Contents

Abstract

Software correctness is a central goal for software development. One way to improve correctness is using strong and descriptive types and verifying them with type systems. In particular Refinement Types demonstrated, that even with a verification system that is restricted to a decidable logic, intricate properties can be expressed and verified in a lightweight and gradual way. However existing approaches for adapting Refinement Types from functional to imperative languages proved hard without compromising on at least one of core features of Refinement Types. This thesis aims to design a Refinement Type system without such compromises by taking advantage of Rust's restriction to unique mutable references. I will define a Refinement Type language and typing rules and argue for their soundness. Additionally I will implement a prototype verifier to evaluate the effectiveness of the approach on selected examples.

1 Introduction

With increasing amount and reliance of software, ensuring the correctness of programs is a vital concern for the future of software development. Although research in this area has made good progress, most approaches are not accessible enough for general adoption by the developers. Especially in light of a predicted shortage of developers[**breaux_2021_2021-1**], it is not sufficient to require developers to undergo year-long training in specialized and complex verification methods to ensure the correctness of their software. It is therefore crucial to integrate with their existing tooling and workflows to ensure the future high quality of software. One avenue for improving accessability for functional verification is extending the expressiveness of the type system to cover more of the correctness properties. Using type systems for correctness was traditionally prevalent in purely functional languages where evolving states are often represented by evolving types, offering approachable and gradual adoption of verification methods. Tracing evolving states in the type system would be especially useful for languages with mutability, since substantial parts of the behaviour of the program is expressed as mutation of state. In particular Rust seems like a promising target language, because mutability is already tracked precisely and thus promising functional verification for relatively minimal effort on the programmer's part.

2 The Case for Rust as a Target Language

Rust is split into two languages: safe and unsafe Rust. Like most type systems, Rust's type and ownership system is conservative, meaning any (safe) Rust program that type checks, will not crash due to memory safety or type errors. Unsafe Rust gives the programmer the ability to expand the programs accepted by Rust outside that known-safe subset.

In this thesis, we will only consider safe Rust, which is the subset of Rust most programmers interact with (see section 3).

Rust features a few unusual design decisions that profoundly influence the design of verification systems for it. The following paragraphs will elaborate on this.

Aliasing XOR Mutabillity The key behind Rusts type system is, that for every variable at every point during the execution, that variable is either aliased or mutable, but never both at the same time.

Rust accomplished this by introducing three types of access permissions for a variable, which are associated with every scope[1]:

1. The scope *owns* the variable. This guarantees, that no other scope has access to any part of the variable and allows the scope to create references to (part of) the variable.

[1] In modern Rust, this model was generalized to cover more cases, but the idea is still valid

2. The scope (immutably) *borrows* the variable. This guarantees, that as long as the variable is used, its *value will not change* and allows the scope to further lend the variable to other scoped.

3. The scope *mutably borrows* the variable. This guarantees, that there are no other active references to any part of the memory of the variable.

A consequence of these rules it, that at any time, every piece of memory has a unique and compile-time known owner. There are no reference cycles.

This makes both aliasing as well as mutability quite tame: If a variable is aliased, it must be immutable and as a result, represents just a value (like in pure functional languages). If a variable is mutable, it can not be aliased and as a result, any effect of the mutation is well known and locally visible.

Opaque Generics Rust does not provide a way to check a generic parameter for its instantiation. This means that we cannot extract any additional information from a generic parameter `T`. A function from `T` to `T` can therefor only be the identity function. Wadler [**wadler_theorems_1989**] shows, that it is possible to derive facts about the behaviour of such polymorphic functions.

In contrast, languages with instance-of-checks allow an implementation of a polymorphic function to distinguish between different instances of the generic parameter, precluding this extensive reasoning.

Explicit Mutable Access A consequence of the ownership rules is, that a function can only mutate (part of) the state, that is passed to it as a parameter. There is no global state and there is no implicit access to an object instance. Thus any intention of mutating state must already be expressed in the function signature, which makes the specification of this mutation quite natural.

