of the body before comparison. This rewrite rule is what triggers the expression TC. There are three main components of expression TC: function call TC, branch TC, and assignment TC. Expressions in K-Rust are modeled with the sort *Rvalue.* The KResult sort of *expTy(RValue, RType, Int, Int, Int, Mutability)* is used to store information about the RValues as expressions are evaluated, where the fields in order are the RValue itself, the type of the expression, the lifetime of the expression, whether the expression is immutably borrowed, whether it is mutably borrowed, and the mutability of the expression.

```
rule <k> fun F:Ident (P:CIdents) newlft E:Exp endlft =>
        newlft ~> bindParamTys(P;Ts;Ls)  ~> rtTyCK(E,T) ~> endlft ...
    </k>
    <typeCtx>
        ... (F |-> (fnTy(Ls:LftVars; Ts:RTypes; T:RType))) ...
    </typeCtx>

rule rtTyCK(expTy(_,T:RType,_,_,_,_),T) => .
rule rtTyCK(T,T) => .
```

**Figure 4.8: Rewrite Rules for Function Decomposition and Return TC**

The function call term has a strict attribute on the expressions passed as arguments to the function being called. After these expressions are evaluated, the rewrite rule, shown in Figure 4.9, invokes the term *argTyCK* for argument type checking and then rewrites the call to an expTy term with an RType matching the function return type. Argument TC iterates through the function arguments and for each argument checks that it has the same type as the respective function parameter, then moves the argument with the *move* term. The rewrite rules for move modify the varInfo of variables that have the own type to indicate that they have been moved (the initialization field is set to false) as long as they are not already immutably or mutably borrowed.

```
rule <k>
        call F:Ident (Rs:CExps) =>
            argTyCK(Rs,Ts) ~> expTy(call F(Rs),T,L,-1,-1,imm) ...
    </k>
    <typeCtx> ... (F |-> fnTy(_;Ts:RTypes;T:RType)) ... </typeCtx>
    currentLft> L:Int </currentLft>
```

**Figure 4.9: Rewrite Rule for Function Call**

The rules for branch TC, shown in Figure 4.10, decompose if-then statements first by checking that the conditional has type bool by evaluating it with the strict attribute. They then evaluate the first branch using the *execBranch* term. This rule also has a strict attribute and simply evaluates the expression until it reaches a KResult. The computation is then rewritten to the *processBranch* term, which takes as arguments the contents of the varInfo and depGraph cells before the branch execution as well as the second branch expression. ProcessBranch executes the expression it was passed using the execBranch term and restores the varInfo and depGraph maps that were saved before executing the branches. It then uses the terms *combineBrwInfoWith* and *mergeDG* to perform a branch merge. This is necessary because the different branch expressions can result in different possible lifetimes for variables. For example, a variable may be initialized in only one of the branches, or a variable may borrow a different resource in each branch. To solve this problem, the RDG is used to model all possible reference relations between variables.

The RDG is represented as a map from a variable ID to a set of variable IDs that the variable references. These can be thought of as edges in the graph. Operations on the RDG include adding new edges, modifying existing edges, merging, removing nodes, and finding all the direct successors of a given node. Merging two RDGs involves taking the set union of the set of variables in each RDG which a given variable points to. The mergeDG term will merge the two RDGs of the branch executions. The rules for combineBrwInfoWith modify the varInfo cell by adding any new variables from branch execution and combining the varInfo for variables in both by setting the $L_1$ and $L_2$ values to the minimum of the two. That is, the largest lifetime for which the variable is borrowed will be set to the minimum of the largest lifetime for which it was borrowed during the execution of either branch expression.

```
rule <k>
        if expTy(_,bool,_,_,_,_) then  E1:Exp  else  E2:Exp  =>
            execBranch(E1) ~> processBranch(Rho1,Rho2,E2) ...
    </k>
    <varInfo>  Rho1:Map  </varInfo>
    <depGraph> Rho2:Map </depGraph>

rule <k>
        processBranch(R:Map,G:Map,E) => execBranch(E) ~>
            combineBrwInfoWith(Rho1) ~> mergeDG(Rho2) ~> void ...
    </k>
    <varInfo> Rho1:Map => R </varInfo>
    <depGraph> Rho2:Map => G </depGraph>
    rule execBranch(_:KResult) => .
```

**Figure 4.10: Rewrite Rules for Branch Decomposition**

The rules for decomposing an assignment, shown in Figure 4.11, involve the term *processLR*, which further decomposes into a compatibility check between the left and right hand sides of the assignment *cpdCK*, binding the value to the variable with *updateL*, and finally moving the value with the previously described move term.

