Karlsruhe Institute of Technology

LiquidRust: Refinement Types for Imperative Languages with Ownership

Master's Thesis of

Carsten Csiky

at the Department of Informatics
Institute of Information Security and Dependability (KASTEL)

Reviewer:	Prof. Dr. Bernhard Beckert
Advisors:	Dr. Mattias Ulbrich Sebastian Graf, M.Sc. Florian Lanzinger, M.Sc.

4th April 2022 – 4th Oktober 2022

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, **4.10.2022**

. . . .

(Carsten Csiky)

Abstract

Software correctness is a central goal for software development. One way to improve correctness is using strong and descriptive types, that express correctness properties as types, and verifying their compatibility with the programs by using 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 in a lightweight and gradual way and verified automatically.

However existing approaches for adapting Refinement Types from functional to imperative languages showed that the presence of mutable data and references entails compromising on at least one of the core features of Refinement Types, which are the automated, decidable verification and minimal, approachable specification.

This thesis presents a Refinement Type system with minimal compromises by taking advantage of Rust's restriction to unique mutable references.

To this end, we augment a flow-sensitive Refinement Type system with a varying typing context that facilitates type updates and tracks reference destinations. We define a notion of subtyping on this context and argue for the soundness of our typing rules. Furthermore we implement a prototype verifier for an extended version of our type system as a plugin to the Rust compiler and evaluate its efficacy on a selection of examples, demonstrating automatic verification and minimal, approachable specification.

Our work provides insights into the unique advantages that Rust's ownership system provides for a Refinement Type system. Additionally we analyse a large dataset of existing Rust code to gauge how relevant language features are used in practice.

Zusammenfassung

Die Korrektheit von Software ist ein zentrales Ziel der Softwareentwicklung. Eine Möglichkeit die Korrektheit zu verbessern, ist das Verwenden starker und deskriptiver Typen, die Korrektheitseigenschaften als Typen kodieren, und die Kompatibilität mit dem Programm mit Hilfe von Typsystemen verifizieren. Insbesondere Refinement Types haben gezeigt, dass selbst mit einem Verifikationssystem, das auf eine entscheidbare Logik beschränkt ist, komplizierte Eigenschaften leicht ausgedrückt und automatisch verifiziert werden können.

Bestehende Ansätze zur Anpassung von Refinement Types von funktionalen an imperative Sprachen haben jedoch gezeigt, dass die Existenz von veränderbaren Daten und Referenzen Kompromisse bei mindestens einer der Kerneigenschaften von Refinement Types mit sich bringt, nämlich der automatisierten, entscheidbaren Verifikation und einer minimalen, leicht verständlichen Spezifikation.

In dieser Arbeit wird ein Refinement-Type-System mit geringfügigen Kompromissen vorgestellt, indem Rusts Beschränkung auf unique mutable Referenzen ausgenutzt wird.

Zu diesem Zweck erweitern wir ein flusssensitives Refinement-Typsystem um einen variierenden Typisierungskontext, der Typaktualisierungen ermöglicht und Referenzen rückverfolgt. Dafür wird der Begriff der Subtypisierung auf diesem Kontext definiert und die Korrektheit der Typregeln begründet. Darüber hinaus wurde Prototyp ein Typ-Checkers als Rust-Compiler Plugin entwickelt, der mit Hilfe einer erweiterten Version des vorgestellten Typsystems, zur Evaluation der Effektivität des Ansatzes anhand einer Auswahl von Beispielen verwendet wird, welche die automatische Verifikation, den geringen Spezifikationsaufwand und die leichte Zugänglichkeit demonstrieren.

Unsere Arbeit bietet Erkenntnisse in die ungewöhnlichen Vorteile, die das Rust-Ownership-System für ein Refinement-Typsystem bietet. Zusätzlich wird ein großer Datensaty von bestehendem Rust-Code analysiert, um festzustellen, wie relevante Sprachmerkmale in der Praxis verwendet werden.

Contents

Abstract i

Zusammenfassung iii

1. Introduction 1

2. Foundations 3
- 2.1. Refinement Types 3
- 2.2. Rust 4
  - 2.2.1. Insights into Rust as a Verification Target 6
- 2.3. Rustc 7

3. Empirical Analysis of Use-Cases 11
- 3.1. Unsafe Rust 11
- 3.2. Mutability 13
- 3.3. Conclusions 15

4. The MiniCorten Language 17
- 4.1. Syntax 17
- 4.2. Semantics 17

5. The Refinement Type System 21
- 5.1. Features 21
  - 5.1.1. Decidable, Conservative Subtyping 21
  - 5.1.2. Path Sensitivity 21
  - 5.1.3. Mutable Values 22
  - 5.1.4. Mutable References 23
  - 5.1.5. Modularity 25
  - 5.1.6. Atomic Updates 25
- 5.2. Type System 26
- 5.3. Soundness of the Type System 31
- 5.4. Extensions 38
  - 5.4.1. Records / `structs` 38
  - 5.4.2. Algebraic Data Types 39
  - 5.4.3. Inference 39
  - 5.4.4. Predicate Generics 39
  - 5.4.5. Convenience Improvements 39

6. Implementation: CortenC 41

7. Evaluation **47**
7.1. Maximum using Path Conditions 47
7.2. Recursion: Fibonacci-Numbers 48
7.3. Loop Invariants: Proof of the Gauß Summation Formula 49
7.4. Complex Mutable References 49
7.5. Pseudo Vectors 50
7.6. Rephrasing built-ins in terms of Refinement Types 51
7.7. Interoperability with Other Tools 51

8. Related Work **55**
8.1. Comparison to Flux 56

9. Conclusion & Future Work **61**

Bibliography **63**

A. Additional Graphs from the Use-Case Analysis **67**

List of Figures

2.1. Diagram providing an overview of the Rustc Compiler Stages 8

3.1. Scatter plot of a point representing creates, relating the number of unsafe occurrences to the number of unsafe LOC 12
3.2. Cumulative, Logarithmic Histogram of the Amount of `unsafe` Uses in each Category 13
3.3. Ratio of Immutable to Mutable Items of Different AST Nodes. 14

4.1. Syntax of MiniCorten expressed in Backus-Naur Form 20

8.1. Figure showing a value space, where inclusion can not be shown 58

A.1. Cumulative, Logarithmic Histogram of the Amount of `unsafe` Uses in each Category 67
A.2. Logarithmic Histogram of the Number of Crates that contain n % immutable / mutable items 68

List of Tables

3.1. Number of Unsafe Uses by Crate Category 13

List of Listings

1. Example demonstrating the Ownership System: `greet(a)` transfers ownership of `a` to `greet` 5
2. Example demonstrating borrowing: `greet(&a)` lends `a` to `greet` 5
3. Example demonstrating mutable borrowing: `age(a)` needs a mutable borrow of user to increase the age, but `a` is borrows immutably, which is forbidden 6
4. Example of an apparent violation of ownership rules 7