What Rust does not solve Eventhough Rust siplifies reasoning about mutability and aliasing of mutable data, Rust does not eliminate the need to reason about multiple references to mutable data. For example, Listing 2 shows how `cell`'s type is influenced by changes to `r`. This does not mean, that the ownership rule *mutability XOR aliasing* are broken: Eventhough both `cell` and `r` are mutable, but only one is active.

```
let mut cell = 2;
let r = &mut cell;
r+= 1;
assert(cell, 2)
```

3 Usage Of Rust

Before designing a system for Rust, it makes sense to gain some understanding of how Rust is used. For this purpose we will look at two key features of Rust,

that influence how an approchable verification should look like. Firstly `unsafe` Rust with similar guarantees to C would make verification and specification significantly harder. But if the use of `unsafe` is limited, like intended by the language designers, it would allow us to focus on the save part of Rust and leave the verification of `unsafe` Rust to more complex verification systems. Secondly with mutability being the main difference to the traditional domain fof Refinement Types, showing the need for covering this area and to what degree of approachability it interesting.

3.1 Unsafe Rust

There is already some research on how `unsafe` is used in Rust. For example Astrauskas et al. [**astrauskas_how_2020**] found, that about 76% of crates did not use any unsafe. On top of that, unsafe signatures are only exposed by

"The majority of crates (76.4%) contain no unsafe features at all. Even in most crates that do contain unsafe blocks or functions, only a small fraction of the code is unsafe: for 92.3% of all crates, the unsafe statement ratio is at most 10%, i.e., up to 10% of the codebase consists of unsafe blocks and unsafe functions." [**astrauskas_how_2020**] Astrauskas et al. also found, that "However, with 21.3'% of crates containing some unsafe statements and – out of those crates – 24.6% having an unsafe statement ratio of at least 20%, we cannot claim that developers use unsafe Rust sparingly, i.e., they do not always follow the first principle of the Rust hypothesis." [**astrauskas_how_2020**]

I makes sense to further distinguish between libraries and executables crates: Libraries are intendet to be used by other Rust programs. Usage of unsafe in libraries may not be as problematic as in executables, because libraries are written once but used by many applications, justifying higher verification effort.

In our analysis, we found, that firstly crates.io contains significantly more libraries than binaries (see Figure 1). And secondly libraries are much more likely to use `unsafe` Rust. Table shows the result of my own analysis. Where uses of `unsafe` are counted and grouped by crate type (see Table 1). The data in the table includes all crates except `windows-0.32.0`, which alone contains 233 608 uses of `unsafe`. Nearly 2/3 of all other library `unsafe` combined. Looking at the distribution of unsafe uses in Figure 2, we can see, that this is an exception: Most other libraries do not use that much unsafe statements. We can also see, that even if a executable crate uses `unsafe`, it uses few

Crate Type	Number of Unsafe Uses
Library	382 997
Both	7 720
Binary	930
n/a	215

Table 1: Number of Unsafe Uses by Crate Category

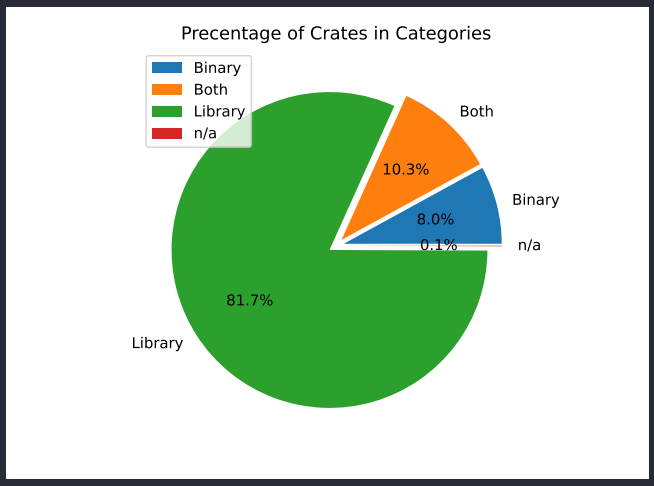

Figure 1: Percentage of Crates, that Contain Libraries, Executables or Both

3.2 Mutability

This was done by analysing all published Rust crates (Rust's version packages) published on crates.io with at least 10 versions were analysed[2]. The analysis uses the syntactical structure to infer mutability information about the following various AST items:

- Local Variable Definitions. These can be tracked with high confidence
- Parameters, which are considered immutable if they are passed as immutable references or owned.
- Function Definitions, which are considered immutable, iff all parameters considered immutable.
- Arguments. Hard to track
- Function Calls, which are considered immutable, iff all arguments are considered immutable.

