Contents

1 Introduction **3**

2 Empirical Analysis of Use-Cases **5**
2.1 Unsafe Rust 6
2.2 Mutability 7
2.3 Conclusions 8

3 Foundations **11**
3.1 Rust 11
3.1.1 The Case for Rust as a Target Language 11
3.2 Refinement Types 13

4 The MiniCorten Language **15**
4.1 Features 15
4.1.1 Mutable References 15
4.1.2 Path Sensitivity 16
4.1.3 Strong Type Updates 16
4.1.4 Modularity 16
4.1.5 Mutual Reference 18
4.1.6 Atomic Updates 18
4.2 Syntax 18
4.3 Semantics 19

5 The Refinement Type System **21**
5.1 Definitions 21
5.2 Expression Typing: $\Gamma \vdash e : \tau \Rightarrow \Gamma'$ 22
5.3 Try: Statements $\Gamma \vdash s \Rightarrow \Gamma'$ 23
5.4 Function Declaration Type Checking 24
5.5 Sub-Typing Rules: $\Gamma \vdash \tau \preceq \tau'$ 24
5.6 Sub-Context Rules: $\Gamma \preceq \Gamma'$ 24
5.7 Soundness of the Type System 25
5.8 Extensions 26
5.8.1 Records / `structs` 26
5.8.2 Algebraic Data Types 26

5.8.3 Inference 27
5.8.4 Predicate Generics 27

6 Implementation **29**

7 Evaluation **31**
7.1 Maximum using Path Conditions 31
7.2 Rephrasing builtins in terms of Refinement Types 31
7.3 Complex Mutable References 31
7.4 Proof of the Gauss Summation Formula 31

8 Related Work **33**
8.1 Comparison to Flux 34

9 Conclusion & Future Work **35**

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.

Chapter 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.

The goal of the thesis is to show that Refinement Types can be idiomatically adapted to languages with unique mutable references. The type system presented in this thesis enables gradual adoption of lightweight verification methods in mutable languages.

The type system is sound and effective in the identified use-cases. A feasibility study on minimal examples of challenging use cases shows how useful the proposed verification system is.

Specifying or verifying complete Rust modules or the entire Rust language is not the goal of the thesis. In particular `unsafe` Rust will not be taken into account in specification nor implementation. Implementing Liquid Type inference in also not a goal of the thesis.

The accompanying implementation extends the Rust compiler, enables auto-

matic parsing of the refinement type language and automated type checking of a subset of Rust, as well as limited inference and error reporting. The thesis also gives a description of the syntax and semantics of the subset of Rust as well as the refinement type system.

The contributions of this thesis are:

1. Automatic empirical analysis of the usage of mutability and unsafe in Rust using syntactic information
2. Extension to the Rust type system to allow for refinement type specifications
3. Description and implementation of a type checker for the introduced type system
4. Evaluation of the type system on minimal benchmarks

The thesis if structured as follows: In chapter XXX an empirical analysis of `unsafe` and mutability uses is performed. Then chapter XXX will give an overview of the foundation the thesis build upon. Chapter XXX defines the subset of Rust, that will be the basis for the type system. Next chapter XXX explains the actual type system and verification, as well as justifying its correctness, followed by chapter XXX, which will provide more information about the implementation. The type system will then be tested in minimal benchmarks in chapter XXX. Chapter XXX reports on related work and alternative approaches. Finally chapter XXX concludes the thesis and gives an overview over possible future work.

Chapter 2

Empirical Analysis of Use-Cases

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 approachable 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.

There are two fundamental assumptions for the usefulness of the type system:

1. Uses of `unsafe` are rare
2. In the rare case, that `unsafe` is used, is is mostly used internally (not exposed to the callee)
3. Mutability is used pervasively in Rust.

To check these assumptions an analysis of existing Rust code was performed. As a basis for the analysis, the Rust package registry crates.io was used. It contains the source code for both Rust libraries as well as various applications written in Rust (e.g. ripgrep).