5. Function computing the maximum of its inputs; guaranteeing that the returned value is larger than its inputs 22
6. Example demonstrating why predicates and mutable values may cause problems 23
7. Example demonstrating interdependencies between mutable references . 24
8. Example demonstrating weak updates 24
9. Example showing how Corten allows for accurate type checking in the presence of function calls 25
10. Example Demonstrating Interdependence between Types 26
11. Example Showing the Correspondence between Function Calls and Context Assumptions 30
12. Example for how `structs` could be handled in Corten 38

13. Definition of CortenC's macros 43
14. Overview of the Central Functions CortenC is Built from 44
15. Simple Example Program used for demonstrating the operation of CortenC 44
16. SMT Requests dispatched by CortenC for checking that the returned type matches the specified type 45

17. Example demonstrating a fully specified `max` function using Corten's path sensitivity 47
18. Example of an error message created by CortenC 48
19. Example demonstrating optional mutation of an external location 48
20. Example demonstrating recursive function calls by proving a divergence property of the fibonacci sequence 49
21. Example loops with complex loop invariants and value updates affecting the invariant 50
22. Example demonstrating modularity and ease of specification for complex mutation patterns 50

23. Example demonstrating modularity and ease of specification for complex mutation patterns 52
24. Example showing how `panic` and `assert` can be naturally specified and verified in CortenC 53
25. Macro provided by CortenC to offload verification to other tools 53

26. Example demonstrating the Ownership System: `greet(a)` transfers ownership of `a` to `greet` 57
27. Comparison of specifying Type Changes Caused by a strong mutation. Flux on the left; Corten on the right 57

1. Introduction

With increasing amount and reliance on 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[6], 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 is 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, some error reporting and a simple counter example generation system.

The main 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 empirical analysis uses a parser framework to extract relevant syntactic structures from open-source Rust code. To cover mutability, we will extend a path-sensitive Refinement Type system with variable typing context to allow types to change, just like values. Mutable References will be handled by a restricted subset of the refinement language called reference predicates. The implementation extends the Rust compiler Rustc with our refinement type system and dispatches subtyping requests to the SMT solver Z3. The evaluation uses Rust code examples in an unaltered syntax.

The thesis is structured as follows: Chapter 2 will give an overview of the foundation the thesis builds upon. Thereafter, in chapter 3 an empirical analysis of `unsafe` and mutability uses is performed to gain an understanding of how Rust is used. Chapter 4 defines the subset of Rust that will be the basis for the type system. Next chapter 5 explains the actual type system and type checking, as well as justifying its correctness, followed by chapter 6, which will provide additional information about how the type system was implemented. The implementation will then be tested on minimal benchmarks in chapter 7. Chapter 8 reports on related work and alternative approaches. Finally chapter 9 concludes the thesis and gives an overview over possible future work.

2. Foundations

We start by giving a basic overview over the fundamental concepts that are relevant for our work. The semantics of our subset of Rust are discussed in chapter 4.

2.1. Refinement Types

The type system that this thesis will adapt, is based on the Refinement Type system described by Vazou et al. [32] and Rondon et al. [25]. 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 $\{v : \text{i32} \mid v > 0\}$ and represent the set of inhabitants that have the specified base type, but also satisfy the predicate.

The notion of refinements is 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). This matches the general maxime of behavioural subtyping: A value of type τ can be used a place that expect τ', because the predicate of τ' is also satisfied for τ. The following rules shows how subtyping is handled by Liquid types [25, p. 6][1]. LT-Sub allows a value e to be typed with the type τ_2 if e has type τ_1 and the subtyping relation $\Gamma \vdash \tau_1 \leq \tau_2$ is satisfied – for instance by $\leq$-Base.

$$\textsc{LT-Sub}\ \dfrac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_1 \leq \tau_2 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash e : \tau_2} \qquad \leq\textsc{-Base}\ \dfrac{\text{SMT-Valid}(\llbracket \Gamma \rrbracket \land \llbracket e_1 \rrbracket \rightarrow \llbracket e_2 \rrbracket)}{\Gamma \vdash \{v : b \mid e_1\} \leq \{v : b \mid e_2\}}$$

The subtyping rule $\leq$-Base requires that in the current context Γ, the predicate of the subtype must imply the predicate of the supertype, meaning a value of type τ_1 may be used in a place requiring τ_2, since τ_2 was shown to satisfy all properties of τ_1. $\llbracket \Gamma \rrbracket$ encodes the predicates in Γ as an SMT formula by substituting v in the type predicates by the program variable's name. For example, we can show, that in the context $\Gamma = (y : \{v : \text{i32} \mid v > 10\}, x : \{v : \text{i32} \mid v > y\})$, x can be typed as $\{v : \text{i32} \mid v > 0\}$:

$$\textsc{LT-Sub}\ \dfrac{\textsc{LT-Var}\ \dfrac{\Gamma(x) = \tau_1}{\Gamma \vdash x : \tau_1} \qquad \leq\textsc{-Base}\ \dfrac{\text{SMT-Valid}(y > 10 \land v > 0 \rightarrow v > y)}{\Gamma \vdash \{v : \text{i32} \mid v > 10\} \leq \{v : \text{i32} \mid v > y\}} \qquad \dots}{\Gamma \vdash x : \{v : \text{i32} \mid v > 0\}}$$

Liquid types also has a mechanism for enabling path sensitivity: When encountering an if expression, the related path condition is added to the context:

---

[1] Names were adapted to better match our notation

$$\textsc{LT-If}\ \frac{\Gamma \vdash e :\ \texttt{bool} \qquad \Gamma; e \vdash e_t : \tau \qquad \Gamma; \neg e \vdash e_e : \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_t\ \texttt{else}\ e_e : \tau}$$

Note, that e_t and e_e need to have the same type, but can use the path condition to create a value of that type. As an example Rondon et al. use the $\max$ function. It is implemented as `fn max(x, y) { if x > y { x } else { y }}` and given the return type $\{v : \text{i32} \mid x \leq v \wedge y \leq v\}$. To show that the then-case has the required return type, we must show, that $x > y \vdash x : \{v : \text{i32} \mid x \leq v \wedge y \leq v\}$, which the following proof tree shows:

$$\text{LT-If}\ \dfrac{\text{LT-Sub}\ \dfrac{\leq\text{-Base}\ \dfrac{\text{SMT-Valid}(x > y \wedge (v = x) \rightarrow (x \leq v \wedge y \leq v))}{\{v : \text{i32} \mid v = x\} \leq \{v : \text{i32} \mid x \leq v \wedge y \leq v\}} \qquad \text{LT-Var}\ \dfrac{\star}{\Gamma \vdash x : \{v : \text{i32} \mid v = x\}}}{\Gamma; x > y \vdash x : \{v : \text{i32} \mid x \leq v \wedge y \leq v\}} \qquad \dots}{\Gamma \vdash\ \texttt{if}\ x > y\{x\}\ \texttt{else}\ \{y\} : \{v : \text{i32} \mid x \leq v \wedge y \leq v\}}$$

Refinement Types can also be understood as dependent types, distinguished by the limited expressiveness of the type. Refinement types trade off expressiveness for automatic type checking. Allowed formulae are chosen from a SMT decidable logic. For instance Rondon et al. [25] choose the EUFA as the logic fragment, meaning a propositional logic with the theories of equality, uninterpreted functions and linear arithmetic.