A total of around 52 million items were found in 228263 files in a combined code-base size of over 64 million lines of Rust code (without comments and white space lines)[3]

Figure 3 shows the ratio of mutable to immutable items. For each syntactical category, the percentages are given relative to different objects:

1. Total: number of unique occurrences

[2]The limit of 10 versions is used to eliminate inactive and placeholder packages
[3]Calculated with `cloc`

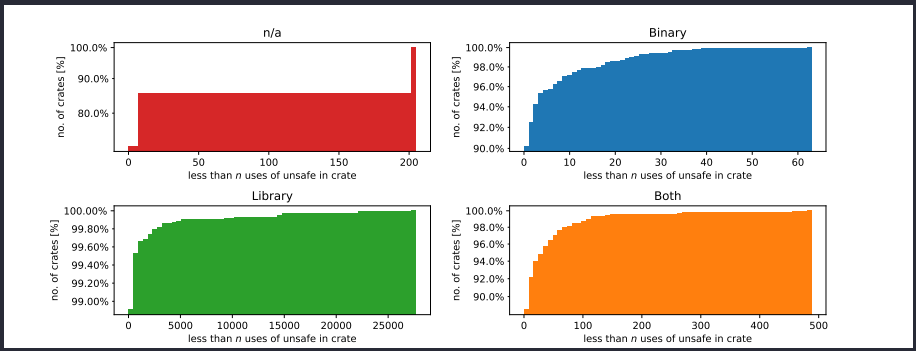

Figure 2: Cummulative, logarithmic histogram of the amount of `unsafe` uses in each category

2. Function: number of unique functions. I.e. 76.54% of functions have a immutable parameter.
3. File: number of unique files

Unfortunately not there are some areas, where the syntactic analysis is not sufficient. Namely the analysis of function calls and arguments, which is mainly caused by the uncertainty of mutabillity of complex arguments. Luckily the data from function definitions and parameters can complete the picture: About 80% of parameters are immutable and between about 10 and 20% of parameters are mutable. And less than 10% of functions have mutable parameters at all. When it comes to local variables, Rust users are more liberal in their use of mutability: About 30% of local variables are defined mutable.

For verification this means, that the use of mutabillity is wide spread. Especially local variables are often mutable and should therefore a verification system should try to minimize the effort for the user. Mutable parameters are less common, but still need to be accounted for in verification.

4 The MiniCorten Language

Rust's main disadvantage as a target language it its size: There is a lot of syntax and semantics that would need to be accounted. A lot of it even incidental to the verification. To reduce the complexity and amount of work, that needs to be done, we will focus on a subset of Rust described in this section.

The goal is to remove as much incidental complexity as possible without compromising to the central topic of research: How to extend LiquidTypes to mutability under the presence of Rust's ownership model.

4.1 Syntax

Call it CortenRust (after Weathering Steel)

This subsection will introduce the syntax of MiniCorten, a language modelled after a simplified version of Rust, with the addition of refinement types. To simplify the formal definitions and proofs, the language is restricted to ANF, meaning arguments of expressions must be variables. Note that the implementation does not have this restriction.

What does the abbrev stand for? XXX Normal Form

program	::=	*func_decl* *	*function declaration*
func_decl	::=	*ident*(*param* *) -> *ty* { *stmt* }	
param	::=	*ident* : *ty*	
stmt	::=	*expr*	*expression*
	|	let mut? *ident* = *expr*	*declaration*
	|	*ident* = *expr*	*assignment*
	|	while (*expr*) { *stmt* }	*while loop*
	|	relax_ctx!{ *pred** ; (*ident* : *ty*)* }	*context relaxation*
expr	::=	*ident*	*variable reference*
	|	*lit*	*constant*
	|	*expr* + *expr*	*addition*
	|	*ident*(*ident* *)	*function call*
	|	if *expr* { *stmt* } else { *stmt* }	*if expression*
	|	*stmt*; *expr*	*sequence*
	|	* *ident*	*dereference*
	|	& *ident*	*immutable reference*
	|	&mut *ident*	*mutable reference*
	|	*expr* as *ty*	*type relaxation*
ty	::=	ty!{ *logic_ident* : *base_ty* | *pred* }	*refinement type*
pred	::=	*ref_pred*	*predicate for a reference type*
	|	*value_pred*	*predicate for a value type*
ref_pred	::=	*logic_ident* = & *ident*	
	|	*ref_pred* || *ref_pred*	*mutable reference*
value_pred	::=	*logic_ident*	*variable*
	|	*pred* && *pred*	*conjunction*
	|	*pred* || *pred*	*disjunction*
	|	! *pred*	*negation*
base_ty	::=	i32	*integer*
	|	unit	*unit type*
	|	bool	*boolean*
	|	& *base_ty*	*immutable reference*
	|	&mut *base_ty*	*mutable reference*
lit	::=	0,1,...,n	*integer*
	|	true	*boolean true*
	|	false	*boolean false*
	|	()	*unit value*

