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 Refinement Types 11
- 3.2 Rust 12
  - 3.2.1 The Case for Rust as a Target Language 12
- 3.3 Rustc 14

4 The MiniCorten Language **17**
- 4.1 Syntax 17
- 4.2 Semantics 18

5 The Refinement Type System **21**
- 5.1 Features 21
  - 5.1.1 Decidable, Conservative Subtyping 21
  - 5.1.2 Mutable Values 21
  - 5.1.3 Mutable References 22
  - 5.1.4 Path Sensitivity 24
  - 5.1.5 Strong Type Updates 24
  - 5.1.6 Modularity 24
  - 5.1.7 Mutual Reference 25
  - 5.1.8 Atomic Updates 25
- 5.2 Definitions 25
- 5.3 Statement Type Checking $\Gamma \vdash s \Rightarrow \Gamma'$ 26
- 5.4 Expressions Typing: $\Gamma \vdash e : \tau$ 27
- 5.5 Function Declaration Type Checking 27
- 5.6 Sub-Typing Rules: $\Gamma \vdash \tau \preceq \tau'$ 27
- 5.7 Sub-Context Rules: $\Gamma \preceq \Gamma'$ 28
- 5.8 Soundness of the Type System 28

5.9 Extensions 30
5.9.1 Records / `structs` 30
5.9.2 Algebraic Data Types 30
5.9.3 Inference 31
5.9.4 Predicate Generics 31

6 Implementation: CortenC 33

7 Evaluation 39
7.1 Maximum using Path Conditions 39
7.2 Recursion: Fibonacci-Numbers 40
7.3 Loop Invariants: Proof of the Gauss Summation Formula 41
7.4 Complex Mutable References 42
7.5 Rephrasing built-ins in terms of Refinement Types 42
7.6 Extendability with Other tools 43

8 Related Work 45
8.0.1 Alternative Approaches for Handling Mutability 46
8.1 Comparison to Flux 46

9 Conclusion & Future Work 47

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 automatic 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 questions, that analysing existing rust code should answer:

1. How rare are uses of `unsafe`
2. How much effort is acceptable when specifying `unsafe` code?
3. How common are mutable variables and references 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 answering the question of `unsafe` usage in 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 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**]

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.

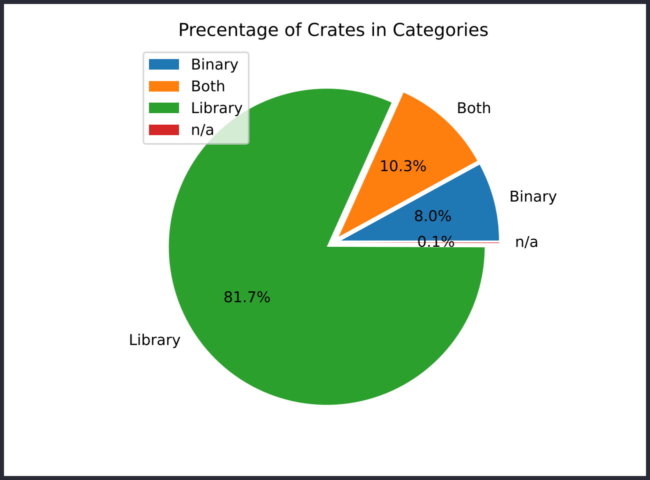

Figure 2.1: Percentage of Crates, that Contain Libraries, Executables or Both

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 analyse how common mutable variables and references are used in Rust. The frequency of usage will inform the acceptable level of specification effort.

To analyse the dataset for usage pattern, we search 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) { ... }`

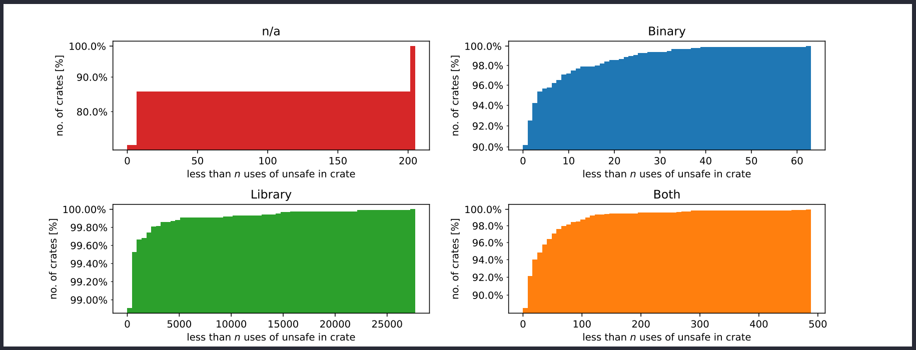

Figure 2.2: Cumulative, logarithmic histogram of the amount of `unsafe` uses in each category

- **Arguments** are parts of a function call and can be arbitrary expression, which makes the tracking hard.
- **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.