2.2. Rust

The Rust programming language [19] is a multi-paradigm systems programming language, originally developed by Graydon Hoare at Mozilla Research. Even though Rust is quite young, with the first stable release published in 2015, it has attracted interest in a wide area of applications and organizations. It is used everywhere from Volvo building embedded software for cars [9] to Western Digital implementing a NVMe driver for Linux in Rust [17] and of course the Rust compiler itself. It 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. However, Rust is also influenced by C: Rust extends its ML base with optional mutability of data and references and manual memory management. Many design decisions of Rust can be explained by Rust's ambitions to offer safe and clear ML-like abstractions without compromising on performance compared to C. For example, Rust elected to use manual memory management like C, which guarantees faster and predictable execution times when compared to garbage collection. But crucially, Rust does not compromise on safety to accomplish this: To this end, Rust incorporates the notion of affine types into its type system (called *ownership system* in Rust). This additional type system is inspired by research language Cyclone introduced by Grossman et al. [12] and guarantees the absence of memory errors. Rust also offers a special language subset called `unsafe` Rust, which deactivates the static ownership checking and places the responsibility of ensuring memory safety on the programmer, analogous to C. `unsafe` Rust is used to provide low-level primitives that expose a safe interface that can be used in safe Rust. From here on out, we will focus on safe Rust.

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 ownership system is an implementation of affine types. It uniquely associates each memory object with an owner. The owner of a Rust object is the binding that is responsible for freeing the object. In the listing 1, first `a` is the owner of the `User` object. An object may only be used once, at which point it is consumed and ownership is transferred to its new binding. In the example `greet(user)` uses `a` which transfers ownership to `greet`. At the end of `greet`, there is no new owner, as a result the user object is freed.

```rust
struct User { name: String, age: u32 }

fn greet(user: User) -> String { // greet takes ownership for user
  format!("Hi {}", user.name)
} // End of scope: Owned value user is dropped
...
let a = User { name: "Alice", age: 25 };
let result = greet(a); // Transfer ownership of user to greet
println!(a.name); // error: use of moved value `a`
```

Listing 1: Example demonstrating the Ownership System: `greet(a)` transfers ownership of `a` to `greet`

On closer inspection, handing over the ownership of `a` to `greet` is not sensible: `greet` just needs to read the value, but should not take ownership of `a`. For that purpose, Rust offers a different kind of value transfer called *references*, which have similar execution semantics to pointers. Because creating a reference does not transfer the ownership of the pointee, is is called a *borrow*. Listing 2 rephrases `greet` to only take a borrow of the user, instead of taking ownership. Because `user` is not owned by `greet`, it is not permitted to extract `user.name` (of Type `String`), instead only borrows to the name are allowed.

```rust
fn greet(user: &User) -> String { // greet borrows user
  let name_ref : &String = &user.name; // borrow name from user
  format!("Hi {}", name_ref)
} // End of scope: The borrows are dropped, but the user object is not
...
let a = User { name: "Alice", age: 25 };
let result = greet(&a); // Lend a to greet
println!(a.name); // No error
```

Listing 2: Example demonstrating borrowing: `greet(&a)` lends `a` to `greet`

Borrowing, just like pointers, entail the risk of dangling pointers. Rust protects from this risk, by tracking the lifetimes (i.e. the required duration, where a reference must be valid) of objects. While interesting in its own right, it is not relevant for this thesis.

Mutable references can exacerbate the problem, which leads to the key insight of Rust: Memory unsafety can arise from the mere presence of two references where at least one is mutable. Thus, Rust ensures that for every variable at every point during the execution that variable can either aliased or mutable, but never both at the same time. Listing 3 demonstrates that Rust will not accept a problem where aliased and mutable access is possible at a time. To resolve the issue, the mutable reference should be borrows after the call to `greet`.

```rust
fn age(user: &mut User) {
  (*user).age = user.age + 1;
}
...
let a = User { name: "Alice", age: 25 };
let b = &a;
let c = &mut a;
//      ↑↑↑↑↑↑ error:
//  cannot borrow `a` as mutable because it is also borrowed as immutable
age(c);
let result = greet(&b); // Lend a to greet
println!(a.name);
```

Listing 3: Example demonstrating mutable borrowing: `age(a)` needs a mutable borrow of user to increase the age, but a is borrows immutably, which is forbidden

In conclusion ownership 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. Although the ownership system was originally introduced to allow for safe, manual memory management, it turned out to be useful for other problems as well: For example concurrency, hardware access and automated reasoning all profit from the strong guarantees of safe Rust. The consequences for the latter will be explained in the next subsection.

2.2.1. Insights into Rust as a Verification Target

Rust justifies a new approach to verification due to its unique set of 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 [33] 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.

Explicit Nullability Rust will ensure definite assignment, meaning variables can only be accessed if they contain a value. For verification this means, that there is no need to prove that a variable is defined. If nullability is desired, Rust offers normal sum type `Option<T>` type with the variants `None` and `Some(T)`. By handling the general case of sum types, verification systems can get the handling of nullable data for free.

Minimal subtyping In contrast to object oriented languages, Rust has no user-extendable notion of object 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. At the type level Rust also has a concept of subtyping for `traits`. Because of the limited, well-known scope, these subtyping relations create no ambiguity with a refinement type system.

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 their interaction. For example, listing 4 shows how `cell`'s value is influenced by changes to another variable `r`. This does not mean that the ownership rule *mutability XOR aliasing* are broken: Even though both `cell` and `r` are mutable, `cell` becomes inaccessible while `r` is live. This form of aliasing is very restrictive: there may only be one owner and one borrower alive at a time. Once the borrow is returned, a new reference may be created.

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

Listing 4: Example of an apparent violation of ownership rules

Unfortunately Rust is also quite a big language with a lot of features and details to keep in mind, which makes it easy to get distracted by complexity, that is accidental to the problem at hand.

2.3. Rustc

Since the implementation 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 [23].

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. Figure 2.1 shows the the main compiler stages of Rustc and the associated data structure.