The cpdCK term simply compares the types and lifetimes of the two sides of the assignment expression for compatibility. The variable binding that occurs in the rules for updateL checks that Rust's borrowing rules are followed with the assignment using the *trySetBorrow* term and updates the varInfo and RDG accordingly with the *updateDG* term.

```
rule L:LValue := R:RValue => processLR(lhs(L),R) ~> void

rule processLR(L:ExpTy,R:ExpTy) => cpbCK(L,R) ~> updateL(L,R) ~> move(R)

rule <k>
        updateL(lhsTy(var(I:Int),_),expTy(R,T:RType,_,_,_,_)) =>
            updateDG(var(I),R,T) ~> trySetBorrow(R,T,L) ...
    </k>
    <varInfo>
        ... var(I) |-> varInfo(L,_,(_ => T),_,_,(_ => true)) ...
    </varInfo>

rule <k>
        updateL(lhsTy(V,_),expTy(R,T:RType,_,_,_,_)) =>
            updateDG(V,R,T) ~> trySetBorrow(R,T,L) ...
    </k>
    <varInfo>
        (.Map => V |-> varInfo(-1,-1,true)) Rho:Map
    </varInfo>
    <currentLft> L:Int </currentLft>
    requires (notBool (V in keys(Rho)))

rule <k>
        updateL(lhsTy(V,T),expTy(R,T:RType,_,_,_,_)) =>
            updateDG(V,R,T) ~> trySetBorrow(R,T,L) ...
    </k>
    <currentLft> L:Int  </currentLft>
    <depGraph> Rho:Map </depGraph>
```

**Figure 4.11: Rewrite Rules for Assignment Decomposition**

Chapter 5

# IMPLEMENTATION

This chapter details the contributions to the K-Rust implementation made by this thesis. An overview of Rust's match expressions is followed by a description of the rules added to both the type checking semantics and the operational semantics of K-Rust to support these match expressions.

## 5.1 The Match Syntax

In Rust match expressions, patterns are compared to a value to determine the control flow of the program. The syntax of these expressions is shown below: with the **match** keyword followed by the value to be matched, then one or more pattern-expression branches.

```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

Rust requires that the patterns be exhaustive. That is, they must account for every possibility of the value being matched. One easy way to achieve this is by using the wildcard pattern "`_`" in the last branch, which will match to any value. The simplest way to match patterns is by matching literals. Here the patterns being matched to are all fixed values. For example, Figure 5.1 matches an integer value to integer literals to print the value as a string, and ends with a wildcard pattern to account for all remaining integers not included in the previous patterns. The value of x will match to the first pattern and the program will output "one", as expected.

```rust
let x = 1;
match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("other"),
}
```

**Figure 5.1: Matching Literals Example**

Rust also allows matching named variables. The match expression will start a new scope, so a named variable in a pattern will shadow any variables of the same name outside of the scope of the expression. The example in Figure 5.2 from the Rust documentation [10] shows a match expression with a named variable in a pattern. The variable x is declared as an Option type, a builtin enum type that can be either None or Some(T), where T is a value. Here x is initialized to Some(5). The pattern on line 4 will compare the values inside the Some values, which here do not match. On line 5, the named variable y is used. This will match to any value in the Some value of x, so the branch will be taken. Since the match expressions starts a new scope, this y variable will shadow the previously declared y variable on line 2. Instead of the value 10 that y was initialized to, the new y variable is bound to the inner value of the Some value of x. Thus, the output will be “Matched, y = 5”. After the match expression, the new y goes out of scope and the value of y will once again be 10.

```rust
1  let x = Some(5);
2  let y = 10;
3  match x {
4      Some(50) => println!("Got 50"),
5      Some(y) => println!("Matched, y = {:?}", y),
6      _ => println!("Default case, x = {:?}", x),
7  }
```

**Figure 5.2: Matching Named Variables Example**

Match expressions can also be used to destructure structs by matching against all the fields of the struct in each pattern. The destructuring functionality is outside the scope of this implementation and will be described further in the Future Work chapter.

## 5.2 Type Checking Semantics

As the type checking semantics are responsible for type checking and borrow checking, the rules added for match expressions perform type checking on the values, patterns, and expressions of the match construct as well as borrow checking on the branches. The syntax added to support match statements is shown in Figure 5.3. A match expression is written as the **match** keyword followed by an RValue and a CExp (a list of expressions) in curly brackets. This rule has a strict attribute on the RValue so that the value will be evaluated before it is matched. Because the "=>" operator is used in K rewrite rules, it is replaced with the operator ">>" in surface-rust syntax for the pattern-expression branches. Similary, the wildcard underscore character, which is also used in K rewrite rules, is replaced with a double underscore "__". The rule on line 6 simply evaluates this wildcard to an expTy with the value "wildcard", which is used in the match expression rewrite rules.

```
1  syntax Exp ::= "match" RValue "{" CExps "}"        [strict(1)]
2              | Exp ">>" "{" Exp "}"
3              | "__"
4
5  syntax RValue ::= "wildcard"
6  rule __ => expTy(wildcard,#TyUndef,0,-1,-1, imm)
```

**Figure 5.3: Type Checking Syntax for Match Expressions**

Similar to decomposition of if-then statements, match expressions are decomposed by checking each branch one by one, then merging branches with the same process-Branch rules used by the if-then statements, shown in Figure 4.10. This ensures that differing variable lifetimes for different branches are consolidated. Figure 5.4 shows the syntax of sort TyCKItem (type check item), which is used to define terms used in match branch checking. The *branchItem* term, defined on line 2, is the main construct used to iterate over match branches. The fields of this term are the RValue of the value being matched, the RType of the value being matched, an RType to hold the type of the branch expressions, an Int to hold the current branch number, a list of expressions making up the body of the match expression, and Int used in determining if the patterns are exhaustive. The third and sixth terms are both specified to be the generic KItem sort and given the strict attribute because the these fields are both passed as syntax terms that must be evaluated before the branchItem rewrite rules are applied (*getBranchType* and *isWildcard* respectively). The *matchPatternTypes* term simply verifies that two terms have the same type, and is strict in the second field to evaluate the type of the expression being compared to the type in the first field.