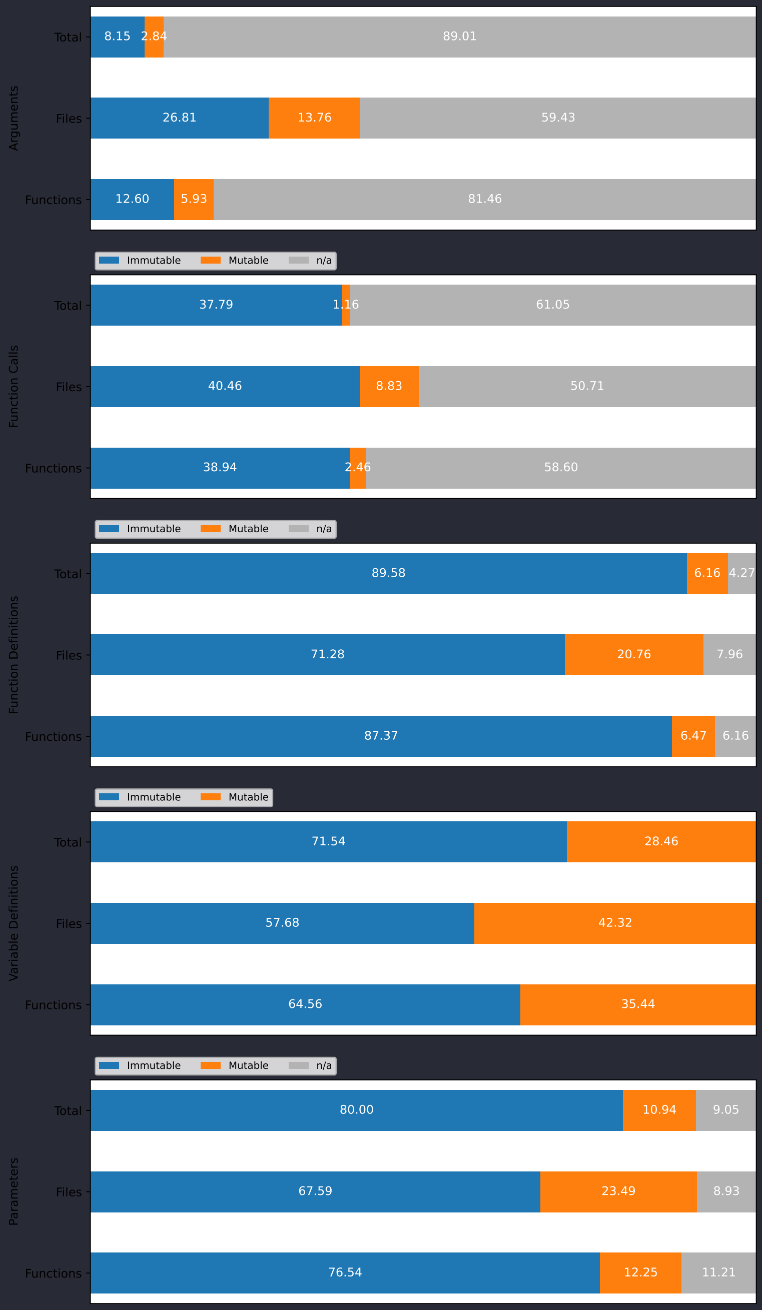

Figure 2.3: Ratio of Immutable to Mutable Versions of Different AST Items. Items are Counted by Unique Occurrences

Chapter 3

Foundations

This chapter will introduce

3.1 Refinement Types

The type system, that this thesis will adapt, is based on the Refinement Type system described by Vazou et al. [**vazou__abstract__2013**] and Rondon et al. [**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 $v > 0$. The refined type is written as $\{v : \text{i32} \mid v > 0\}$. In terms of semantics this means, that any inhabitant of that type must also satisfy the predicate.

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). As is common with subtyping systems, the subtype can then be used in the place where the super type is expected. The following rule shows how subtyping is handled by Liquid types [**rondon__liquid__2008**]:

$$\frac{\text{SMT-Valid}(\llbracket\Gamma\rrbracket \land \llbracket e_1\rrbracket \Rightarrow \llbracket e_2\rrbracket)}{\Gamma \vdash \{v : b \mid e_1\} \preceq \{v : b \mid e_2\}}$$

Refinement Types are related to dependent types, distinguished by the limited expressiveness of the type. Refinement types trade off expressiveness for automatic type checking. Allowed formulas are chosen from a SMT decidable logic. Rondon et al. [**rondon__liquid__2008**] chose the EUFA as the logic fragment, meaning a propositionaly logic with the theories of equality, uninterpreted functions and linear arithmetic.

3.2 Rust

The Rust programming language, first conceived of by Graydon Hoare, is multi-paradigm programming language, that draws inspiration from functional and imperative programming languages. The core forms a set of features usually found in functional programming languages: Functions, closures, algebraic data types, type classes (called traits in Rust) and pattern matching, immutability are all common place in Rust. But what is unusual is, that Rust extends this base with optional mutability of data and references. The Rust programming language is design to closed control the amount of mutability and what it can effect. To this end, Rust incorporates the notion of affine types into its type system (called ownership system in Rust). Rust took inspiration from a research language called Cyclone introduced by Grossman et al. [**grossman__region-based__nodate**]. Among other things, the ownership system is used to handle memory management, safe concurrency and simplifies reasoning about program behaviour. The next paragraph will explain how the ownership system works.

Aliasing XOR Mutabillity The key behind Rusts ownership system is, that for every variable at every point during the execution, that variable can 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.

3.2.1 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.

Minimal subtyping In contrast to object oriented languages, Rust has no user-extendable notion of subtyping. Rust does have a notion of subtyping for its permission system, which serves the purpose of coercing mutable references `&mut T` to immutable references `& T` etc. where needed. Because of the limited, well-known scope, these subtyping relations create no ambiguity with a refinement type system.