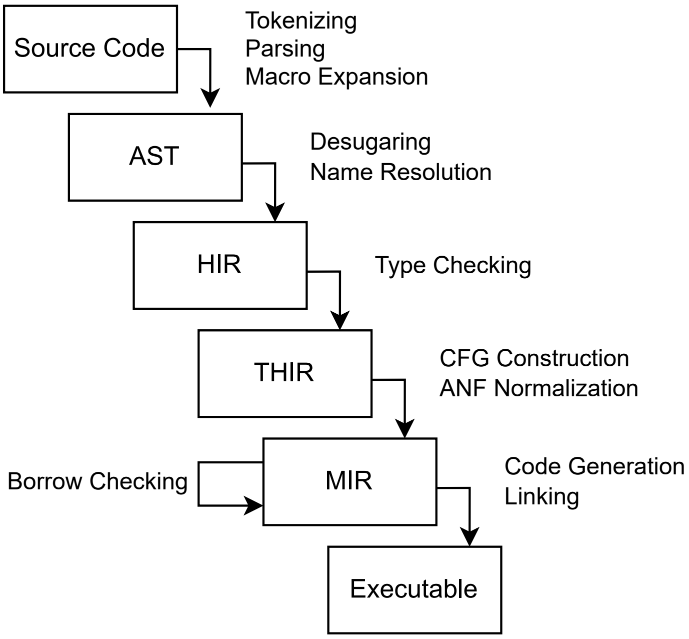

Figure 2.1.: Diagram providing an overview of the Rustc Compiler Stages

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 (HIR for short) 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 (THIR for short), 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. the executable code; this includes function bodies [..]. Consequently, the THIR has no representation for items like `structs` or `traits`." [29, p. 1] 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.

An especially useful feature for this thesis is that 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. Plugins can either wrap the Rustc compiler and hook into callback after each compiler stage or expose a lint pass, that can be registered to be called on different AST items. Although the linting

API is not guaranteed to be stable, some parts of Rust's official tooling depends on it. In particular, the popular Rust linter clippy uses the interface.

3. 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 a similar lack of 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 of Refinement Types, estimating the need for covering this language feature is sensible. The prevalence of mutability should also inform the acceptable level of user effort.

Thus, the use-case analysis should answer the following questions about Rust code:

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?
4. How much effort is acceptable when specifying mutable variables and references in Rust?

To check these assumptions an analysis of existing Rust code was performed[1]. 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[2], 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)[3].

The analysis parses all rust 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.

3.1. 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. [2] 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`.

---

[1] The source code is available at `https://gitlab.com/csicar/crates-analysis/`

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

[3] Calculated with `cloc`

”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.” [2, p. 13] 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.” [2, p. 14]

Although true, when analyzing unsafety 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 expectedly crates.io contains significantly more libraries than binaries[4]: About 81,7% of crates contained just a library, 10,3% contained just an executable target and 10,3% contained both. We also found that libraries are much more likely to use `unsafe` Rust.

Figure 3.1 gives an overview over how `unsafe` sections are distributed in the crates. Each point represents a Rust crate with the x-coordinate representing the number of unsafe section and the y-coordinate representing the ratio of unsafe LOC relative to the total LOC in that crate. The plots for crates containing an executable look empty, but in fact most points are clustered around zero unsafe uses. Also notice the different scales of the x- and y-axis.

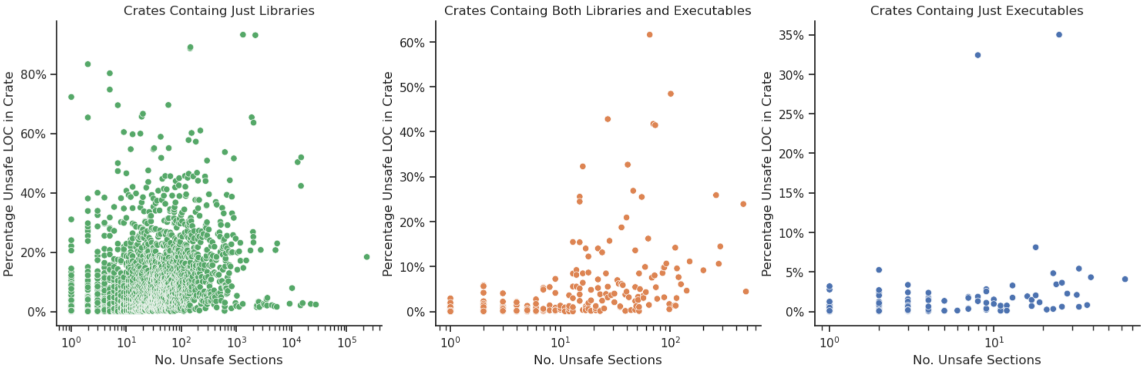

Figure 3.1.: Scatter plot of a point representing creates, relating the number of unsafe occurrences to the number of unsafe LOC

Table 3.1 shows the totaled result of our analysis: The total number of unsafe uses in an order of magnitude higher for binaries compared to libraries. 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 3.2, 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 an executable crate uses `unsafe`, it uses few. Note the difference in x- and y-axis scaling: No executable contains more than 63 occurrences on `unsafe` blocks, but more than 10% of libraries contain more than 63 `unsafe` sections. On average, a library contains about 63 `unsafe` sections, while crates with binaries contain on average about 6 `unsafe` sections.

---

[4] 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.

Crate contains	No. Crates	Total No. of Unsafe Sections	Total Unsafe LOC
Library	9 707	382 997	2 166 213
Both	1 224	7 720	51 004
Binary	951	940	5 873

Table 3.1.: Number of Unsafe Uses by Crate Category

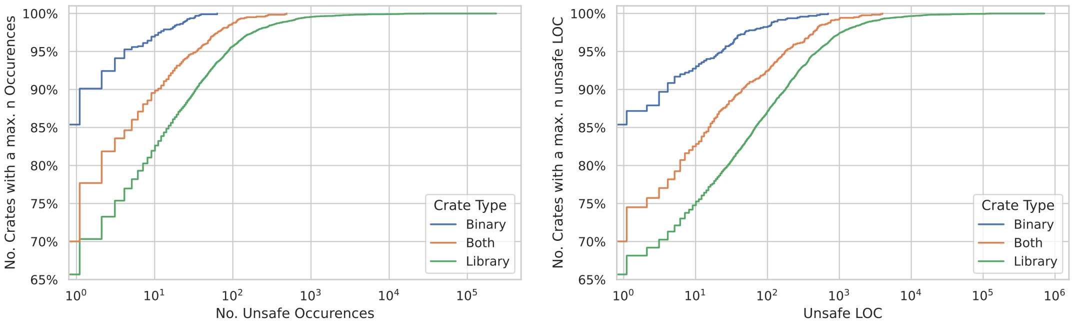

Figure 3.2.: Cumulative, Logarithmic Histogram of the Amount of `unsafe` Uses in each Category

3.2. 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 patterns, we search the dataset for certain syntactical structures to infer mutability information about the following 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** are considered mutable if either the owned value or the reference is mutable. They take the syntactic form: `mut a: i32` or `b: &mut i32`

- **Function Definitions** 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. We consider arguments mutable if are syntactically passed as a mutable reference (for example `&mut i`) and immutable if they are passed as immutable references or literals.