We analyzed all published Rust crates (Rust's version packages) on February 2nd 2022 on crates.io with at least 10 versions[1], which totals 11 882 crates, containing 228 263 files with a combined code-base size of over 64 million lines of Rust code (without comments and white space lines)[2].

[1] The limit of 10 versions is used to eliminate inactive and placeholder packages

[2] Calculated with `cloc`

The analysis parses these files and searches for certain AST patterns, which are subsequently extracted and saved. Thanks to using the tree-sitter parsing framework, the analysis framework can easily be extended to other queries and languages.

2.1 Unsafe Rust

Firstly we will be checking the hypothesis 1. 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 34.7% of crates, that use `unsafe`.

"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**] Our data seems to confirm this: 8 044 of the 11 882 crates (67.7%) did not use any unsafe.

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**] If this is the case for applications, it might disprove hypothesis hypothesis 2. When checking unsafely for our use case, it makes sense to further distinguish between libraries and executables crates: Libraries are intended 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[3] (see Figure 2.1). And secondly libraries are much more likely to use `unsafe` Rust. Table shows the result of our analysis. Where uses of `unsafe` are counted and grouped by crate type (see Table 2.1). The data in the table includes all crates except the outlier `windows-0.32.0`, which alone contains 233 608 uses of `unsafe`. Nearly 2/3 of all other library uses of `unsafe` combined. Looking at the distribution of unsafe uses in Figure 2.2, 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.

[3] Libraries and Executables are distinguished by checking if they contain `bin` or `lib` target or one of the corresponding files according to the cargo naming convention.

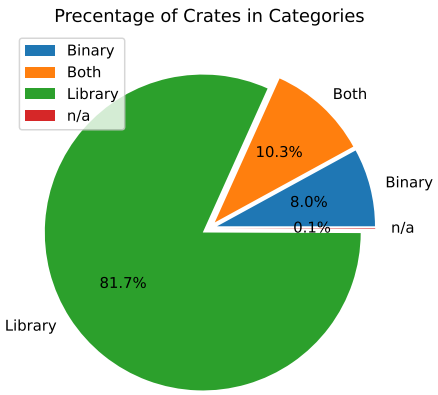

Figure 2.1: Percentage of Crates, that Contain Libraries, Executables or Both

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

Table 2.1: Number of Unsafe Uses by Crate Category

2.2 Mutability

Finally we will check hypothesis 3, which asserts, that mutable variables and references are used pervasively in Rust. This was checked by analysing the dataset for certain syntactical structures to infer mutability information about the following various AST items:

- **Local Variable Definitions** can be tracked with high confidence. They occur in function bodies and take the form: `let mut a = <expr>`
- **Parameters**, which are considered immutable if they are passed as immutable references or owned. They take the syntactic form: `mut a: i32` or `b: &mut i32`
- **Function Definitions**, which are considered immutable, if all parameters considered immutable. They take the syntactic form: `fn f(mut a: i32, b: &mut i32) { ... }`
- **Arguments** are parts of a function call and can be arbitrary expression, which makes the tracking hard.

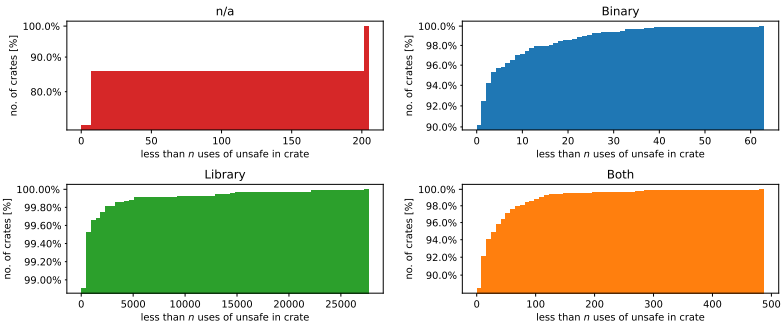

Figure 2.2: Cumulative, logarithmic histogram of the amount of `unsafe` uses in each category

- **Function Calls**, which are considered immutable, if all arguments are considered immutable.