No overloaded functions

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

3.3 Rustc

Since the implementation of the proposed interacts with Rustc, the reference implementation of a Rust compiler, this section will summarize the relevant parts of Rustc mainly based on the Rustc dev guide [**noauthor_overview_nodate**]

Rustc offers an APIs for writing compiler plugins, which can access the intermediate languages of Rust at different stages during the compilation and allows the plugin to emit warnings and errors. At a high level, rustc is built with stages, that incrementally lower the source code to an executable binary. Rustc has multiple stages and associated data structures. Relevant for the thesis are the first few stages: Firstly the AST, which is a representation of the source code after lexing and parsing,

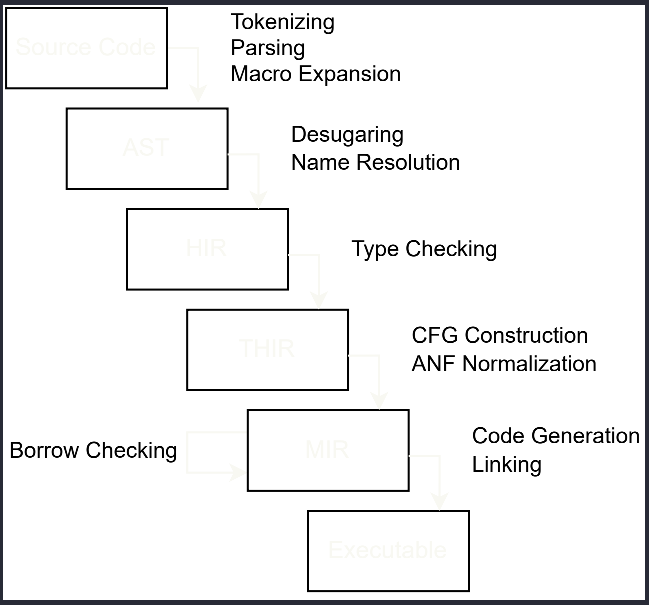

Figure 3.1: Diagram providing an overview of the Rustc Compiler Stages

Figure 3.1 shows the the main compiler stages of Rustc and the associated data structure. The first data structure, that is extracted from the source code is the AST (short for abstract syntax tree), which is the result of parsing and expanding macros. Based on the AST, the high-level intermediate representation (short HIR) is created by resolving the names and desugaring some language constructs. The HIR contains accurate source labels, which map all nodes in the HIR to their source code locations. The HIR is then used as the basis for type checking returning the typed high-level intermediate representation (short THIR), which annotates every node with its type. THIR also lowers some additional constructs like automatic dereferencing and overloading and more. In contrast to the previous representations, ”[t]he THIR only represents bodies, i.e. «executable code»; this includes function bodies [..]. Consequently, the THIR has no representation for items like `struct`s or `trait`s.” [**noauthor_thir_nodate**] THIR will then be used to create the mid-level IR by drastically simplifying the language: The tree structure with complex expression of the THIR is turned in to a control-flow graph (short CFG) with basic blocks containing simple (i.e. non-nested) expressions.

Finally Rustc uses external tools like llvm to generate executable files and link them.

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 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

There are two namespaces for identifiers: $x \in \mathrm{PVar}$ for variables of program variables and $\alpha \in \mathrm{LVar}$ for variable in the refinement predicates. The syntax is defined by the following BNF-Grammar:

program	::=	*func_decl* `*`	*function declarations*
`func_decl`	::=	*ident*`(` *param* `*` `)` `->` *ty* `{` *stmt* `}`	
param	::=	x `:` τ	
s: `stmt`	::=	e	*expression*
	$\mid$	s_1`;` s_2	*sequence*
	$\mid$	`let mut?` x `=` e	*declaration*
	$\mid$	x `=` e	*assignment*
	$\mid$	`let` x_v `=` $x_f(x_1,,x_n)$	*function call*
	$\mid$	`while` (x) `{` s `}`	*while loop*
	$\mid$	`if` e_c `{` s_t `} else {` s_e `}`	*if statement*
	$\mid$	`relax_ctx!{` φ`*` `;` `(`x `:` τ`)*` `}`	*context relaxation*
e `: expression`	::=	*ident*	*variable reference*
	$\mid$	*lit*	*constant*
	$\mid$	*expr* `+` *expr*	*addition*
	$\mid$	`*` *ident*	*dereference*
	$\mid$	`&` *ident*	*immutable reference*
	$\mid$	`&mut` *ident*	*mutable reference*
	$\mid$	*expr* `as` *ty*	*type relaxation*
τ `: type`	::=	`ty!{` *logic_ident* `:` *base_ty* `|` *pred* `}`	*refinement type*
φ `: pred`	::=	*ref_pred*	*predicate for a reference type*
	$\mid$	*value_pred*	*predicate for a value type*
ref_pred	::=	*logic_ident* `=` `&` *ident*	
	$\mid$	*ref_pred* `||` *ref_pred*	*mutable reference*
value_pred	::=	*logic_ident*	*variable*
	$\mid$	*pred* `&&` *pred*	*conjunction*
	$\mid$	*pred* `||` *pred*	*disjunction*
	$\mid$	`!` *pred*	*negation*
b: `base_ty`	::=	`i32`	*integer*
	$\mid$	`unit`	*unit type*
	$\mid$	`bool`	*boolean*
	$\mid$	`&` *base_ty*	*immutable reference*
	$\mid$	`&mut` *base_ty*	*mutable reference*
v: `lit`	::=	`0,1,...,n`	*integer*
	$\mid$	`true`	*boolean true*
	$\mid$	`false`	*boolean false*
	$\mid$	`()`	*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 $\texttt{ref}(x)$ as a value itself. The following definitions show the new execution rules.

Definition 4.2.1 (Execution-State)**.** The execution state is a partial function from program variables to values: $\sigma : \text{PVar} \rightharpoonup \text{Value}$.