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

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

Figure 3.3 shows the ratio of mutable to immutable items. For each syntactical category, the percentages are relative to the total number of occurrences in the dataset. The grey bar represents data points that cound not be determined to be mutable nor immutable.

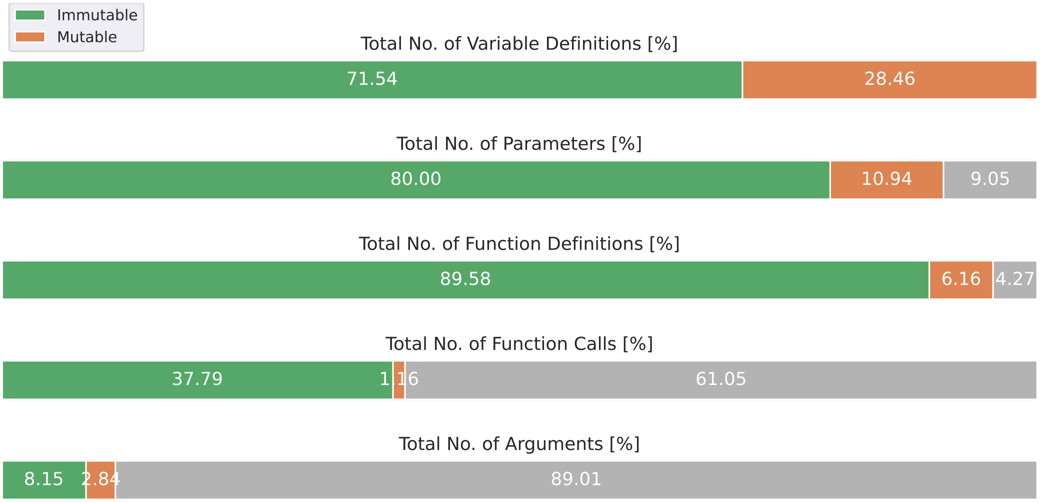

Figure 3.3.: Ratio of Immutable to Mutable Items of Different AST Nodes.

As expected 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 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.

3.3. Conclusions

For this thesis the following conclusion can be drawn:

- Even though uses of `unsafe` are not rare, it is acceptable to ignore them 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.

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 Refinement Types to mutability under the presence of Rust's ownership model.

4.1. Syntax

This subsection will introduce the syntax of MiniCorten[1], 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 A-Normal Form (short ANF), as described by Sabry and Felleisen[27]. ANF normalizes the syntactic structure by requiring that arguments of expressions must be variables. Nested expression are translated into variable declaration for each subexpression. Note that the implementation (for the most part) does not have this restriction.

There are two namespaces for identifiers: $x \in \text{PVar}$ for variables of program variables and $\alpha \in \text{LVar}$ for variable in the refinement predicates. The syntax is defined by Backus-Naur Form grammar in fig. 4.1.

Most constructs are quite standard, the main difference being the addition of refinement types that consist of a logic variable α, a base type b (from the target language) and a predicate φ. Intuitively this means that the value inhabiting that type satisfied the φ in α represents the value. The statement `relax_ctx!{...}` allows the user (or a future inference system) to relax the restrictions in the current typing context (as long as they do not conflict with the actual context). The details will be explained in chapter 5.

4.2. Semantics

The semantics are loosely based on Jung's MiniRust [13], but greatly reduce the complexity by removing stack frames, pointer among other things. However, we of course added refinement types to the language, which do not influence the execution semantics.

In terms of the formal description, the rules are similar to Pierce's [24, p. 166f] "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

---

[1] Corten is named after a alloy of steel, named COR-TEN, that forms an unusually stable and strong form of rust when oxidized (a "refined" rust, if you will).

$\texttt{ref}(x)$ as a value itself where x is the owner. The following definitions show the new execution rules.

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

Definition 4.2.2 (Evaluation of Expressions: $[\![e]\!]\sigma$). Evaluation of an expression e in a state σ is mostly standard. $[\![*x]\!]\sigma$ dereferences x by reading the target from x's value and $\&\texttt{mut}\ x$ introduces a reference to x. The subtype introduction $e\ \texttt{as}\ \tau$ has no effect on the evaluation of e.

$$
\begin{aligned}
[\![v]\!]\sigma &= v \\
[\![x]\!]\sigma &= \sigma(x) \\
[\![x_1 + x_2]\!]\sigma &= [\![x_1]\!]\sigma + [\![x_2]\!]\sigma \\
[\![*x]\!]\sigma &= \sigma(y) \qquad \text{if } \sigma(x) = \texttt{ref}(y) \\
[\![\texttt{\&mut}\ x]\!]\sigma &= \texttt{ref}(x) \\
[\![e\ \texttt{as}\ \tau]\!]\sigma &= [\![e]\!]\sigma
\end{aligned}
$$

Definition 4.2.3 (Declaration Environment: Σ). The environment of function declarations is constant and globally known. It is defined as a partial function Σ : Fn-Name $\rightarrow$ Fn-Decl. A function declaration $\texttt{fn}(\overline{a_i})\ \{s; r\}$ contains the function parameters $\overline{a_i}$, the function body s and the local variable r, that contains the return value.

Definition 4.2.4 (Small-Step Semantics of MiniCorten: $\langle s \mid \sigma \rangle \rightsquigarrow \langle s' \mid \sigma' \rangle$). The execution semantics of MiniCorten are described by the small step semantic $\langle s \mid \sigma \rangle \rightsquigarrow \langle s' \mid \sigma' \rangle$ denoting that the execution of statement s in σ reduces to the execution of s' in σ'. The small-step semantic for MiniCorten consist of:

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

$$\textsc{SS-Assign-Ref}\quad \frac{\sigma(x) = \&y}{\langle *x = e \mid \sigma\rangle \rightsquigarrow \langle \texttt{unit} \mid \sigma[y \mapsto \llbracket e \rrbracket \sigma]\rangle}$$

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

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

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

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

$$\textsc{SS-Relax}\quad \frac{\star}{\langle \texttt{relax\_ctx!\{ ... \}} \mid \sigma\rangle \rightsquigarrow \langle \texttt{unit} \mid \sigma\rangle}$$

$$\textsc{SS-Call}\quad \frac{\Sigma(f) = \texttt{fn}(\overline{a_i}) \longrightarrow (\bar{s}, r) \qquad \bar{l} \text{ local vars in } s \qquad \bar{l'} \text{ fresh names for } \bar{l} \text{ wrt. } \sigma}{\langle \texttt{let}\ x = f(\overline{x_i}) \mid \sigma\rangle \rightsquigarrow \langle \bar{s}[\bar{l} \triangleright \bar{l'}][\overline{a_i} \triangleright \overline{x_i}]; \texttt{let}\ x = r[\bar{l} \triangleright \bar{l'}] \mid \sigma\rangle}$$

The rule SS-Call executes functions by inlining them after renaming the local variable to fresh names and replacing the parameter variables by the argument variables. Rule SS-Relax states, has relaxing the context no effect at runtime.