A total of around 52 million of these items were found in the dataset.

Figure 2.3 shows the ratio of mutable to immutable items. For each syntactical category, the percentages are given relative to different objects:

1. Total: number of occurrences
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 mutability 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 mutability 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.

2.3 Conclusions

For this thesis the following conclusion can be drawn:

- Even though uses of `unsafe` are not rare, it is acceptable to ignore in favour of a simpler type system.
- Mutable variables are used very often and should therefore have very little associated specification effort.
- Mutable parameters are used less frequently justifying a higher specification effort, but their specification must still be possible.

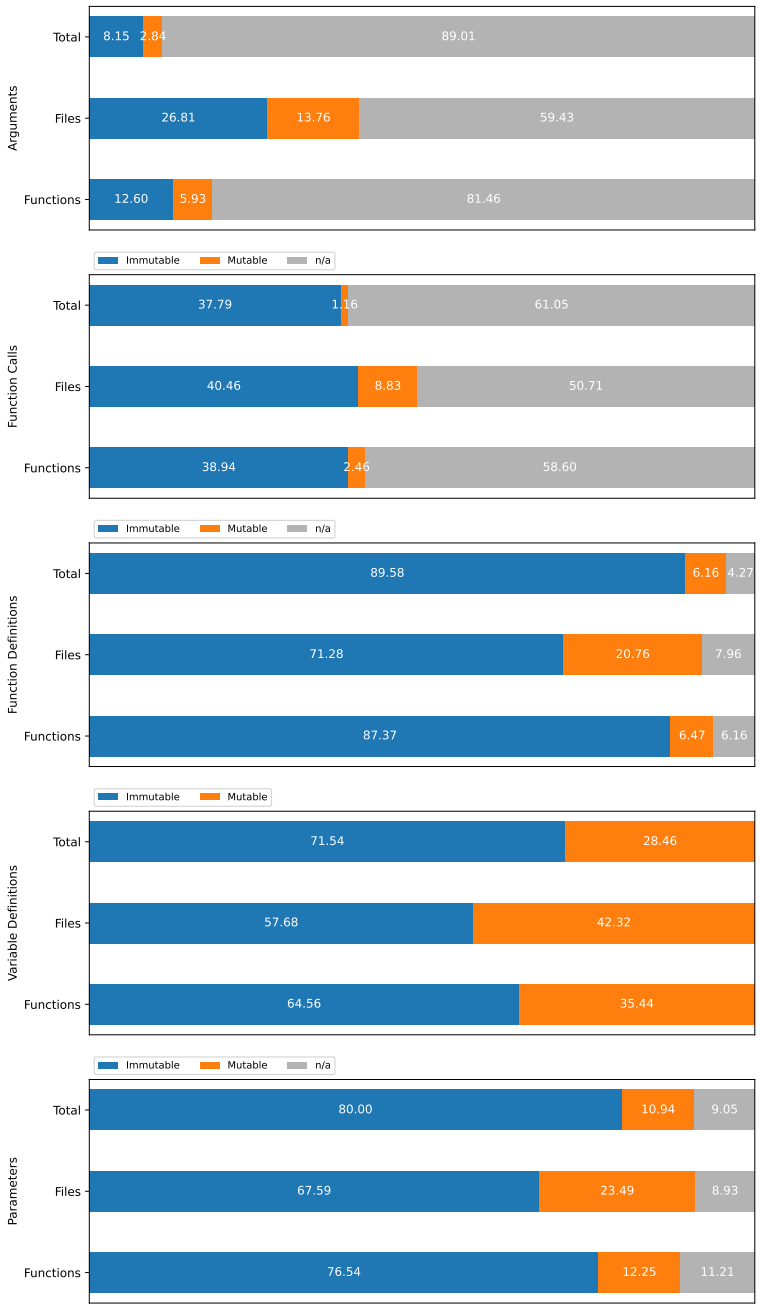

Figure 2.3: Ratio of Immutable to Mutable Versions of Different AST Items. Items are Counted by Unique Occurrences

Chapter 3

Foundations

3.1 Rust

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.