```
1  syntax TyCKItem ::=
2                  "branchItem" "(" RValue ";" RType ";" KItem
3                      ";" Int ";" CExps ";" KItem ")"        [strict(3,6)]
4                  | matchPatternTypes(RType, KItem)          [strict(2)]
5                  | getBranchType(Exp)                       [strict]
6                  | isWildcard(KItem, KItem)                 [strict]
```

**Figure 5.4: Match Branch Type Checking Syntax**

### 5.2.1 Matching Literals

Figure 5.5 presents the rewrite rules for decomposing match statements. The first rule on line 1 rewrites the match expressions into the first branchItem term. For a match expression to type check, all the patterns must have the same type as the value being matched, and all expressions must have the same type as one another. Since the type of the branch expressions can be anything, the type of the first branch must be evaluated first and then compared to all subsequent branches. To accomplish this, the rule on line 2 rewrites a branchItem on its first iteration by matching the value 1 in the current branch number field. It also requires that the first expression in the match

```
1  rule match expTy(R,T,A,B,C,D) Es:CExps => branchItem(R; T; T; 1; Es; 0)
2  rule <k>
3          branchItem(R:RValue;T1;T2;1;E1 >> {E2}, Es:CExps;A) =>
4              matchPatternTypes(T1, E1)
5              ~> execBranch(E2)
6              ~> processBranch(Rho1,Rho2,
7              branchItem(R;T1;getBranchType(E2);2;Es;isWildcard(A, E1))) ...
8      </k>
9      <varInfo> Rho1:Map </varInfo>
10     <depGraph> Rho2:Map </depGraph>
11 rule <k>
12         branchItem(R;T1;T2;N;E1 >> {E2}, Es:CExps;A) =>
13             matchPatternTypes(T1, E1) ~> matchPatternTypes(T2, E2) ~>
14             execBranch(E2) ~> processBranch(Rho1,Rho2,
15                 branchItem(R;T1;T2;N +Int 1;Es; isWildcard(A, E1))) ...
16     </k>
17     <varInfo> Rho1:Map </varInfo>
18     <depGraph> Rho2:Map </depGraph>
19
20 rule getBranchType(expTy(_,T,_,_,_,_)) => T
21 rule getBranchType(void) => void
22
23 rule matchPatternTypes(_, expTy(wildcard,_,_,_,_,_)) => .
24 rule matchPatternTypes(T, expTy(_,T,_,_,_,_)) => .
25 rule matchPatternTypes(own(T), expTy(_,T,_,_,_,_)) => .
26 rule matchPatternTypes(void, void) => .
27
28 rule isWildcard(_, expTy(wildcard,_,_,_,_,_)) => expTy(1,i32,0,-1,-1,imm)
29 rule isWildcard(E, _) => E
30 rule branchItem(_;_;_;_;.CExps; expTy(1,i32,0,-1,-1,imm)) => .
```

**Figure 5.5: Rewrite Rules for Match Type Checking**

statement body is of the form “Exp >> {Exp}”. To type check the first branch, this rule first matches the type of the value being matched T1 and the type of pattern of the first expression E1 using the matchPatternTypes term on line 4. The rules for this term, shown on lines 23-26, will type check as long as the types are the same or the type of the pattern is a wildcard. The next step, shown on lines 5-7, is executing the branch expression E2 and processing the branch using the processBranch term shown previously in Figure 4.10. The third field of the processBranch term should be the next branch to be evaluated, so here it is set to a new branchItem term on line

7. On this first iteration, the Rtype in the third field is set to the type of first branch expression using the term getBranchType on the expression E2. The rules for this term, shown on lines 20-21, simply rewrite to the type of the expression passed. The branch number field is increased to 2 so this rule is not used again on the subsequent branches. The expressions passed will be the list of expressions without the first expression already type checked by this rule. Finally, the field used to check that patterns are exhastive is evaluated with the term isWildcard.

For simplicity, exhaustive patterns are enforced by requiring that all match expressions must contain a wildcard pattern. This is accomplished by the *isWildcard* term. Every branchItem has as its last field an integer which is initialzed to 0 and set to 1 whenever a wildcard pattern is encountered. Unreachable patterns (patterns that cannot be reached because the patterns above it are exhaustive) are only a compiler warning in Rust and so are ignored by this implementation. This means the wildcard pattern can occur in any of the branches within the match expression, though any pattern that comes after a wildcard will be unreachable. The isWildcard rewrite rules are shown on lines 28-29. When all the branches of a match expression have been checked, the resulting branchItem containing an empty expression list must have a 1 in last field to pass the type checker, shown on line 30. This ensures that a wildcard pattern was encountered at some point during the branch checking, so the match expression is guaranteed to be exhaustive.

The second rule for rewriting branch items, shown on line 12, rewrites the remaining branch expressions in much the same way as the first rule. The only difference is that now the type of each branch expression is compared to the type of the first expression using the matchPatternTypes term.

### 5.2.2 Matching Named Variables

The defined behavior in Rust for binding a named variable in a pattern is as follows. Named variables in a pattern are bound to the value being matched with one of three binding modes (move, copy, or reference), where the scope of the binding is the expression of the given branch. The binding mode is specified in the pattern. If no mode is specified, it will be move by default. If the value implements Copy (a method for copying the value into a new location), it will be copied instead. To bind a reference to the value, the keyword **ref** can be used before the variable name in the pattern, or **ref mut** for a mutable reference. This is used in place of the **&** operator so there is no effect on the matching itself; **ref** is exclusively an indicator of how to bind the value to the variables in a pattern. To handle cases where a referenced value is matched to a non-referenced pattern, Rust has a strategy for determining the binding mode of a pattern. If this case is encountered, the value is dereferenced and the binding mode is changed to reference. If the reference is mutable, the binding mode will be changed to mutable reference unless it is already set to reference, in which case it is not changed. The process of dereferencing the value repeats for nested references until the value itself is reached.

The above rewrite rules will work for matching named variables with one addition. When defining a pattern containing a named variable in surface-rust, the expression that follows must first bind the value being matched to the variable named in the pattern. This step can be accomplished when translating a Rust program into the surface-rust syntax by rewriting the term "match VALUE {PATTERN(x) => EXPRESSION}" to "match VALUE {PATTERN(x) >> let x = VALUE in {EXPRESSION}}". Because of the rules already defined for assignment, this will achieve the desired result of any named variables shadowing previously defined variables of the same name inside the body of the match expression. Here the binding mode for the variable can be defined explicitly in the code itself, so it does not need to be determined from the pattern.

## 5.3 Operational Semantics

The match expression rules for the operational semantics are much simpler than those for the type checking semantics. They involve evaluating the pattern value to find the correct branch to take. Since the only types allowed in the operational semantics are integers and arrays, these rules are defined only for matching integers. These rules expand upon the previously defined case statements for the operational semantics, which originally could not have defined patterns, but instead used increasing integer values as the default for pattern values and were used only for rewriting if-then statements. The expanded syntax terms for Exp, shown in Figure 5.6, include the same terms from the type checking semantics for pattern-expression and wildcard. The pattern-expression term is strict on the first Exp to evaluate the pattern before matching occurs. Instead of “match VALUE {...}”, the syntax for a match expression in core-language is “case VALUE of {...}”.