Definition 4.2.2 (Evaluation of Expressions: $[\![e]\!]\sigma$)**.**

$$\begin{aligned}
[\![v]\!]\sigma &= v \\
[\![x]\!]\sigma &= \sigma(x) \\
[\![e_1 + e_2]\!]\sigma &= [\![e_1]\!]\sigma + [\![e_2]\!]\sigma \\
[\![*x]\!]\sigma &= \sigma(y) \qquad \text{if } \sigma(x) = \texttt{ref}(y) \\
[\![\texttt{\&mut } x]\!]\sigma &= \texttt{ref}(x) \\
[\![e \texttt{ as } \tau]\!]\sigma &= [\![e]\!]
\end{aligned}$$

Definition 4.2.3 (Declaration Environment)**.** The environment of function declarations is constant and globally known. It is defined as a partial function $\Sigma : \text{Fn-Name} \rightharpoonup (n, s)$ where n is the number of arguments and s is the body of the function

Definition 4.2.4 (Small-Step Semantics of MiniCorten: $\langle e \mid \sigma\rangle \rightsquigarrow \langle e' \mid \sigma'\rangle$)**.**

$$\textsc{SS-Assign}\ \frac{\star}{\langle x = e \mid \sigma\rangle \rightsquigarrow \langle \texttt{unit} \mid \sigma[x \mapsto [\![e]\!]\sigma]\rangle}$$

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

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

$$\text{SS-IF-T}\ \frac{[\![x]\!]\sigma = \texttt{true}}{\langle \texttt{if } x\ \{s_t\} \texttt{ else } \{s_e\} \mid \sigma\rangle \rightsquigarrow \langle s_t \mid \sigma\rangle} \quad \text{SS-IF-F}\ \frac{[\![x]\!]\sigma = \texttt{false}}{\langle \texttt{if } x\ \{s_t\} \texttt{ else } \{s_e\} \mid \sigma\rangle \rightsquigarrow \langle s_e \mid \sigma\rangle}$$

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

$$\textsc{SS-Call}\ \frac{\langle \Sigma(f)[arg_1 \triangleright x_1, \ldots, arg_n \triangleright x_n] \mid \sigma\rangle \rightsquigarrow \langle v_r \mid \sigma'\rangle}{\langle f(x_1, \ldots, x_2) \mid \sigma\rangle \rightsquigarrow \langle v_r \mid \sigma' \oplus \sigma\rangle}$$

where

$$(\sigma' \oplus \sigma)(x) = \begin{cases} \sigma(x) & \text{if } x = arg_i \\ \sigma'(x) & \text{if } x \in \text{dom}(\sigma) \cap \text{dom}(\sigma') \\ \sigma(x) & \text{otherwise} \end{cases}$$

The SS-Call rule does not scope the variables from the callee to the caller. This is actually fine, because by definition variable names are distinct. Therefore only the arguments need to be replaced in the resulting state.

Chapter 5

The Refinement Type System

5.1 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 5.1.7.

5.1.1 Decidable, Conservative Subtyping

The idea

5.1.2 Mutable Values

Corten elected to ...Therefore assigning a new value to a variable will give that variable a new type, but crucially keep the logic variable in the typing context, which means that types of other variables can still refer to it. Logic variables, that are (no longer) associated with a program variable, are called dangling variables. That means, that the number of predicates in the context increases with every new value assignment. In the example, the typing context at the end of the function body contains v_1, v_2 and their predicates and the association of `i` to v_2.

There are other approaches for handling changing values in refinement types, which will be described and compared in 8.

```
fn inc() -> ty!{ v: i32 | v == 3 } {
    let mut i = 2;        // {v_1 : i32 | v_1 == 2}
    i = i + 1;            // {v_2 : i32 | v_2 == v_1 + 1}
    i
}
```

Listing 2: Example demonstrating why predicates and mutable values may cause problems

5.1.3 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.

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]

The problem with mutable references is possibility of aliasing. Aliasing is problematic, because it might create interdependencies between the types of different variables: If one variable is changed, it might effect wether another variable is typed correctly. In listing 3 the return value `a` is effected by changes done to a different variable `a`. Conservative approximation requires, that all possible effects must be tracked.

- <dangling>, monotonically increasing φ

Corten can accurately track references by phrasing them as predicates. In the example, in `b = &mut a`, `&mut a` will have inferred type `{ r : &mut | r == & a }`, meaning any value of this type can at most refer to to the variable called `y`. When a new value is assigned to `*b`, the type system will look at the typing context to find out what `b` might refer to. In this case `a` is the *only* possible target und we can therefore *update* its type. I.e. in type system, `*b = 0` will change the type of `a` to `{ s : i32 | s == 0 }`, but the type of `b` stays the same (because it still refers to the same location). We call this kind of assignment a strong assignment, as it can change the type.

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 7.3

[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

Note, that the type can only be changed because we know exactly what `b` refers to. If there is ambiguity about the destination of a reference, strong updates are no longer possible.

```
fn client() -> ty!{ v: i32 | v == 4 } {
    let a = 2;        // {v1 : i32 | v1 == 2}
    let b = &mut a;   // {v2 : &i32 | v2 == &a}
    *b = 0; // changes a's value and type
    let c = &mut b;
    **c = 4; // changes a's value and type
    a
}
```

Listing 3: Example demonstrating interdependencies between mutable references

To support these use cases, Corten also supports weak updates, which can not change types, but allow assigning to ambiguous reference destinations. The example 4 demonstrates how ambiguity about the reference destination may emerge: Depending on the if condition, `res` could refer to either `y` or `z`. Naturally, we can weaken the reference type to `ty!{ r1 : &mut i32 | r1 == &y || r1 == &z }`, meaning `res` could refer to either `y` or `z`. Because the destination is ambiguous, assigning to `*res` can not change the type of `y` or `z`, which is the case if the type of the assigned value is at least as specific as the types of all possible reference destinations.