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.

3.1.1 The Case for Rust as a Target Language

Rust justifies a new approach to mutable verification, because it is possible to take advantage of guarantees provided by (safe) Rust's ownership system.

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

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 2).

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

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 Even though Rust simplifies 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 1 shows how `cell`'s type is influenced by changes to `r`. This does not mean, that the ownership rule *mutability XOR aliasing* are broken: Even though both `cell` and `r` are mutable, but only one is active.

```
let mut cell = 2;
let r = &mut cell;
r+= 1;
cell += 1;
assert!(cell, 3)
```

Listing 1: Example of an apparent violation of ownership rules

3.2 Refinement Types

The type system, that this thesis will adapt, is based on the Refinement Type system described in [**vazou_abstract_2013**] and [**rondon_liquid_2008**]. The central idea of Refinement Types is adding predicates to a language's type: For example, a type `i32` might be refined with the predicate $v > 0$. In terms of semantics this means, that any inhabitant of that type must also satisfy the predicate.

- Dependent Type but automatic - limited expressiveness - SMT solvable decidable logic

The notion of refinements in embedded into the base language using subtyping: A refined type τ is a subtype of τ' if τ's predicate implies the predicate of τ' (in the current typing context Γ)

$$\text{SMT-Valid}([\![\Gamma]\!])$$

Chapter 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 Features

This subsection will explain some of the key features of the Corten type system. Corten is directly embedded in Rust using two macros. Firstly the macro `ty!` can be used in place of a Rust type and adds a predicate to the Rust type, that any inhabitant of that type must satisfy. For example, the type `ty!{ v : i32 | v >= 0}` stipulates, that a value of Rust type `i32` is positive. The second macro is `relax_ctx!{ ... }` which will be explained in subsection 4.1.5.

4.1.1 Mutable References

The key difference to Refinement Type Systems for functional languages to Rust it the pervasive use of mutation. Therefore the most important feature of Corten it the support for mutation and also mutable references.

The problem with references is the possibility of interdependencies: If one variable is changed, it might effect wether another variable is typed correctly. In listing 2 the return value `a` is effected by changes done to a different variable `a`. Conservative approximation requires, that all possible effects must be tracked.

Corten can accurately track references by phrasing them as predicates. For example the expression `&mut a` will be typed as `ty!{ _1 : &mut b | _1 == &a }` (where `b` is the Rust type of `a`). This is sensible, because in Rust's ownership system, `a` must be the unique owner of the memory belonging to `a`. A

more involved example is given in the evaluation 7.2

Corten has two ways of dealing with assignment to mutable references: If the reference destination is known, the destination's type will be updated with the assigned values type (i.e. strong update). If there are multiple possible reference destinations, Corten will require the assigned value to satisfy the predicates of all possible destinations (i.e. weak updates). This is a standard approach (For example [**kloos_asynchronous_2015**]), but more precise in Rust: The set of possible destinations only grows, when it depends on the execution of an optional control flow path. This means most of the time, strong updates are possible and weak updates are only needed if the reference destination is actually dynamic (i.e. dependent on the execution) [1]

- importance of tracing return of mutable references

```
fn client() -> ty!{ v: i32 | v == 4 } {
    let a = 2; let b = &mut a;
    *b = 0; // changes a's value
    let c = &mut b;
    **c = 4; // changes a's value
    a
}
```

Listing 2: Example demonstrating interdependencies between mutable references

4.1.2 Path Sensitivity

Firstly, the type system is path sensitive, meaning that the type system is aware of necessary XXX that need to be passed for an expression to be evaluated. For example listing 3 shows a function computing the maximum of its inputs. In the then branch, a is only a maximum of a and b, because the condition $a > b$ implies it. Corten will symbolically evaluate the condition and store it in its typing context.