$program$	::=	$func\_decl\ \star$	*function declarations*
$func\_decl$	::=	x_f`(` $param\ \star$ `) ->` τ `{` s `}`	
$param$	::=	x `:` τ	
s	::=		*statement*
	$\mid$	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` x `{` s_t `} else {` s_e `}`	*if statement*
	$\mid$	`relax_ctx!{` $\varphi\star$ `; (`x `:` τ`)`$\star$ `}`	*context relaxation*
e	::=		*expression*
	$\mid$	x	*variable reference*
	$\mid$	lit	*constant*
	$\mid$	$e \odot e$	*binary operation*
	$\mid$	$\star$ x	*dereference*
	$\mid$	`&` x	*immutable reference*
	$\mid$	`&mut` x	*mutable reference*
	$\mid$	e `as` τ	*type relaxation*
τ	::=		*type*
	$\mid$	`ty!{` α `:` b $\mid$ φ `}`	*refinement type*
φ	::=		*pred*
	$\mid$	ref_pred	*predicate for a reference type*
	$\mid$	$value\_pred$	*predicate for a value type*
ref_pred	::=		*mutable reference predicate*
	$\mid$	α `=` `&` x_1 `|| ... ||` α `=` `&`x_n	*proper reference constraint*
	$\mid$	α_1 `=` α_2	*reference predicate delegation*
$value\_pred$	::=	α	*variable*
	$\mid$	v	*literal*
	$\mid$	$value\_pred \odot value\_pred$	*binary op*
	$\mid$	`!` $value\_pred$	*negation*
b	::=		*base_ty*
	$\mid$	`i32`	*integer*
	$\mid$	`unit`	*unit type*
	$\mid$	`bool`	*boolean*
	$\mid$	`&` b	*immutable reference*
	$\mid$	`&mut` b	*mutable reference*
v	::=		*lit*
	$\mid$	`0,1,...,n`	*integer*
	$\mid$	`true`	*boolean true*
	$\mid$	`false`	*boolean false*
	$\mid$	`()`	*unit value*
$\odot$	::=		*binary operation*
	$\mid$	$\wedge \mid \vee \mid \geq \mid + \mid ==$	

Figure 4.1.: Syntax of MiniCorten expressed in Backus-Naur Form

5. The Refinement Type System

Based on the description of the base language in chapter 4, we will define a describe the proposed refinement type system ”Corten”. Before describing the typing rules itself in section 5.2, the next section will give an overview over the features of the type system, which will also help in understanding why certain design decisions were taken. Section 5.3 will then justify, why the typing rules are sensible, followed by section 5.4 which describes how the type system could be extended.

5.1. Features

The type system surfaces in two ways in the language, which can be 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 `relax_ctx!{ ... }` is used for relaxing the context, which will be explained in the subsection 5.1.6. To make the examples more readable, we will take the liberty to use parts from the implementation CortenC if the formal language is not elaborate enough.

5.1.1. Decidable, Conservative Subtyping

Having a decidable subtyping system is essential for making it feasible in practice: A developer expects type checking to be automatic and part of compilation without additional effort. There is no need for user interaction, if the SMT solver can automatically verify the program. The refinement predicates are translated into assertions of satisfiability of formulae in predicate logic with the theories of linear arithmetic and equality. Thus, an SMT solver can be used to decide if the formula is satisfiable.

5.1.2. Path Sensitivity

First of all, the type system retains the basic features of the refinement type system we build upon.

In particular that it is path sensitive, meaning that the type system is aware of necessary guards that need to be passed for a statement to be evaluated. For example listing 5 shows a function computing the maximum of its inputs. The return type requires, that the result must be at least as big as both inputs[1]. In the then branch, a is only a maximum of a and

---

[1] Corten can also prove the characteristic property of the maximum function. The simpler specification is used for brevity.

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

Listing 5 obviously also demonstrates, that subtyping judgements, like `a as ty!{ x : i32 | x >= av && x >= bv }`, can be introduced and checked.

Because inference is explicitly not the goal of the thesis, our type system is not able to infer, that both branches of the if expression must have the same type as the return type. By adding adding the type relaxation to one branch of the expression, we assist our limited inference system in proving the program type correct.

It should be noted, that path sensitivity is not limited to the value of the expression, but also applies to the mutable effects. Section 7.1 will expand on this example to demonstrate this.

5.1.3. Mutable Values

As seen in section 3.2, mutable declarations are quite common in Rust. Therefore handling them is essential for Corten. To explain the problems that mutable values can cause, consider the problem seen in listing 6: Suppose the value given to `i` is known to be positive, which is expressed by the type $\{v_1 : i32 \mid v_1 > 0\}$, then assigning a new value to `i` may change the type. Notably, the new value's type might depend on the old value, which is the case here. Corten elected to treat predicates as immutable, meaning assigning to a variable will not change the old predicates that was associated with the variable. 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. In this case, the update `i = i + 1` changes the type of `i` to $\{v_2 : i32 \mid v_2 = v_1 - 1\}$. Logic variables that are no longer associated with a program variable (like v_1), are called unassociated variables. That means that the number of predicates in the context increases with every new value assignment. In the example, the typing context initially contains the predicate $v_1 > 0$ and the association of $\{\texttt{i} \mapsto v_1\}$. After the assignment, the predicate $v_2 = v_1 - 1$ is *added* to the context and the association is *changed* to $\{i \mapsto v_2\}$, resulting in the context $\Gamma' = (\{\texttt{i} \mapsto v_2\}, v_1 > 0 \land v_2 \doteq v_1 + -1)$.

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

```
fn decr() -> ty!{ v: i32 | v >= 0 } {
    let mut i = ...;     // i : $\{v_1 : i32 \mid v_1 > 0\}$
    i = i - 1;           // i : $\{v_2 : i32 \mid v_2 == v_1 - 1\}$
    i
}
```

Listing 6: Example demonstrating why predicates and mutable values may cause problems

5.1.4. Mutable References

Besides the mutation of values, mutable references are also common in Rust, which is underpinned by the analysis, which found more than 10% of parameters to be mutable.

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 (e.g. Kloos et al. [15]), 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)

The problem with mutable references is the possibility of aliasing. Aliasing is problematic, because it might create interdependencies between the types of different variables: If one variable is changed, it might affect wether another variable is typed correctly. In listing 7 the return value `a` is affected by changes done to a different variable `b`. Conservative approximation requires that all possible effects must be tracked.