- importance of tracing return of mutable references

```
fn weak_updates(x : ty!{ x1 : i32 }) -> ty!{ v : i32 | v > 2 * x1 + 10 } {
  let mut y = x as ty!{ y1 : i32 | y1 >= x1 };
  let mut z = x + 10 as ty!{ z1 : i32 | z1 >= x1 + 10 };
  let mut res;
  if x > 0 {
    res = &mut y as ty!{ r : &mut | r == &y || r == &z };
                  // branches of if need same type => weaken
  } else {
    res = &mut z as ty!{ r : &mut | r == &y || r == &z };
  }
  *res = x + 11; // res could refer to b or c
                // -> assigned value must satisfy both types
  y + z
}
```

Listing 4: Example demonstrating weak updates

5.1.4 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 5 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.

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

- inference - also with mutations

5.1.5 Strong Type Updates

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

Listing 6: Example of changes to `a`'s value effecting its type

5.1.6 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 7 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 7: Example showing how Corten allows for accurate type checking in the presence of function calls

5.1.7 Mutual Reference

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

why?

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

Listing 8: Function computing the maximum of its inputs; guaranteeing that the returned value is larger than its inputs

5.1.8 Atomic Updates

- relax ctx - non-atomic might not be sufficient

5.2 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 $(\mu[a \mapsto \alpha], \Phi)$

5.3 Statement Type Checking $\Gamma \vdash s \Rightarrow \Gamma'$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.5 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 5.1.2, 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.

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

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

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

alternative (should be equivalent):

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

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

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

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.

5.8 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 5.8.1 (State Conformance)**.** A state σ is conformant with respect to a typing context $\Gamma = (\mu, \varphi)$ (written as $\sigma : \Gamma$), iff:

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

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

Definition 5.8.2. Progress: If t is closed and well-typed, then t is a value or $t \rightsquigarrow t'$, where t' is a value.

Definition 5.8.3 (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 5.8.4 (Conformance of Evaluation)**.** *If* $\sigma : \Gamma$ *and* $[\![e]\!]\sigma = v$ *then* $\vDash \varphi \rightarrow [\![e]\!]\Gamma$

Proof. □

Lemma 5.8.5 (Preservation of State Conformance)**.** *If* $\Gamma \vdash s \Rightarrow \Gamma_2$, $\sigma : \Gamma$ *and* $\langle s \mid \sigma \rangle \rightsquigarrow \langle s_1 \mid \sigma_1 \rangle$, *then* $\sigma_1 : \Gamma_1$ *and* $\Gamma_1 \vdash s_1 \Rightarrow \Gamma_2$ *for some* Γ_1

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

- Case SS-Assign: Given $\sigma :: \Gamma$, $\Gamma \vdash x = e \Rightarrow \Gamma_2$ show $\exists \Gamma_1, \sigma[x \mapsto [\![e]\!]\sigma] : \Gamma_1 \wedge \Gamma_1 \vdash \texttt{unit} \Rightarrow \Gamma_2$

  Rule inversion over $\Gamma \vdash x = e \Rightarrow \Gamma_2$ gives $\Gamma \vdash e : \{\alpha : b \mid \alpha \doteq [\![e]\!]\}$ and $\Gamma_2 = (\Gamma[x \mapsto \alpha], \alpha \doteq [\![e]\!])$

  For $\Gamma_1 = \Gamma_2$, $\sigma[x \mapsto [\![e]\!]\sigma] : (\Gamma[x \mapsto \alpha], \alpha \doteq [\![e]\!])$, α is fresh and therefore monotonicity applies.

- Case SS-Seq-Inner: Given: $\langle c_1 \mid \sigma \rangle \rightsquigarrow \langle c_1' \mid \sigma' \rangle$,
  $\Gamma \vdash c_1; c_2 \Rightarrow \Gamma_2$,
  $\forall \Gamma_2, \Gamma \vdash c_1 \Rightarrow \Gamma_2 \wedge \sigma : \Gamma \rightarrow \exists \Gamma_1, \sigma' : \Gamma_1 \wedge \Gamma_1 \vdash c_1' \Rightarrow \Gamma_2$
  show $\exists \Gamma_1, \sigma' : \Gamma_1 \wedge \Gamma_1 \vdash c_1'; c_2 \Rightarrow \Gamma_2$.

  Rule inversion over $\Gamma \vdash c_1; c_2 \Rightarrow \Gamma_2$ yields $\Gamma \vdash c_1 \Rightarrow \Gamma_1$, $\Gamma_1 \vdash c_2 \Rightarrow \Gamma_2$

  The preconditions for the third induction hypothesis are satisfied for Γ_1 delivers $\exists \Gamma_1', \sigma' : \Gamma_1' \wedge \Gamma_1' \vdash c_1' \Rightarrow \Gamma_1$

  State conformance $\sigma' : \Gamma_1'$ follows directly from this.

  For Γ_1' the preconditions for the Seq rule are satisfied.

- Case SS-Seq-N: Given: $\Gamma \vdash \texttt{unit}; c \Rightarrow \Gamma_2$, $\sigma : \Gamma$ show $\exists \Gamma_1, \sigma : \Gamma_1 \wedge \Gamma \vdash c \Rightarrow \Gamma_2$.

  Rule inversion over $\Gamma \vdash \texttt{unit}; c \Rightarrow \Gamma_2$ yields $\Gamma \vdash c \Rightarrow \Gamma_2$. Together with the second induction hypothesis this implies the goal.

- Case SS-Decl: analogous to Case SS-Assign

- 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, $\tau' = \{l \mid l \doteq v\}$

$\square$

Lemma 5.8.6. *State Conformance is monotone: If $\sigma : \Gamma$, $(\beta = v) \rightarrow \varphi$ then $\sigma[x \mapsto v] : \Gamma[x \mapsto \beta] \wedge \varphi$* sensible name?