- inference - also with mutations

4.1.3 Strong Type Updates

4.1.4 Modularity

For a verification system to be scalable, it needs to be able modularize a proof. Corten can propagate type information across function calls and by taking advantage of Rust's ownership system, it can do so very accurately. Listing 5

[1] It would also be possible to encode the path condition in the reference type, but this was decided against for the stated goal of simplicity for the user

```
fn max(a : ty!{ av: i32 }, b: ty!{ bv: i32 })
    -> ty!{ v: i32 | v >= av && v >= bv} {
    if a > b {
      a as ty!{ x : i32 | x >= av && x >= bv }
    } else {
      b
    }
}
```

Listing 3: Function computing the maximum of its inputs; guaranteeing that the returned value is larger than its inputs

```
fn strong_updates(b: ty!{ bv: i32 | bv > 0}) -> ty!{ v: i32 | v > 2} {
    let mut a = 2;
    a = a + b;
    a
}
```

Listing 4: Example of changes to `a`'s value effecting its type

shows, how an incrementing function `inc` can be specified: `inc` signature stipulates, that it can be called with any `i32` (given the name `a1`) and will return a value `a2`, which equals `a1 + 1`. Only the signature of `inc` is necessary for the type checking of `client`.

Notice, that all of the type information about `y` is preserved when `inc(x)` is called. There are no further annotations needed to type check this program. This is possible, because in safe Rust, any (externally observable) mutation done by a function must be part of the function signature. Corten expands on this, by enabling the user specify exactly how a referenced value is mutated.

```
fn inc(a: &mut ty!{ a1: i32 => a2 | a2 == a1 + 1 }) {
  *a = *a + 1;
}
fn client(mut x: ty!{ xv: i32 | xv > 2 }) -> ty!{ v: i32 | v > 7 } {
  let mut y = 2;
  inc(&mut x); inc(&mut x);
  inc(&mut y)
  x + y
}
```

Listing 5: Example showing how Corten allows for accurate type checking in the presence of function calls

4.1.5 Mutual Reference

why?

Complex mutation patterns often result in complex interdependencies. We deem it necessary to allow different types to refer to each other.

```
let sum = 0;
let i = 0;
```

Listing 6: Function computing the maximum of its inputs; guaranteeing that the returned value is larger than its inputs

4.1.6 Atomic Updates

- relax ctx - non-atomic might not be sufficient

4.2 Syntax

Call it CortenRust (after Weathering Steel)

What does the abbrev stand for? XXX Normal Form

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.

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*

4.3 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 4.3.1 (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 4.3.2 (Small-Step Semantics of MiniCorten: $\langle e \mid \sigma\rangle \rightsquigarrow \langle e' \mid \sigma'\rangle$)**.**

$$\text{SS-Assign}\ \frac{\star}{x = v \mid \sigma \rightsquigarrow \texttt{unit} \mid \sigma[x \mapsto v]} \quad \text{SS-Assign-Inner}\ \frac{t \mid \sigma \rightsquigarrow t' \mid \sigma'}{x = t \mid \sigma \rightsquigarrow x = t' \mid \sigma'}$$

$$\text{SS-Decl}\ \frac{\star}{\langle \texttt{let}\ \ x = v \mid \sigma\rangle \rightsquigarrow \langle unit \mid \sigma[x \mapsto v]\rangle}$$

$$\text{SS-Deref}\ \frac{\sigma(x) = v}{*\texttt{ref}\ \ x \mid \sigma \rightsquigarrow v \mid \sigma} \quad \text{SS-Deref-Inner}\ \frac{t \mid \sigma \rightsquigarrow t' \mid \sigma'}{*t \mid \sigma \rightsquigarrow *t' \mid \sigma'}$$