```
syntax Exp ::=
            Exp ">>" "{" Exp "}"        [strict(1)]
            | "__"
```

**Figure 5.6: Operational Syntax for Match Expressions**

Figure 5.7 shows the rewrite rules for match operations, which in the operational semantics are rewritten to the term *caseItem*. A caseItem has the value being matched as the first agrument and the list of expressions making up the body of the match expression as the second argument. The term is strict in the first argument to evaluate the value being matched before matching occurs. The rule on line 4 rewrites if-then statements to a case statement where the first pattern is 1 for true and the second is 0 for false (recall that in the the operational semantics booleans are rewritten to integers). This rule eliminates the need to translate if-then statements in core-language, as the rewrite rule will translate them automatically. The rule for decomposing a case statement, shown on line 6, rewrites the case statement as a caseItem. The caseItem rewrite rules iterate through the branches of the case statement. The rule for a caseItem with a mismatched pattern, shown on line 8, checks the pattern of the first expression, then rewrites the term to a caseItem with the remaining expressions. The rules for a matching pattern, shown on lines 10 and 12, rewrite to the branch expression if the pattern matches the value or the pattern is a wildcard.

```
1  syntax CaseItem
2         ::= caseItem(Exp, ExpList) [strict(1)]
3
4  rule if E:Exp then E1:Exp else E2:Exp => case E of {1 >> {E1}, 0 >> {E2}}
5
6  rule case I of { EL:ExpList} => caseItem(I, EL)
7
8  rule caseItem(I, P >> {_}, EL:ExpList) => caseItem(I, EL)
9
10 rule caseItem(I, I >> {E}, EL:ExpList) => E
11
12 rule caseItem(I, __  >> {E}, EL:ExpList) => E
```

**Figure 5.7: Rewrite Rules for Match Operations**

## 5.4 Type Checking and Evaluating a Match Expression

```
let x = 1;
match x {
    0 => false,
    _ => true
}
```

**Figure 5.8: Match Example Rust Code**

This section demonstrates the rewrite process for a simple match expression in both the type checking semantics and the operational semantics. The Rust code in

Figure 5.8 matches an integer variable x, and will evaluate to false if the integer value is 0 and true otherwise.

First the code will be evaluated with the type checking semantics. The translation into surface-rust code is shown in Figure 5.9. The code on lines 1-2 will create the variable x and initialize its value to 1. Next the match expression is encountered.