4.2 Semantics

The semantics are mostly standard. The main difference being that Rust and our simplified language enable most statements to be used in place of an expression. The value is determined by the last statement in the sequence. For example If-Expressions, sequences of statements and function all follow this rule: Their (return) value is determined by the last statement or expression in the sequence.

The semantics loosely based on Jung's MiniRust.

Another difference between Rust and MiniCorten is of course the addition of refinement types.

In terms of the formal description, the rules are similar to Pierce's [**pierce_types_2002-3**] Reference language. The main difference is, that in Rust, every piece of data has a unique, known owner. This fact makes the concept of locations redundant. Instead we treat `ref` x as a value itself. The following definitions show the new execution rules.

Definition 4.1 (Execution-State)**.** The state of execution is given by σ : PVar $\to$ Value. The set of function declarations is constant and given by Σ : Fn-Name $\to ((\text{Arg}_1, \ldots, \text{Arg}_n), (ReturnValue))$

Definition 4.2 (Small-Step Semantics of MiniCorten: $t \mid \mu \rightsquigarrow t' \mid \mu'$)**.**

$$\text{SS-Assign}\ \frac{\star}{x = v \mid \mu \rightsquigarrow \texttt{unit} \mid \mu[x \mapsto v]}$$

$$\text{SS-Deref}\ \frac{\mu(x) = v}{*\texttt{ref}\ \ x \mid \mu \rightsquigarrow v \mid \mu}$$

$$\text{SS-Deref-Inner}\ \frac{t \mid \mu \rightsquigarrow t' \mid \mu'}{*t \mid \mu \rightsquigarrow *t' \mid \mu'}$$

$$\text{SS-Ref-Inner}\ \frac{t \mid \mu \rightsquigarrow t' \mid \mu'}{\texttt{ref}\ \ t \mid \mu \rightsquigarrow \texttt{ref}\ \ t' \mid \mu'}$$

$$\text{SS-Assign-Inner}\ \frac{t \mid \mu \rightsquigarrow t' \mid \mu'}{x = t \mid \mu \rightsquigarrow x = t' \mid \mu'}$$

5 The Refinement Type System

5.1 Definitions

P is the set of program variables used in rust program. Common names a, b, c

L is the set of logic variables used in refinement types. Common names: α, β

$\Gamma = (\mu, \Phi)$ is a tuple containing a function $\mu : P \to L$ mapping all program variables to their (current) logic variable and a set of formulas Φ over L. During execution of statements, the set increases monotonically

τ is a user defined type $\{\alpha : b \mid \varphi\}$. Where α is a logic variable from L, b is a base type from Rust (like `i32`) and φ is a formula over variables in L.

Abbreviations We write:

- Γ, c for $(\mu, \Phi \wedge c)$
- $\Gamma[a \mapsto \alpha]$ for $(\mu[a \mapsto \alpha], \Phi)$

5.2 Expression Typing: $\Gamma \vdash e : \tau \Rightarrow \Gamma'$

$$\textsc{Lit}\ \frac{l \text{ fresh} \qquad \text{base\_ty}(v) = b}{\Gamma \vdash v : \{l : b \mid l \doteq v\} \Rightarrow \Gamma}$$

$$\textsc{Var}\ \frac{\alpha \text{ fresh} \qquad \mu(x) = \beta}{\Gamma = (\mu, \Phi) \vdash x : \{\alpha : b \mid \beta \doteq \alpha\} \Rightarrow \Gamma}$$

$$\textsc{Var-Ref}\ \frac{\Gamma \vdash y : \tau \qquad \Gamma \vdash x : \{\beta : \&b \mid \beta \doteq \&y\}}{\Gamma \vdash *x : \tau \Rightarrow \Gamma}$$