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

$$\text{SS-Seq-Inner}\ \frac{e_1 \mid \sigma \rightsquigarrow e_1' \mid \sigma'}{e_1; e_2 \mid \sigma \rightsquigarrow e_1', e_2 \mid \sigma'} \quad \text{SS-Seq-N}\ \frac{\star}{\texttt{unit}; e_2 \mid \sigma \rightsquigarrow e_2 \mid \sigma'}$$

$$\text{SS-IF-T}\ \frac{\sigma(x) = \texttt{true}}{\langle \texttt{if}\ \ x\ \{e_t\}\ \texttt{else}\ \{e_e\} \mid \sigma\rangle \rightsquigarrow \langle e_t \mid \sigma\rangle} \quad \text{SS-IF-F}\ \frac{\sigma(x) = \texttt{false}}{\langle \texttt{if}\ \ x\ \{e_t\}\ \texttt{else}\ \{e_e\} \mid \sigma\rangle \rightsquigarrow \langle e_e \mid \sigma\rangle}$$

$$\text{SS-While}\ \frac{\star}{\langle \texttt{while}\ \ e_c\ \{\ e_b\} \mid \sigma\rangle \rightsquigarrow \langle \texttt{if}\ \ e_c\{e_b; \texttt{while}\ \ e_c\{e_b\}\}\ \texttt{else}\ \{\texttt{unit}\} \mid \sigma\rangle}$$

Chapter 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: $\Gamma \vdash e : \tau \Rightarrow \Gamma'$

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

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

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

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

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

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

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

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

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

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

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

$$\textsc{Fn-Call}\ \frac{\begin{array}{c}\text{subst} = \ldots \qquad (\mu[subst], \alpha \doteq \mu(a) \wedge \cdots \wedge \varphi_\alpha \wedge \ldots) \preceq (\mu, \Phi) \\ f : (\{\alpha \mid \varphi_\alpha\} \Rightarrow \{\alpha' \mid \varphi'_\alpha\}, \ldots) \to \tau\end{array}}{(\mu, \Phi) \vdash f(a, \ldots, \&mutb, \ldots) : \tau \Rightarrow (\mu[a \mapsto \alpha', \ldots, subst^{-1}], \Phi \wedge \varphi'_\alpha \wedge \ldots)}$$

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

5.3 Try: Statements $\Gamma \vdash s \Rightarrow \Gamma'$

$$\textsc{If}\ \frac{\Gamma \vdash \mu(x) : \text{bool} \Rightarrow \Gamma \qquad \Gamma, \mu(x) \doteq \texttt{true} \vdash s_t \Rightarrow \Gamma' \qquad \Gamma, \mu(x) \doteq \texttt{false} \vdash s_e \Rightarrow \Gamma'}{\Gamma \vdash \text{if } x \text{ then } s_t \text{ else } s_e \Rightarrow \Gamma'}$$

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

$$\textsc{Seq}\ \frac{\Gamma \vdash s_1 \Rightarrow \Gamma' \qquad \Gamma' \vdash s_2 \Rightarrow \Gamma''}{\Gamma \vdash s_1; s_2 \Rightarrow \Gamma''}$$

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

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

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

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

$$\textsc{Fn-Call}\ \frac{\begin{array}{c}\text{subst} = \dots \qquad (\mu[\mathit{subst}], \alpha \doteq \mu(a) \wedge \dots \wedge \varphi_\alpha \wedge \dots) \preceq (\mu, \Phi) \\ f : (\{\alpha \mid \varphi_\alpha\} \Rightarrow \{\alpha' \mid \varphi'_\alpha\}, \dots) \to \tau\end{array}}{(\mu, \Phi) \vdash \texttt{let res} = f(a, \dots, \texttt{\&mut}b, \dots) \Rightarrow (\mu[a \mapsto \alpha', \dots, \mathit{subst}^{-1}], \Phi \wedge \varphi'_\alpha \wedge \dots)}$$

Expressions: $\Gamma \vdash e : \tau$

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

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

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

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

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

5.4 Function Declaration Type Checking

Without loss of generality, we assume arguments are ordered by their type: First immutable / owner arguments and then mutable arguments.