References are tracked by recognizing the fact that it is sufficient to let the reference's type solely constrain the reference target, while the value owner's type constrains the value. In the example, `b = &mut a` will have inferred type `{ r : &mut | r == & a }`, meaning any inhabitant of this type can at most refer to the variable `a`[2]. 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`, meaning no other value predicate can be affected by the change.

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.

To also support these use cases, Corten features supports for weak updates, which can not change types, but allow assigning to ambiguous reference destinations. Listing 8

---

[2] 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. This why no new construct needs to be introduced

```
fn client() -> ty!{ v: i32 | v == 4 } {
    let a = 2;        // a : {v1 : i32 | v1 == 2}
    let b = &mut a;   // b : {v2 : &i32 | v2 == &a}
    *b = 0; // changes a's value and type
    let c = &mut b;   // c : {v3 : &i32 | v3 == &b}
    **c = 4; // changes a's value and type
    a
}
```

Listing 7: Example demonstrating interdependencies between mutable references

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.

```
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 i32 | r == &y || r == &z };
                  // branches of if need same type => weaken
  } else {
    res = &mut z as ty!{ r : &mut i32 | r == &y || r == &z };
  }
  *res = x + 11; // res could refer to b or c
                 // -> assigned value must satisfy both types
  y + z
}
```

Listing 8: Example demonstrating weak updates

An intricacy about conservative reference tracking are functions that return references. In Rust this is possible, if the reference was passed as an argument: The function signature `fn f(a : &mut T, b : &mut T) -> &mut T` would allow `f` to return either a reference to `a` or `b`. For the callee knowing where the returned value points to, is important because the type of dereferencing it depends on that knowledge. Consequently, for the return type to be conservative, it must state every possible reference destination. Our approach naturally can be naturally extended to covert this problem: If the callee does not specify the target (i.e. the return type is $\{v : \texttt{T} \mid \text{true}\}$), then any assignment to it would need to show, that it satisfies the predicate of all possible targets, which in this case would be every target, so no assignment to the return type is possible. Conversely, if the callee forgot a possible reference target in the return type, then type checking of the body would fail, because the actual (supposedly weaker) type (e.g. $\{r : \&\texttt{mut T} \mid r \doteq \&y\}$) does not imply the required (supposedly stronger) type (e.g. $\{r : \&\texttt{mut T} \mid r \doteq \&y \lor r \doteq \&z\}$).

5.1.5. 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 9 shows, how an incrementing function `inc` can be specified: `inc` signature stipulates that the function can be called with a mutable reference to any `i32` whose reference target (given the logic variable name `a1`) satisfies the predicate `true` and will update it to the value `a2`, which equals `a1 + 1`. Only the signature of `inc` is considered for the type checking `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 | true => 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 9: Example showing how Corten allows for accurate type checking in the presence of function calls

5.1.6. Atomic Updates

Complex mutation patterns can result in complex interdependencies. We deem it necessary to allow different types to refer to each other. The properties of arguments might depend on each other, consequently their refinement types should be able to refer to each other.

The listing 10 is extracted from the evaluation example 7.3. It requires establishing an invariant, where `i` has the predicate ≥ 0 and `sum` has the predicate $= i \cdot n$.

Attempting to update one type at a time, will not yield the desired result. If `i` is updated first, then it is not possible to establish `sum`'s type, because `i`'s type is no longer expressive enough. If `sum` is updated first, we can establish its type, but once `i` is updated, it will refer to a different logic variable for `i` than `sum`. If we tried to carry the information that the two logic variables are the same in the type, it would no longer be an invariant, because reassigning the value would invalidate it[3].

---

[3] For this use-case, it might also be possible to relax a single variable at a time, by renaming the logic variables to their old names after the subtyping relation is shown. This approach was not chosen, because it may only cover some special cases.

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

relax_ctx!{
  i: ty!{ iv : i32 | iv > 0  },
  sum: ty!{ sv: i32 | 2 * sv == iv * (iv + 1) }
}