$$\textsc{If}\ \frac{\begin{array}{c}\Gamma \vdash \mu(x) : \text{bool} \Rightarrow \Gamma \\ \Gamma, \mu(x) \doteq \texttt{true} \vdash e_t : \tau \Rightarrow \Gamma' \qquad \Gamma, \mu(x) \doteq \texttt{false} \vdash e_e : \tau \Rightarrow \Gamma'\end{array}}{\Gamma \vdash \text{if } x \text{ then } e_t \text{ else } e_e : \tau \Rightarrow \Gamma'}$$

$$\textsc{While}\ \frac{\Gamma_I, \mu(x) \doteq \texttt{true} \vdash s \Rightarrow \Gamma_I' \qquad \Gamma_I', \mu(x) \doteq \texttt{true} \preceq \Gamma_I}{\Gamma_I \vdash \texttt{while}\ \ x\{s\} \Rightarrow \Gamma_I, \mu(x) \doteq \texttt{false}}$$

$$\textsc{Seq}\ \frac{\Gamma \vdash s_1 : \tau_1 \Rightarrow \Gamma' \qquad \Gamma' \vdash \bar{s} : \tau \Rightarrow \Gamma''}{\Gamma \vdash s_1; \bar{s} : \tau \Rightarrow \Gamma''}$$

$$\textsc{Add}\ \frac{\gamma \text{ fresh} \qquad \mu(x_1) = \alpha \qquad \mu(x_2) = \beta}{\Gamma \vdash x_1 + x_2 : \{\gamma : b \mid \gamma \doteq \alpha + \beta\} \Rightarrow \Gamma}$$

$$\textsc{Decl}\ \frac{\Gamma \vdash e : \{\beta \mid \varphi\} \Rightarrow \Gamma'}{\Gamma \vdash \texttt{let}\ \ x = e : \texttt{()} \Rightarrow \Gamma'[x \mapsto \beta], \varphi}$$

$$\textsc{Assign}\ \frac{\Gamma \vdash e : \{\beta \mid \varphi\} \Rightarrow \Gamma'}{\Gamma \vdash x = e : \tau \Rightarrow \Gamma'[x \mapsto \beta], \varphi}$$

$$\textsc{Assign-Strong}\ \frac{\Gamma \vdash e : \{\beta \mid \varphi\} \Rightarrow \Gamma' \qquad \Gamma \vdash x : \{\alpha : \&b \mid \alpha \doteq \&y\}}{\Gamma \vdash *x = e : \tau \Rightarrow \Gamma'[y \mapsto \beta], \varphi}$$

$$\textsc{Assign-Weak}\ \frac{\begin{array}{c}\Gamma \vdash e : \tau \Rightarrow \Gamma' \\ \Gamma' \vdash x : \{\alpha : \&b \mid \alpha \doteq \&y_1 \vee \cdots \vee \alpha \doteq \&y_n\} \Rightarrow \Gamma' \\ \Gamma' \vdash \tau \preceq \{\beta_1 \mid \varphi_1\} \qquad \Gamma'[x \mapsto \beta_1], \varphi_1 \preceq \Gamma' \\ \cdots \\ \Gamma' \vdash \tau \preceq \{\beta_n \mid \varphi_n\} \qquad \Gamma'[x \mapsto \beta_n], \varphi_n \preceq \Gamma'\end{array}}{\Gamma \vdash *x = e : \tau \Rightarrow \Gamma'}$$

$$\textsc{Fn-Call}\ \frac{\begin{array}{c}(\mu, \alpha \doteq \mu(a) \wedge \cdots \wedge \varphi_\alpha \wedge \dots) \preceq (\mu, \Phi) \\ f : (\{\alpha \mid \varphi_\alpha\} \Rightarrow \{\alpha' \mid \varphi'_\alpha\}, \dots)\end{array}}{(\mu, \Phi) \vdash f(a, \dots) \Rightarrow (\mu[a \mapsto \alpha', \dots], \Phi \wedge \varphi'_\alpha \wedge \dots)}$$

$$\textsc{Intro-Sub}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash e\ \ \texttt{as}\ \ \tau' : \tau'}$$