Proof.

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

...? $\square$

5.9 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?

5.9.1 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: $\mu : \text{Path} \to \text{LVar}$. The relevant typing rules will work without major changes:

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

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

5.9.2 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 `enums`.

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.

5.9.3 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.

5.9.4 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: CortenC

The type system described in the previous chapter was implemented to test the practical feasibility of our approach. There are a few differences and details between the type system described above and the implementation CortenC (short for Corten Checker) that will be highlighted in this chapter.

In contrast to the MiniCorten and the described type system, CortenC uses actual Rust as its target language. In addition to MiniCorten's features, CortenC also covers expressions with side-effects, non-ANF expressions, statements as well as basic inference. These features do not change the expressiveness of the type system, but make it a lot more practical.

Target Selection The architecture of the rustc compiler makes it possible to implement a refinement type system quite cleanly: Rustc's plugin system allows CortenC to access rustc's intermediate representations and also emit diagnostics with source locations. Since the diagnostics from CortenC use the same interface as Rust's diagnostics, other tools like IDEs can handle them without any adaptation.

CortenC uses the HIR. It is a good candidate for extensions to the type system, because it contains source labels, allowing accurate problem reporting to the user. Secondly, in HIR all names are resolved making and types can be queried for HIR nodes, which means the Rustc plugin can reuse a lot of work done in the Rustc Compiler. Thirdly, non-executable segments of the program, like type declarations, are represented in the HIR.

Using MIR is also a sensible option: The MIR is a drastically simplified representation of just the executable segments of the program as a CFG with non-nested expressions. Because of the simplicity, it would be appealing to use the MIR as the basis for the implementation. An additional advantage of MIR could be, that ownership analysis is done on MIR, which would allow the refinement type system to use that information. Surprisingly, we did not encounter any situation, where ownership information is necessary for typing refined types.

Ultimately, it was decided against using the MIR for the following reasons:

Firstly a concern for the quality of diagnostics: The MIR code is quite distant from the user-provided code. For example, when trying to explain a error when typing a while-loop, the implementation may need to reconstruct the original source code structure from the CFG, which could be inaccurate and error-prone. In addition, source locations may be less accurate or unavailable. Finally the MIR does not contain type declarations, which could make it hard to extend the implementation for algebraic data type or predicate generics (see section 5.8)

Language Embedding CortenC exposes a minimal interface to the programmer: The interface consists of two macros: `ty!{ ... }` which allows the programmer to specify a refined type for a rust base type and `relax_ctx!{ ... }` to allow the programmer to relax the typing context (mostly used for introducing a loop-invariant).

Listing 9 shows how the `ty!` macros is defined: Most important are the first two macro forms, which handle immutable types like `ty!{ v : i32 | v > 0}` and mutable parameter types like `ty!{ v1: i32 | v1 > 0 => v2 | v2 < 0}`. There are additional short forms of these macro calls, which where cut from the listing for brevity. Both macro calls are translated into the type alias `Refinement` (`MutRefinement` respectively). For the examples above, it would be `Refinement<i32, "v", "v > 0">` and `MutRefinement<i32, "v1", "v1 > 0", "v2", "v2 > 0">` respectively. These type aliases are quite useful for CortenC: They ensure, that CortenC keeps compatibility with Rust. I.e. a program with CortenC type annotations can be compiled in normal Rust without problems and the type aliases also simplify the implementation of CortenC, because the refined type can be extracted directly from the Rust type alias. Consequently, CortenC does not need to perform name resolution, even for external function declarations.

Note that, because the `ty!` macro takes the place of a Rust type, normal IDE features, like ”Goto Type Definition” still work on the base types without any adaptation.

For the second macro `relax_ctx!` takes the place of a statement. It is translated to a function call, that can then be used to reconstruct the described context.

Architecture CortenC follows the structure implied by the type system: All concepts and type checking rules have a correspondence in CortenC: `RContext` is the representation of Γ in CortenC, with the main difference being, that predicates stay associated with a logic variable (and potentially a program variable). The purpose being, that in the future, CortenC could create better error messages, by associating ”blame” to a type checking failure: If the predicate of a logic variable is part of the reason why a subtyping check fails, it would be useful to convey that information to the user. `RefinementType` is the representation of τ in CortenC.

CortenC registers a callback in Rustc to run after HIR construction. The callback calls `type_check_function` for each function in the HIR, which either

```
pub type Refinement<
  T,                      // Rust type
  const B: &'static str,  // Logic Variable
  const R: &'static str   // Predicate
> = T;

pub type MutRefinement<
    T,                      // Rust type
    const B1: &'static str, // Logic Variable Before
    const R1: &'static str, // Predicate Before
    const B2: &'static str, // Logic Variable After
    const R2: &'static str, // Predicate After
> = T;

#[macro_export]
macro_rules! ty {
  ($i:ident : $base_ty:ty | $pred:expr) => {
      $crate::Refinement< $base_ty, {stringify! { $i }}, {stringify! { $pred }}>
  };
  ($i:ident : $base_ty:ty | $pred:expr => $i2: ident | $pred2:expr) => {
      $crate::MutRefinement< $base_ty, {stringify! { $i }}, {stringify! { $pred }}, {stringify!
  };
  // -- snip --
}
```

Listing 9: Definition of CortenC's macros

returns `Result::Ok` indicating, that the function type checks or `Result::Err` if type checking failed. In that case, the callback will emit a diagnostic to rustc to inform the user about the refinement type error. Listing 10 shows the core functions signatures involved in type checking. `type_check_function` corresponds to the Fn-Decl rule by constructing the initial context and delegating checking of the body to `type_expr`. Since Rust expressions can contain statements, `type_expr` will use `transition_stmt` to type check these statements before returning the final type context. Both `type_expr` and `transition_stmt` carry a collection of parameters, that contain the Rust type checking data (The global type context `TyCtxt` and the local type context `TypeckResults`), the refinement type context `RContext`, a handle to the SMT solver `SmtSolver` and a way to generate fresh logic variable names `Fresh`.