```
1  let x in {
2      x := 1;
3      match x {
4          0 >> {false},
5          __ >> {true}
6      }
7  }
```

**Figure 5.9: Match Example Surface-Rust Code**

Match terms are strict on the value being matched, so it will be evaluated first. This will involve a simple variable lookup, which will result in the RValue expTy (1, i32, 0, -1, -1, imm), representing the i32 1. The first rewrite rule for match terms can now apply. It will put the match expression into a branchItem term as follows:

```
    branchItem(1; i32; i32; 1; 0 >> {false}, __ >> {true}; 0)
```

Now the rewrite rules for branchItem will be applied. Since the integer in the fourth field is 1, the rule for checking the first branch arm of the match expression will apply on the expression "0 >> {false}". The branchItem will be rewritten as follows, where Rho1 is the varInfo map containing the variable x and Rho2 is the depgraph map, which is currently empty.

```
matchPatternTypes(i32, 0) ~> execBranch(false) ~>
    processBranch(Rho1, Rho2, branchItem(1; i32; getBranchType(false);
        2; __ >> {true}; isWildcard(0, 0)))
```

The matchPatternTypes term will rewrite to the empty computation, since the type of the expression 0 is i32. The execBranch term will similarly rewrite to the empty computation as the expression false type checks. Next the rewrite rule for processBranch will apply. The computation now looks like this:

```
execBranch(branchItem(1; i32; getBranchType(false); 2; __ >> {true};
    isWildcard(0, 0)))
~> combineBrwInfoWith(Rho1) ~> mergeDG(Rho2) ~> void
```

First the execBranch term will evaluate the second branchItem term. Now the fourth field of the branchItem is 2, so the rule for evaluating the remaining branches of the match expression will apply. This rule will match the types of the branch expressions by comparing them to the type of the first branch expression. The branchItem term is strict in the third and sixth fields, so before rewriting the term, the getBranchType and isWildcard terms must be evaluated. The getBranchType term evaluates to the type of the expression false, which is bool. Since the value passed to isWildcard is not a wildcard, it will evaluate to 0. The computation now looks like this:

```
branchItem(1; i32; bool; 2; __ >> {true}; 0)
~> combineBrwInfoWith(Rho1) ~> mergeDG(Rho2) ~> void
```

After rewriting the brachItem term for the second branch of the match expression, the computation looks like the following (note that since no borrowing occurs, the values of Rho1 and Rho2 do not change throughout this process).

```
matchPatternTypes(i32, __) ~> matchPatternTypes(bool, true) ~>
execBranch(true) ~> processBranch(Rho1, Rho2, branchItem(1; i32;
    bool; 3; .CExps, isWildcard(0, __)))
~> combineBrwInfoWith(Rho1) ~> mergeDG(Rho2) ~> void
```

As before, the matchPatternTypes terms both rewrite to the empty computation since the types match, and execBranch rewrites to the empty computation because the expression true will type check. Now the rewrite rules for the second process-BranchTerm can apply, resulting in the following computation.

```
execBranch(branchItem(1; i32;
    bool; 3; .CExps, isWildcard(0, __))) ~>
~> combineBrwInfoWith(Rho1) ~> mergeDG(Rho2) ~> void
~> combineBrwInfoWith(Rho1) ~> mergeDG(Rho2) ~> void
```

The isWildcard term will be evaluated before rewriting the branchItem term. Since this expression is a wildcard, the term will evaluate to 1. Now the computation looks like this:

```
branchItem(1; i32; bool; 3; .CExps, 1) ~>
~> combineBrwInfoWith(Rho1) ~> mergeDG(Rho2) ~> void
~> combineBrwInfoWith(Rho1) ~> mergeDG(Rho2) ~> void
```

The branchItem term has a 1 in the wildcard field, meaning it encountered a wildcard pattern in at least one branch, and an empty list of expressions, so it will be rewritten to the empty computation. Since the Rho1 and Rho2 maps did not change throughout the process, the combineBrwInfoWith and mergeDg terms will not change the contents of the varInfo and depgraph maps and will all rewrite to the empty computation. The type checking of the match expression is now complete.

The next step is to evaluate the code on the operational semantics. For this it must be translated into core-language, which is shown in Figure 5.10.

```
let x = 1 in (
    case x of {
        0 >> {false},
        __ >> {true}
    }
)
```

**Figure 5.10: Match Example Core-Language Code**

As with the type checking semantics, the focus of this example will be only the rewrite rules for the case expression. The case term is strict on the value being matched, so the first step is a variable lookup on x, which will result in the Int value

1. Then the rule for case terms will apply to rewrite the expression into a caseItem term as follows.

```
caseItem(1, (0 >> {false}, __ >> {true}))
```

Now the caseItem will attempt to find a matching rewrite rule by examining the pattern of the first expression 0 >> false. This pattern of 0 does not match the value 1, so the computation is rewritten to a caseItem with the remaining expression list.