5.3 Sub-Typing Rules: $\Gamma \vdash \tau \preceq \tau'$

$$\preceq\textsc{-Ty}\ \frac{\Phi \wedge \varphi'[\beta \triangleright \alpha] \vDash \varphi}{\Gamma = (\mu, \Phi) \vdash \{\alpha \mid \varphi\} \preceq \{\beta \mid \varphi'\}}$$

alternative (should be equivalent):

$$\preceq\text{-Ty-Alt}\ \frac{\Gamma[f \mapsto \alpha], \varphi \preceq \Gamma[f \mapsto \beta], \varphi' \qquad f \text{ fresh}}{\Gamma \vdash \{\alpha \mid \varphi\} \preceq \{\beta \mid \varphi'\}}$$

5.4 Sub-Context Rules: $\Gamma \preceq \Gamma'$

$$\preceq\text{-Ctx}\ \frac{\Phi'[\mu(\alpha) \triangleright \mu'(\alpha) \ \mid \alpha \in dom(\mu)] \vDash \Phi \qquad dom(\mu) \subseteq dom(\mu')}{(\Phi, \mu) \preceq (\Phi', \mu')}$$

In constrast to other refinement type systems, Corten allows types in the context to refer to one another. For example a type specifications `a : { α : i32 | α > β}, b: { β: i32 | β != α}` would be valid and result in the context $\Gamma = (\emptyset, \alpha \neq \beta \wedge \beta \geq \alpha)$. In the example 0 is a subtype of `a`'s type as long as $\texttt{b} \geq 1$.

This means that types may need to be changed atomically.

5.5 Soundness of the Type System

As usual, we consider the three properties (see Pierce [**pierce_types_2002**]) of a type system when assessing the correctness of MiniCorten:

1. Progress: If t is closed and well-typed, then t is a value or $t \rightsquigarrow t'$, where t' is a value.

   need to show for program or term?

2. Preservation: If $\Gamma \vdash e : \tau \Rightarrow \Gamma$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : \tau \Rightarrow \Gamma$

3. State Conformance: A state σ is conformant with respect to a typing context Γ (written as $\sigma : \Gamma$), iff $\bigwedge_{x \in \text{Var}} \mu(x) \doteq \sigma(x) \vDash \Gamma$

The state conformance rule differs from the usual rule in two ways. Firstly we consider the runtime value instead of the runtime type. Secondly there needs to be one set of variable assignments that satisfies all predicates in Γ.

Because MiniCorten is based on Rust, we will assume, that progress, as well as preservation for the base-types. This includes preservation of type- and ownership-safety. Thus it is sufficient to prove, that assuming these properties hold, preservation of the refinement types is holds.

5.6 Preservation

By induction over the typing derivations $\Gamma \vdash t : \tau \Rightarrow \Gamma'$.

LIT is trivial, Var and VAR-REF cannot occur because t is closed.

Case SEQ

6 Implementation

- Type Aliases - working without special compiler - compiler plugin - IDE - HIR - Location data - Additional features - basic inference - no ANF

7 Evaluation

7.1 Maximum using Path Conditions

7.2 Rephrasing builtins in terms of Refinement Types

panic

assert

7.3 Proof of the Gauss Summation Formula

8 Related Work

There currently does not exist an implementation of refinement types for Rust.

Relevant papers originate from two lines of work. Firstly additions to refinement types for mutability, asynchronous execution etc. and secondly other verification frameworks for Rust.

For example, Lanzinger [**lanzinger_property_2021**] successfully adapted refinement types to Java, which allows the user to check, that property types described by java annotations hold true throughout the program. At this point in time, specification and verification is limited to immutable (`final`) data.

Kloos et al. [**kloos_asynchronous_2015**] extended refinement types to mutable and asynchronous programs. The paper explores how changes to possibly aliased memory cells can be tracked throughout a OCaml program. For that purpose the types are extended by a set of requirements on memory objects, which track distinctness and refined types of these memory cells. In contrast to OCaml, Rust already guarantees that mutable memory is not aliased and in particular *all mutable memory locations must be accessible by a variable name in the current context*, which offers substantial advantages in terms of simplicity to specification and verification of Rust programs.

Refinement types are also used in other applications. For example Graf et al. [**graf_lower_2020**] use refinement types to check the exhaustiveness of pattern matching rules over complex (G)ADT types in Haskell. To check the exhaustiveness of patterns in Liquid Rust with ADTs may require similar approaches.