When handling mutable references in parameters some subtleties need to be considered. The function can change both the referenced *value* as well as the reference *location*. To describe the referenced value is normally done using the $\{\alpha \mid \alpha \doteq \&b\}$ syntax. The question arises: If α was the logic variable for a parameter, what should be the analog for b? In contrast to local variables, there is no variable representing the referenced value for parameters. As seen in section 4.1, using the dereference operator would come with a lot of complications. Instead we introduce arg_n^i, a special variable denoting the initial abstract value (i.e. stack location), that the mutable reference of argument n points to. i denotes the level of nesting: For the nth parameter with the Rust type `&mut &mut i32` we would generate arg_n^1 and arg_n^2.

$$\textsc{Fn-Decl}\ \frac{(\{a_1 \mapsto \alpha_1, b_1 \mapsto \delta_1\}, \varphi_1^\alpha \wedge \dots \wedge \delta_1 = arg_1^1 \wedge \varphi_1^\beta \wedge \dots) \vdash \bar{s} \Rightarrow \Gamma' \quad \Gamma' \vdash s_{res} : \tau_{res} \Rightarrow \Gamma'' \quad \Gamma'' \vdash \tau_{res} \preceq \tau \quad \Gamma'' \preceq (\{\beta_1 \mapsto \gamma_1\}, \varphi_1^\gamma)}{\texttt{fn}\ f(a_1 : \{\alpha_1 \mid \varphi_1^\alpha\}, \dots, b_1 : \{\beta_1 \mid \varphi_1^\beta \Rightarrow \gamma_1 \mid \varphi_1^\gamma\}) \to \tau\{\bar{s}; s_{res}\}}$$

5.5 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):

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

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

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

In contrast 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, the value 0 i a valid inhabitant of `a`'s type as long as `b` ≥ 1.

This means that types may need to be changed atomically.

5.7 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:

Definition 5.7.1. State Conformance: A state σ is conformant with respect to a typing context $\Gamma = (\mu, \varphi)$ (written as $\sigma : \Gamma$), iff:

$$\vDash \left(\bigwedge_{x \in \mathrm{Var}} \mu(x) \doteq \sigma(x) \right) \to \varphi$$

I.e. the a execution's value assignments imply the type refinements

Definition 5.7.2. Progress: If t is closed and well-typed, then t is a value or $t \rightsquigarrow t'$, where t' is a value.

Definition 5.7.3. Preservation: If $\Gamma \vdash e : \tau \Rightarrow \Gamma$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : \tau \Rightarrow \Gamma$

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.

Lemma 5.7.4. *Preservation of State Conformance: If* $\Gamma \vdash e : \tau \Rightarrow \Gamma''$, $\sigma : \Gamma$ *and* $\langle t \mid \sigma \rangle \rightsquigarrow \langle t' \mid \sigma' \rangle$, *then* $\sigma' : \Gamma'$ *and* $\Gamma' \vdash e' : \tau' \Rightarrow \Gamma''$

Proof. Rule Induction over $\langle t \mid \sigma \rangle \rightsquigarrow \langle t' \mid \sigma' \rangle$

- Case SS-Assign: $\sigma' = \sigma[x \mapsto v]$. With rule inversion Assign: $e' = \texttt{unit}$ and post state: $\Gamma'' = \Gamma'[x \mapsto \beta], \varphi$ with $\Gamma \vdash e : \{\beta \mid \varphi\}$.

  e is a value, because of the requirement in SS-Assign. By rule inversion Lit over assumption $\Gamma \vdash v : \{\beta \mid \varphi\} \Rightarrow \Gamma$: $\varphi = (\beta \doteq v)$

  $\mu'(x) = \beta$, $\sigma'(x) = v$ and induction hypothesis $\sigma : \Gamma$. Since $\mu'(x) \doteq \varphi(x) \to \varphi$ is trivially valid. Monotonicity of State Conformance gives us the goal.

  Because e' is `unit`, the second proof obligation is trivially true, because $\Gamma' = \Gamma''$

- Case SS-Assign-Inner: Induction hypothesis: $\langle e \mid \sigma \rangle \rightsquigarrow \langle e' \mid \sigma' \rangle$, $\sigma' : \Gamma'$, $\Gamma \vdash e' : \tau' \Rightarrow \Gamma''$

  Induction hypothesis is equal to proof obligation.

- Case SS-Decl: $\sigma' = \sigma[x \mapsto v]$, $e' = \texttt{unit}$.