These function utilize `require_is_sub_context` and `require_is_subtype_of` to dispatch the actual SMT solver requests. These functions are also responsible for encoding the predicates and substituting variable names where necessary. The following paragraph will give an example for how these SMT solver requests look like.

Simple Example Consider the example program **??**, which returns the increment of the argument. Rustc will call the CortenC callback with the item `fn inc` once the HIR construction is completed. Next, the callback calls `type_check_function` where the initial context $\Gamma = (a \mapsto n, \{n > 0\})$ is constructed and calls `type_expr` with the the function body expression. Base the function body is a sequence `transition_stmt` is called for all expect the last expression in the sequence. `let b = 1` updates the context to $\Gamma_1 = (a \mapsto n, b \mapsto v_1, \{n > 0, v_1 = 1\})$ (v_1 is a name generated by `Fresh`). Finally `type_expr` types the expression `a + b` according to the typing rules described above resulting in the type $\{v_2 : i32 \mid v_2 = n + v_1\}$. This type is then returned as the overall type of the body expression to `type_check_function`, which has the responsibility of checking that in the context Γ_1, the actual return type is a subtype of the specified type in the signature. Thusly `require_is_subtype_of` is called with the two types and subsequently dispatches the SMT shown in listing **??**. Because the SMT call returned `unsat`, the subtype relation is valid and the function type checking was successful.

```
fn type_check_function(
  function: &hir::Item,
  tcx: &TyCtxt,
) -> Result<RefinementType> { .. }

fn transition_stmt(
  stmts: &a [hir::Stmt],
  tcx: & TyCtxt,
  ctx: & RContext,
  local_ctx: & TypeckResults,
  solver: &mut SmtSolver,
  fresh: &mut Fresh,
) -> Result<RContext> { .. }

fn type_expr(
  expr: &a Expr,
  tcx: & TyCtxt,
  ctx: & RContext,
  local_ctx: & TypeckResults,
  solver: &mut SmtSolver,
  fresh: &mut Fresh,
) -> Result<(RefinementType, RContext)> { .. }

fn require_is_sub_context(
  super_ctx: &RContext,
  sub_ctx: &RContext,
  tcx: &TyCtxt,
  solver: &mut SmtSolver,
) -> anyhow::Result<()> { .. }

fn require_is_subtype_of(
    sub_ty: &RefinementType,
    super_ty: &RefinementType,
    ctx: &RContext,
    tcx: &TyCtxt,
    solver: &mut SmtSolver,
) -> anyhow::Result<()> { .. }
```

Listing 10: Overview of the central function CortenC is built from. (Lifetimes were removed for clarity)

```
fn inc(a : ty!{ n : i32 | n > 0}) -> ty!{v : i32 | v > 1} {
  let b = 1;
  a + b
}
```

Listing 11: Simple Example Program used for demonstrating the operation of CortenC

```
(declare-datatypes () ((Unit unit)))

; <Context>
  ; decl for <fud.rs>:79:73: 79:74 (#0) local b
  (declare-const _0 Int)

  ; decl for <fud.rs>:79:9: 79:10 (#0) local a
  (declare-const n Int)

  ; predicate for _0: local b
  ;     ty!{ _0 : i32 | _0 == 1 }
  (assert
      (= |_0| 1)
  )

  ; predicate for n: local a
  ;     ty!{ n : i32 | n > 0 }
  (assert
      (> |n| 0)
  )
; </Context>

(declare-const _1 Int)
(assert
    (= (+ |n| |_0|) |_1|)
)

(assert
    (not (> |_1| 1))
)

; checking: ty!{ _1 : i32 | n + _0 == _1 }    ty!{ v : i32 | v > 1 }
(check-sat)

; done checking is_subtype_of! is sat: false
```

Listing 12: SMT Requests dispatched by CortenC for checking that the returned type matches the specified type

Chapter 7

Evaluation

In this chapter Corten will be tested on a selection of examples to test the practicality.

7.1 Maximum using Path Conditions

The first example will show how a the `max` function can be implemented and fully verified in CortenC. The mathematical maximum is defined as $r = max(a, b) iff r \geq a \wedge r \geq b \wedge (r == a \vee r == b)$. Listing 13 shows how this can be expressed in CortenC.

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

Listing 13: Example demonstrating a fully specified `max` function using Corten's path sensitivity

Of course an incorrect implementation will produce a type judgement error. If, for example, the else branch returned `a` instead of `b`, the system would return the error shown in listing 14. The exact location of where the error occurred is also the current implementation does not expose this data yet.

Path sensitivity is not limited to the return value: Effect on the type context can also be path sensitive as demonstrated by listing 15, which implements a clamping function. The function `clamp` ensures, that the reference passed to it will be at most `max` or stay the same. `client` uses `clamp` and can use the

```
Subtyping judgement failed:
ty!{ _1 : i32 | true && _1 == a } is not a sub_ty of
ty!{ v : i32 | v >= a && v >= b && (v == a || v == b) }

in ctx RContext {
  // formulas
  // types
  local a : ty!{ a : i32 | true }
  local b : ty!{ b : i32 | true }
}
```

Listing 14: Example of an error message created by CortenC

facts from `clamp`'s mutation specification when proving its own return type specification.