In terms of alternative verification approaches, Prusti[**astrauskas_leveraging_2019**] is notable, because of their work on formalizing the full Rust semantics, including `unsafe`. Prusti is a heavy-weight functional verification framework for Rust; based on separation logic.

Alternative verification approaches also exists: For example RustHorn[**matsushita__rusthorn__2020**] employs constrained horn clauses based verification to Rust. Particularity relevant for this thesis is the novel formalization for mutable references used in the paper. The authors stipulate that mutable references should be specified by a pre- and post-state from before a reference is borrowed to after it is returned.

The notion of Abstract Refinement Types that this thesis is based on is defined by Vazou et al. [**vazou__abstract__2013**]. The basic idea is to allow the programmer to ”refine” language types with predicates from a decidable logic. The type system has a notion of subtyping for refined types, where one type is a subtype of another if one predicate implies the other.

8.1 Comparison to Flux

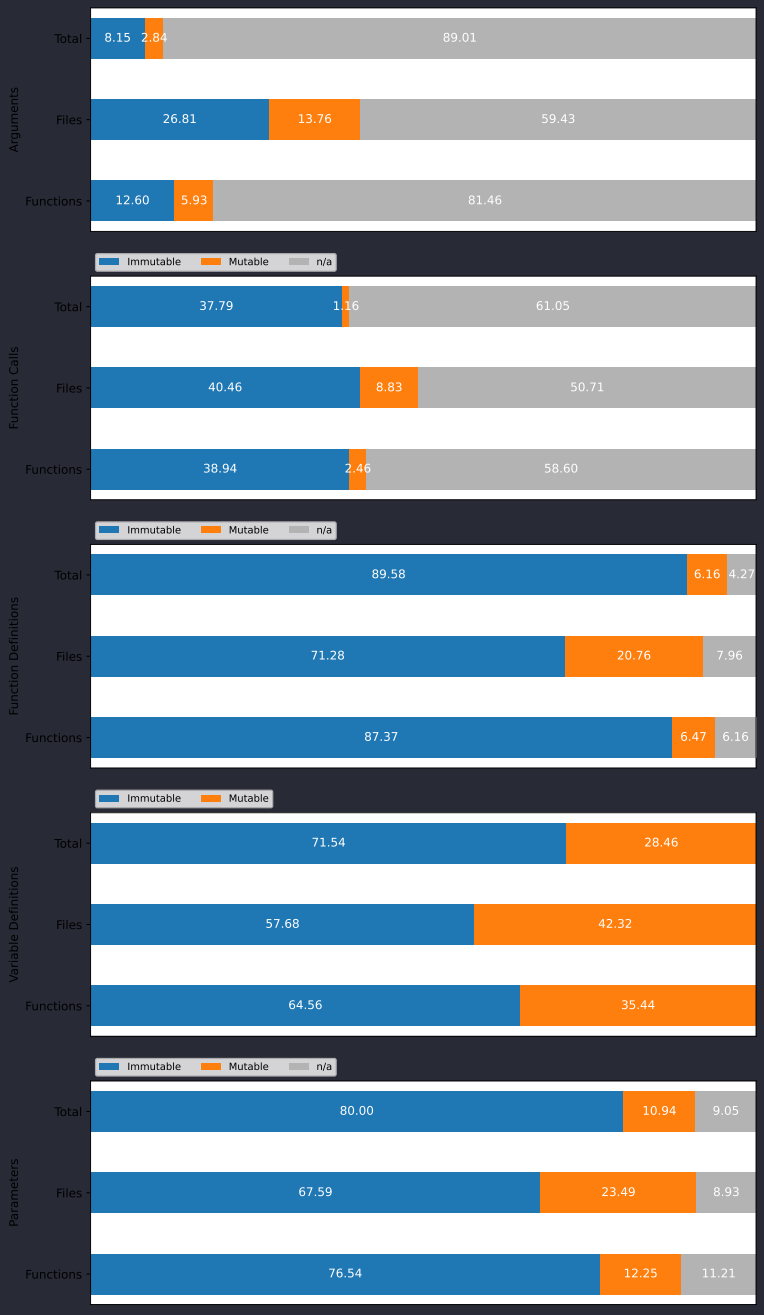

Figure 3: Ratio of Immutable to Mutable Versions of Different AST Items. Items are Counted by Unique Occurrences