  $\sigma' : \Gamma'$ because of monotonicity; $\Gamma' \vdash \texttt{unit} : \texttt{Unit} \Rightarrow \Gamma'$ by SS semantics true

- Case Deref $\sigma(x) = v$, $e' = v$, $\sigma' = \sigma$
  For $\Gamma' = \Gamma$, $\sigma : \Gamma'$ true by assumption
  Show: $\Gamma \vdash v : \tau' \Rightarrow \Gamma$. By rule inversion Lit, $\tau' = \{l \mid l \doteq v\}$

□

sensible name?

Lemma 5.7.5. *State Conformance is monotone: If $\sigma : \Gamma$, $(\beta = v) \to \varphi$ then $\sigma[x \mapsto v] : \Gamma[x \mapsto \beta] \wedge \varphi$*

Proof.

$$\vDash \left(\bigwedge_{x \in \mathrm{Var}} \mu(x) \doteq \sigma(x) \right) \to \varphi$$

...? □

5.8 Extensions

To limit the complexity a few features are not defined as part of the Corten or the implementation. Even though these extensions are not part of the scope of the thesis, these extensions were taken into consideration when designing the type system. To show, that the type system is capable of handling the additional challenges, we will shortly describe how these extensions are planned to be added to the type system.

move to future work?

5.8.1 Records / `structs`

Firstly, a key part of realistic programs are data structures that comprise multiple basic data types. In Rust these are called `structs` and work similar to records or product types in functional languages.

Once again, we can take advantage of Rust ownership system: Any part of a struct (even nested fields) can only belong to one variable. This means, that the proposed system for handling mutable references extends seamlessly to structs: The variable mapping in Γ is generalized: $\mu : \mathrm{Path} \to \mathrm{LVar}$. The relevant typing rules will work without major changes:

$$\textsc{Var}\ \frac{\mu(p.x.y.z) = \alpha}{\Gamma \vdash p.x.y.z : \{\beta \mid \beta \doteq \alpha\} \Rightarrow \Gamma}$$

$$\textsc{Assign-Strong}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash p.x.y.z = e : \texttt{unit} \Rightarrow \Gamma[p.x.z.y \mapsto \tau]}$$

5.8.2 Algebraic Data Types

With records added, the only thing missing for support of algebraic data types are sum types. Sum types allow the programmer to express, that an inhabitant

of a type may be one variant of a set of multiple fixed options: For example the result of a fallible computation may either be `Ok(V)` meaning a successful computation with the result `V` or a failure `Err(E)` with a description of the error `E`. In rust these sum types are called `enum`s.

Sum types influence the type system in two key ways:

Firstly, the specification of its values. Suppose a programmer would like an authentication function signature to state, that the function returns an `Err` code "403" if the password was incorrect. This requires the type language to assert what variant is expected as well as access to its fields (if the variant is known).

Secondly, path sensitivity needs to be extended to cover `match` expressions (called `case` is most functional programming languages). `match` expression allow the programmer to branch depending on the variant of a value.

5.8.3 Inference

While the current implementation is able to type expressions without explicit type annotations, solving a set of type constraints is not implemented. Rondon et al. [**rondon_liquid_2008**] describe a mechanism to infer complex refined types by combining known predicates from the context. This approach should be adaptable to Corten to reduce the amount of needed type annotations even further.

5.8.4 Predicate Generics

Vazou et al. [**vazou_abstract_2013**] found, that the expressiveness of refinement types can still be expanded without leaving a decidable fragment by adding uninterpreted functions to the logic. In the type system, these uninterpreted functions represent "abstract predicates". At the definition site, abstract predicates can not be inspected and restricted, but the caller can instantiate then with a concrete predicate.

Chapter 6

Implementation

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

Chapter 7

Evaluation

7.1 Maximum using Path Conditions

7.2 Rephrasing builtins in terms of Refinement Types

panic

assert

7.3 Complex Mutable References

7.4 Proof of the Gauss Summation Formula

Chapter 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

Chapter 9

Conclusion & Future Work