```
caseItem(1, (__ >> {true}))
```

Now the pattern being compared is a wildcard, so it will match the value 1. The computation is then rewritten to the expression of this branch: true. In core-language, booleans are rewritten to integers, so the final result of this code will be the value 1.

# Chapter 6

# TESTING AND VALIDATION

## 6.1 Translating Rust Programs

In order to run Rust programs on the compiled K framework models, they must first be translated into the defined syntax for those models. As previously mentioned, the language defined for the type checking semantics is called *surface-rust*, and the syntax for the operational semantics is called *core-language.* Running a Rust program with the entire implementation requires that the program first be translated into surface-rust and run on the type checking semantics, then translated further into core-language and run on the operational semantics. The details of this translation are provided below.

### 6.1.1 Rust to Surface-Rust

As the syntax for surface-rust closely resembles that of Rust, translating a Rust program into surface-rust is fairly straightforward. Table 6.1 highlights the main syntactic differences between Rust and surface-rust and how they are translated. In surface-rust, all compound types, including function types, must be defined at the start of the program with the ":=:" operator. Surface-rust also uses the keyword **fun** instead of Rust's **fn** for functions and the **newlft** and **endlft** keywords to start and end a function body in place of curly brackets. To create new variables, surface-rust uses the **new** keyword and desired variable type. New variables cannot be initialized directly, and the scope of a new variable must be strictly specified by the "in {}" following the variable declaration. Row three of the table shows how a new variable of type Point is defined in surface-rust and then set to a certain value using the ":="

operator. This row also shows how functions are called in surface-rust, with the **call** keyword. The last row of the table shows the match syntax translation.

**Table 6.1: Translating Rust to Surface-Rust**

| Rust | Surface-Rust |
|---|---|
| fn main() {...} | main :=: fnTy(;;void)<br>fun main() newlft<br>...<br>endlft |
| struct Point {x: i32, y: i32} | Point :=: prodTy(i32,i32) |
| let p = Point {x: 0, y: 1};<br>let x = get(&mut p);<br>... | let p = new(ty(Point)) in {<br>   p.1 := 0<br>   p.2 := 1<br>   let x = call get(& mut p) in {...} |
| let x = 19;<br>match x {<br>   0 => 0,<br>   1 => 1,<br>   _ => 2<br>} | let x = new(i32) in {<br>x := 19<br>match x {<br>   0 >> {0},<br>   1 >> {1},<br>   __ >> {2}<br>} |

### 6.1.2 Surface-Rust to Core-Language

After a surface-rust program passes the type checker, it must be translated again into the syntax of core-language. Table 6.2 shows some of the main syntactic differences between surface-rust and core language. It is assumed that any program being run on the operational semantics has already passed the type checker, so there is no need for type annotations in core-language. The function syntax for core-language is the same as that of Rust. The match syntax is the same as that of surface-rust except the **match** keyword is replaced with **case of**. Row three illustrates how variable declarations are replaced with function calls in core-language. Creating a value with the **new** keyword in surface-rust is replaced with a memory allocation.

The **na** keyword used to translate dereferences refers to non-atomic memory reads. The syntax for assigning a value to a variable does not change. The last row of the table shows how a sequential composition of expressions is translated into a tail function call, a call which does not allocate any stack space.

**Table 6.2: Translating Surface-Rust to Core-Language**

| Surface-Rust | Core-Language |
|---|---|
| fun main() newlft ... endlft | fn main () {...} |
| match x { 0 >> 0, 1 >> 1, _ >> 2 } | case x of { 0 >> {0}, 1 >> {1}, _ >> {2} } |
| let x = e1 in {e2} | (fn(x){e2})(e1) |
| new(T) | allocate(N), where N is the size of the type T |
| *v | * na v |
| x := v | x := v |
| *x := v | x := na v |
| e1; e2 | tailcall((fn (#anonymous) {e2}) (e1)) |

## 6.2 Testing Benchmarks and Example Test Cases

The main tests used to validate this implementation were the tests provided by the K-Rust implementation at [1]. As a newer version of the K-framework required significant refactoring of the original implementation, these tests were used to verify that the implementation retained the original functionality. The tests are organized into six categories: ownership, branches, sum types, product types, lifetimes, and functions. Many of the tests are examples taken from the Rust documentation [10]. In addition to these tests, additional tests were created to target the functionality of the new match semantic rules. Some examples of these tests are provided in the following sections. To test a program, the results of the K execution are compared to the results of the Rust compiler. The purpose of testing on a semantic model for a language is twofold: testing must verify that the implementation correctly models the Rust language as specified, and also serves to verify the correctness of the Rust compiler itself. For example, test cases taken directly from the Rust documentation are assumed to compile correctly, so the K execution should match the Rust compiler in these cases. If the Rust compiler accepts the program and produces the correct result, the K execution should do the same. If the Rust compiler rejects the program, the K execution should get stuck at the same point as the Rust compiler. That is, the reason for rejecting the program should be the same for both executions. Testing the Rust compiler involves creating test cases by deriving the expected result from the documented Rust specifications, then verifying the Rust compiler matches the K execution for the semantic model of these specifications. For the purpose of this implementation, tests created for the match expression rules serve only to test the correctness of the implementation itself, though the test cases provided by the K-Rust implementation include both types.

### 6.2.1 Match Test Example

```
fn f(X:Box<i32>,Y:Box<i32>) -> i32 {
    match *X {
        1 => *X + 1,
        2 => *X + 10,
        _ => *Y
    }
}
let x = Box::new(1);
let y = Box::new(3);
f(x,y)
```

**Figure 6.1: Match Test Example Rust Program**

This example was created to test the match semantics and shows the execution of a Rust program from beginning to end with the K framework implementation. The original Rust program is shown in Figure 6.1. This program creates two variables using the Box constructor (a builtin Rust struct that allocates values on the heap instead of the stack). It then calls the function f with the two variables which will return an i32 value based on a match statement. This program will compile and will match the first branch of the match expression, so the output will be 2.

Figure 6.2 shows the program translated into surface-rust syntax. This code passes the type checker as expected (the contents of the k cell are the empty computation when execution is finished).