Note, that when typing `client`, CortenC needs to match `&mut x` with `a` and thusly `x` with `s` to correctly handle the effects of calling `clamp`. Because of this reason, function calls require all arguments to be variables or reference to variables. This is not a restriction of the type system, but a simplification for the implementation.

```
fn clamp(
    a: &mut ty!{ a1: i32 | true => s | (s <= max) && (s == a1 || s == max) },
    max: ty!{ max: i32 }
) -> ty!{ v:  () } {
    if *a > max {
        *a = max as ty!{ r | (r <= max) && (r == a1 || r == max) }; ()
    } else {};
    ()
}

fn client() -> ty!{ v : i32 | v == 42 } {
    let mut x = 1337; let max = 42;
    clamp(&mut x, max);
    x
}
```

Listing 15: Example demonstrating optional mutation of an external location

7.2 Recursion: Fibonacci-Numbers

Thanks to Corten's modular approach, handling recursive functions is strait forward in CortenC. Listing 16 demonstrates this by implementing a function, that returns the nth fibonacci number F_n and asserting that $F_n \geq n^2$. Because n is arbitrary, this is proves that the fibonacci sequence grows faster than the n^2.

```
fn fib(n: ty!{ nv: i32 | nv >= 0}) -> ty!{ v: i32 | v >= nv * nv } {
  if n >= 2 {
      let n1 = n - 1; let n2 = n - 2;

      let f1 = fib(n1); let f2 = fib(n2);
      (f1 + f2) as ty!{ r : i32 | r >= nv * nv }
  } else {
      1
  }
}
```

Listing 16: Example demonstrating recursive function calls by proving a divergence property of the fibonacci sequence

7.3 Loop Invariants: Proof of the Gauss Summation Formula

The next example demonstrates, that CortenC can also verify loop invariants. Listing 17 shows a function, that calculates the sum $\sum_{i=1}^{N} i$ by repeatedly increment a variable by i. Of course there exists a closed formula computing the same result: $\frac{n\cdot(n+1)}{2}$, which is given as the specification. CortenC will therefore proof, that the closed form return the same result compared to the iterative summation.

This example is quite challenging, because of interdependencies between `i` and `sum` in the invariant as well as the old value of `i` and old values of `sum` in the loop body.

To proof, that the invariant holds after each loop body execution, we need the exact information, that `i` was incremented by one and `sum` by `i` as well as the knowledge, that `i` and `sum` satisfied the predicates. Corten solves this problem by treating predicates as immutable and retaining ”dangling” predicates (i.e. no longer associated with a program variable).

The second challenge is establishing the loop invariant in the first place: Relaxing the type of `i` from $= 0$ to $\leq nv$ is only valid, because we know relaxing the type of `sum` from $= 0$ to $2 * sv == iv * (iv + 1)$ is valid if $iv = 0$ and vice versa: To update the type of `sum` first, we need the knowledge, that $iv = 0$, but once we relax the type of `i`, `i` gets a new logic variable, which is distinct from the logic variable used in the type of `sum`. Thus the loop invariant needs to be established atomically.

For this purpose, CortenC provides the `relax_ctx!` macro, that allows the programmer to change the type of multiple variables at a time, as long as the previous state *as a whole* implies the newly chosen predicates.

```
fn gauss(n: ty!{ nv : i32 | nv > 0 })
-> ty!{ v : i32 | 2 * v == nv * (nv + 1) } {
  let mut i = 0;
  let mut sum = 0;

  // Loop Invariant:
  relax_ctx!{
      n |-> nv | nv > 0,
      i |-> iv | iv <= nv,
      sum |-> sv | 2 * sv == iv * (iv + 1)
  }
  while i < n {
      i = (i + 1);
      sum = (sum + i);
  }
  sum
}
```

Listing 17: Example loops with complex loop invariants and value updates effecting the invariant

7.4 Complex Mutable References

7.5 Rephrasing built-ins in terms of Refinement Types

The following example demonstrates, that Corten naturally extends to unusual edge cases and still allows easy verification based on them. Rust has built-in functions for aborting execution called `panic` and a function for conditionally aborting execution called `assert`. Listing 18 shows a reimplementation of these functions with full refined type specifications. `panic` asserts that after a call to it, `false` is valid. This naturally follows from inverting the path condition after exiting the loop. `assert` also uses the path condition to assert the correctness of `cond` and uses panic for the else case. The return type of `assert` can use the logic variable `c` from the input directly as the predicate, that is satisfied after the execution. The `client` just needs to make sure, that the proof of `c` is stored in the context by storing the value in the variable `_witness`: If the returned value is discarded, it will not be stored in the context and therefore will not be available when proving the return value specification.

```
fn panic() -> ty!{ v : () | false } {
    while(true) { () }
}

fn assert(cond: ty!{ c : bool }) -> ty!{ v: () | c } {
    if cond {
        () as ty!{ v: () | c }
    } else {
        panic()
    }
}

fn client(a : ty!{ av: i32 }) -> ty!{ v: i32 | v > 0 } {
    let arg = a > 0;
    let _witness = assert(arg);
    a
}
```

Listing 18: Example showing how `panic` and `assert` can be naturally specified and verified in CortenC

7.6 Extendability with Other tools

- Corten: only safe - Expend Corten by using external tools to verify unsafe - use assert to tell Corten what to assume and external tool what to verify

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.

8.0.1 Alternative Approaches for Handling Mutability

Of course the design space permits choosing other tradeoffs. For example, Rondon et al. [**rondon_low-level_2010**] chooses - only interdependence in single record using fold mechanism -

this is not the only way to deal with mutability:

8.1 Comparison to Flux

Chapter 9

Conclusion & Future Work