```
Box :=: prodTy(i32)
main :=: fnTy(;;i32)
f :=: fnTy(;own(ty(Box)),own(ty(Box));i32)
fun f(X, Y) newlft
    let return in {
        match X.(1) {
            1 >> {return := X.(1) + 1},
            2 >> {return := X.(1) + 10},
            __ >> {return := Y.(1)}
        };
        return
    }
endlft
fun main() newlft
    let x = new(ty(Box)) in {
        x.(1) := 1;
        let y = new(ty(Box)) in {
            y.(1) := 3;
            call f(x,y)
        }
    }
endlft
```

**Figure 6.2: Match Test Example Surface-Rust Program**

Finally, Figure 6.3 shows the core-language translation of the program (sc is an atomic memory read). After execution finishes the k cell contains the expected result of 2. Since this test matches the behavior of the Rust compiler it is considered passed.

```
fn f(X,Y) {
    case (*sc X) of {
        1 >> {(*sc X) + 1},
        2 >> {(*sc X) + 10},
        __ >> {(*sc Y)}
    }
};
let x = allocate(1) in (
    x := na 1;
    let y = allocate(1) in (
        y :=na 3;
        f(x,y)
    )
)
```

**Figure 6.3: Match Test Example Core-Language Program**

### 6.2.2 Compiler Rejection Example

This example, from the test cases provided by the K-Rust implementation, shows a test program that is rejected by the Rust compiler. The original Rust program (from the Rust documentation [10]) is shown in Figure 6.4. This program results in a compiler error because y borrows x on line 4, then x goes out of scope and is dropped on line 5, but y attempts to reference the dropped value on line 6. This is an example of a “used after free” error.

```
1 let y;
2 {
3     let x = 5;
4     y = &x;
5 }
6 println!("{}", &y);
```

**Figure 6.4: Compiler Rejection Example Rust Program**

Since the borrow checker is part of the type checking semantics, this program should not pass the type checking stage of execution. Figure 6.5 shows the surface-rust translation of the program.

```
1  main :=: fnTy(;;void)
2
3  fun main() newlft
4      let y in {
5          newlft
6              let x = new(i32) in {
7              *x := 5;
8              y := & imm x
9              }
10         endlft;
11             let print = & imm y in void
12          }
13 endlft
```

**Figure 6.5: Compiler Rejection Example Surface-Rust Program**

Figure 6.6 shows the resulting k cell when execution is finished. Clearly this program did not pass the type checker. Examining the contents of this cell shows where the execution got stuck. The “#freezer” followed by a rule name indicates that the computation got stuck while trying to complete the named rewrite rule. Computations are rewritten from left to right, so the first term in the k cell is the term that was unable to match any rule. Here the term is *lessInt*, which was written somewhere inside the frozen term *processLR*. By examining the semantic rules and the rest of the resulting configuration, it can be determined that the lessInt term is used by the *cpbCK* term for compatibility checking an assignment within the processLR rule. The lessInt term is a K builtin which will rewrite to the empty computation if the first integer agrument is less than or equal to the second. Here it is used for comparing lifetimes of the left and right hand sides of the assignment, since a value can only be assigned to a variable if the variable’s lifetime includes the value’s lifetime. After the two newlft terms, the variable x had a lifetime of 2, so when y was set to a reference to x, its lifetime also became 2. Since the previous lifetime ended on line

10, the lifetime of the new variable print on line 11 is 1. Then when lessInt is called, the lifetime of the right hand side of the expression (y) has a lifetime of two (the first argument to lessInt), but the lifetime of the left hand side (print) has a lifetime of 1 (the first field of the expTy in the second lessInt argument), so the rule gets stuck. This is the same reason that the program was rejected by the Rust compiler, so this test passes.

```
<k>
     lessInt(2, expTy(1, i32, 0, -1, -1, imm ) ) ~>
        expTy(& imm var(0),
            ref(1, imm, ref(2, imm, own(i32))), 1, 1, -1, mut)
        ~> #freezerprocessLR(_,_)_LSTATEMENT_TyCKItem_KItem_KItem1_(
        lhs(print)~>.) ~> void ~>
        #freezerrtTyCK(_,_)_LFUNCTION_TyCKItem_Exp_K0_ (void~>.)
        ~> endlft ~> .Decls ~> .
</k>
```

**Figure 6.6: Compiler Rejection Example Results**

# Chapter 7

# FUTURE WORK

## 7.1 Implementation

The primary focus of future work for this implementation should be further expanding its scope. While the match semantics for the type checking system can include structs and named variables, the operational semantics currently only supports the primitive types of integer and boolean. The first goal of any future work would be to include the third type allowed by the operational semantics: arrays. Arrays in core-language are a way of representing compound data types at the memory level. Including them in the match semantic rules would allow support for the destructuring of structs and enums allowed by Rust. Destructuring involves matching against a compound data type like a struct by matching against each of the individual fields within the type. Patterns of the type being matched may contain fields with literals, fields with named variables, or some combination of the two. Matching against compound types is possible with the type checking semantics because it simply verifies that all patterns have the same type as the value being matched. However, matching compound data types to determine which branch to take, as the operational semantics would do, requires more complex rewrite rules that take into account proper variable bindings.

Another option for expanding the implementation is including semantic rules for Rust's unsafe constructors. These are Rust libraries that are implemented using unsafe code, which allows the use of features that are not checked by the compiler for memory safety. Unsafe libraries are largely ignored by efforts to formalize Rust, with the exception of the Rustbelt implementation [8], which outlines a verification process for including unsafe features in the formal proofs given for safe Rust. A non-trivial number of Rust's core libraries are implemented with some amount of unsafe code, and more focus should be placed on these features in future work exploring the safety of the language.

## 7.2 Testing

The requirement of manually translating programs from Rust to surface-rust to core-language creates a bottleneck in the testing capability of this implementation. Creating an automated translation process of programs would allow for a much more exhaustive testing suite that could take unaltered Rust programs as input directly. This could be achieved by adding rewrite rules to the implementations which would simply rewrite the input language syntax to its appropriate translation before the execution begins. K provides support for such rules with the *structural* attribute. This is used for rules that only serve to rearrange the structure of a computation without making any computational steps. Tests that are rejected by the compiler would still require manual verification of the reason for failure, but an automated translation tool would significantly improve the practical usefulness of this implementation.

Chapter 8

# CONCLUSION

This thesis had the primary goal of getting incrementally closer to a complete formal semantics of the Rust programming language. This was achieved by expanding the scope of K-Rust, a formal semantic model implemented in the rewrite based semantic framework K. The implementation is now compatible with version 5 of the K framework, the most recent version to date. It is also able to perform type checking, borrow checking, and operational execution on Rust's match expressions. All tests run on the implementation match the results of the Rust compiler, both for cases where the program is accepted and executed and cases where the compiler rejects the program. The goal of any future testing, which would be made more efficient with automated program translation capabilities, would be to further verify the correctness of the implementation and ideally identify use cases where the Rust compiler produces an incorrect result based on the defined rules for ownership and borrowing.

In summary, the major contributions of this work include:

- Reworking the K-Rust implementation so that it is compatible with a newer version of the K framework
- Extending the functionality of the K-Rust implementation to include pattern matching semantics
- Testing and validating the functionality of the extended K-Rust implementation against the Rust execution environment

Formal semantics, while often overlooked or disregarded in practice, are essential for ensuring the safety and reliability of programming tools. Rust's principles of ownership and borrowing present an interesting new programming language paradigm with great potential, but will ultimately require more work in order to formally verify their safety guarantees.

# BIBLIOGRAPHY

[1] K-rust formal semantics for the rust programming language in the k-framework. `https://securify.sce.ntu.edu.sg/SoftVer/KRUST/`.

[2] K tutorial. `https://github.com/kframework/k/tree/master/k-distribution/tutorial`, 2020.

[3] D. Bogdanas and G. Roşu. K-java: A complete semantics of java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 445–456, New York, NY, USA, 2015. Association for Computing Machinery.

[4] D. Bogdanas and G. Roşu. K-java: A complete semantics of java. *SIGPLAN Not.*, 50(1):445–456, Jan. 2015.

[5] C. Ellison and G. Rosu. An executable formal semantics of c with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 533–544, New York, NY, USA, 2012. Association for Computing Machinery.

[6] C. Ellison and G. Rosu. An executable formal semantics of c with applications. *SIGPLAN Not.*, 47(1):533–544, Jan. 2012.

[7] D. Filaretti and S. Maffeis. An executable formal semantics of php. In *European Conference on Object-Oriented Programming*, pages 567–592. Springer, 2014.

[8] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017.

[9] S. Kan, D. Sanán, S.-W. Lin, and Y. Liu. K-rust: An executable formal semantics for rust. *arXiv preprint arXiv:1804.07608*, 2018.

[10] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[11] N. D. Matsakis and F. S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[12] P. D. Mosses. Formal semantics of programming languages: — an overview —. *Electronic Notes in Theoretical Computer Science*, 148(1):41 – 73, 2006. Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004).

[13] D. Park, A. Stefănescu, and G. Roşu. Kjs: A complete formal semantics of javascript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–356, 2015.

[14] E. R. Reed. Patina : A formalization of the rust programming language. 2015.

[15] G. Rosu. K: a rewrite-based framework for modular language design, semantics, analysis and implementation-version 2. Technical report, 2006.

[16] G. Roșu and T. F. Șerbănută. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397 – 434, 2010. Membrane computing and programming.

[17] F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang. Krust: A formal executable semantics of rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 44–51, 2018.

[18] A. Weiss, D. Patterson, N. D. Matsakis, and A. Ahmed. Oxide: The essence of rust, 2019.