i = i + 1;
sum = sum + i;
```

Listing 10: Example Demonstrating Interdependence between Types

For this purpose, Corten provides a mechanism for the programmer to instruct Corten to relax all variables at once named `relax_ctx`. The syntax is similar to the refinement type syntax, but allows for multiple variables to be described at a time. The type system will check all refinement types at once, resulting in the successful type checking of the example.

5.2. Type System

The following section will introduce the type checking rules of Corten. We assume the program has passed the Rust type checking rules, meaning (non-refined) type- and ownership-checking was successful.

Definition 5.2.1 (Typing Context $\Gamma = (\mu, \Phi)$). The typing context consist of an injective function $\mu : \text{PVar} \to \text{LVar}$ that is used for tracking the current logic variable associated with a program variables and a set of predicates Φ. It is used to constrain the logic variables in the image of μ and holds the relevant varying context for the various type checking rules. μ must contain a logic variable for every program variable and Φ can contain additional logic variables not found in the image of μ.

For $\Gamma = (\mu, \Phi)$ we abbreviate $(\mu, \Phi \wedge \varphi)$ to Γ, φ for the addition of φ to the constraint set, $(\mu[x \mapsto \alpha], \Phi)$ to $\Gamma[x \mapsto \alpha]$ for the update of the mapping for program variable x to α and $\mu(x)$ to $\Gamma(x)$ for accessing to the logic variable associated with x.

We start with the rule for checking function declarations. Without loss of generality, we assume arguments are ordered by their type: First immutable / owned arguments and then mutable arguments.

Definition 5.2.2 (Function Declaration Type Checking). When handling mutable references in parameters some subtleties need to be considered. To describe the referenced value is normally done using the $\{\alpha \mid \alpha \doteq \&y\}$ 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. Also for an argument x_i, the function could change both the referenced *value* (`*i = 2`) as well as the reference *target* (`i = &mut y`). As seen in section 5.1.4, using the dereference operator would come with a lot of complications. Instead we introduce arg^i, a special variable denoting the initial abstract value (i.e. stack location) that the mutable reference of argument n points to. For the jth parameter with the Rust type `&mut i32` we would generate pseudo location arg_j.

$$\textsc{Fn-Decl}\ \dfrac{\begin{array}{c}\mu_{init} = \{x_i \mapsto \alpha_i, y_j \mapsto \delta_j, arg_j \mapsto \beta_j\} \qquad \Phi_{init} = \varphi_i, \psi_j, \delta_j \doteq \& arg_j \\ (\mu_{init}, \Phi_{init}) \vdash s \Rightarrow \Gamma' \\ \Gamma' \vdash x_r : \tau_r \qquad \Gamma' \vdash \tau_r \leq \tau \qquad \Gamma' \leq (\{arg_j \mapsto \gamma_j\}, \chi_j)\end{array}}{\texttt{fn}\ f(x_i : \{\alpha_i : b_i \mid \varphi_i\}, y_j : \&\texttt{mut}\ \{\beta_j : b_j \mid \psi_j \Rightarrow \gamma_j \mid \chi_j\}) \rightarrow \tau\{s; x_r\}}$$

The first two antecedents initialize the context, which contains the preconditions from the function signature ϕ_i and ψ_j and the pseudo variables for the mutable arguments arg_j, which are constrained to be the reference destinations for y_j. The next antecedent stipulates that the execution of the body s updates the context to Γ'. The last three terms are concerned with ensuring, that the asserted postconditions for the return type as well as the context update hold. The rules will be introduced in the following definitions.

Next, we introduce the rule for checking the subtype relation, which is introduced by the function return type check or by the user / future inference system as part of an expression.

Definition 5.2.3 (Sub-Typing Rule: $\Gamma \vdash \tau \leq \tau'$). A type τ is a subtype of τ' in the context Γ, if the predicate of the supertype implies the predicate of the subtype, with the supertype's logic variable substituted for the subtype's variable to the predicate so that they refer to the same variable. The validity check performed in the decidable, propositional logic with the theories of equality and linear arithmetic. Corten uses an SMT solver for deciding these requests.

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

Note that, $\Gamma \vdash \tau \leq \tau'$ can be rephrased in terms of $\Gamma \leq \Gamma'$ by introducing a fresh variable with the predicates τ and τ'. The rephrased form is equivalent, but rephrasing the sub context rule in terms of sub-typing is in general not possible: Because all constraints need to be satisfied at the same time when checking for sub-contexts, just checking each variable at a time would be a weaker proposition.

Definition 5.2.4 (Sub-Context Rules: $\Gamma \leq \Gamma'$). In contrast to other refinement type systems, Corten allows types in the context to refer to one another. For example the type specifications `a : { α : i32 | α > β}, b: { β: i32 | β != α}` are considered well formed in Corten and result in the context $\Gamma = (\{\texttt{a} \mapsto \alpha, \texttt{b} \mapsto \beta\}, \alpha \neq \beta \land \beta \geq \alpha)$. In the example, the value 0 is a valid inhabitant of `a`'s type as long as $\texttt{b} \geq 1$.

The rule $\leq$-Ctx serves the purpose of allowing these interdependencies between predicates while still providing an applicable concept of subtyping: As long as the set of predicates in Γ' imply the predicates for Γ, the sub-context relation is satisfied and Γ' and Γ is called the super-context and sub-context respectively. To be more precise, the rule needs to relate the predicates for each *program* variable with the opposing predicates for that program variable. The substitution $\Phi'[\mu'(x) \triangleright \mu(x) \ \mid x \in \mathrm{dom}(\mu)]$ forces the logic variable associated with x in Γ' to be substituted by the logic variable $\mu(x)$, which is associated with x in Γ'.

The domain restriction ensures that the sub-context does not contain additional program variables, which would leave unconstrained variables in the antecedent.

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

Example 5.2.1 (Sub-Context Relations). *To get a idea of how the sub-context relation works, consider the following examples:*

$$\begin{aligned}
(\{a \mapsto \alpha\}, \alpha \doteq 2) &\preceq (\{a \mapsto \beta\}, \beta > 0) \\
(\{a \mapsto \alpha\}, \alpha > 0, \beta \doteq 4) &\preceq (\{a \mapsto \alpha\}, \alpha \doteq 5) \\
(\{a \mapsto \gamma, b \mapsto \delta\}, \gamma \doteq 0, \delta \geq 1) &\preceq (\{a \mapsto \alpha, b \mapsto \beta\}, \alpha \neq \beta \land \beta \geq \alpha) \\
(\{\}, \mathtt{true}) &\preceq \Gamma' \quad \text{for all } \Gamma' \\
\Gamma &\preceq (\{\}, \mathtt{false}) \quad \text{for all } \Gamma
\end{aligned}$$

Definition 5.2.5 (Fresh Variable $\Gamma \vdash \alpha$ fresh). Corten requires that new logic variables are distinct from all others. The rule ensures that α is a free variable in $\Gamma = (\mu, \Phi)$:

$$\Gamma \vdash \alpha \text{ fresh iff } \alpha \notin FV(\Phi) \cup \mathrm{img}(\mu)$$

Definition 5.2.6 (Expression Constraint $\alpha \simeq \llbracket e \rrbracket \Gamma$). An expression constraint is used to restrict α to (as close to) the value of expression e. For instance, when typing the expression `b + c`, $\alpha \simeq \llbracket a + b \rrbracket \Gamma$ will generate the strongest type $\alpha \doteq \beta + \gamma$ where α, β are the logic variable for a, b in Γ.

$$\begin{aligned}
\alpha \simeq \llbracket v \rrbracket \Gamma &= \alpha \doteq v \\
\alpha \simeq \llbracket x_1 \odot x_2 \rrbracket \Gamma &= \alpha \doteq (\Gamma(x_1) \odot \Gamma(x_2)) && \text{if } \odot \text{ allowed} \\
\alpha \simeq \llbracket x \rrbracket \Gamma &= \alpha \doteq \Gamma(x) \\
\alpha \simeq \llbracket \& y \rrbracket \Gamma &= \alpha \doteq \& y \\
\alpha \simeq \llbracket e \rrbracket \Gamma &= \text{true} && \text{otherwise}
\end{aligned}$$

In some cases, it may not be possible to generate the strongest type in the predicate logic. Take for instance the expression `x * y`, which is not admissible in EUFA. In these cases, the expression will be conservatively approximated by the predicate `true`.

Definition 5.2.7 (Expression Typing: $\Gamma \vdash e : \tau$).

$$\textsc{Lit}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash v : \{\alpha : b \mid \alpha \simeq \llbracket v \rrbracket \Gamma\}} \qquad \textsc{Add}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash x_1 \odot x_2 : \{\alpha : b \mid \alpha \simeq \llbracket x_1 \odot x_2 \rrbracket \Gamma\}}$$

$$\textsc{Var}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash x : \{\alpha : b \mid \alpha \simeq \llbracket x \rrbracket \Gamma\}} \ \textsc{Ref}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash \& x : \{\alpha : \& b \mid \alpha \simeq \llbracket \& x \rrbracket \Gamma\}}$$

$$\textsc{Var-Deref}\ \frac{\Gamma \vdash *x \in \{y\} \qquad \Gamma \vdash y : \tau}{\Gamma \vdash *x : \tau} \qquad \textsc{Intro-Sub}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash e \ \mathtt{as}\ \tau' : \tau'}$$

Because we assume that the base language type checking already accepted the program, the expression typing rule become quite simple. The rules Lit, Var, Add, Ref are solely concerned with delegating to the expression constraint generation. Var-Deref stipulates that dereferencing `*x` has type τ if the target of `x` is known and unique. Intro-Sub allows for a subtype to be introduced, which would mostly be used internally by an inference system.

Definition 5.2.8 (Statement Type Checking $\Gamma \vdash s \Rightarrow \Gamma'$).

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

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

$$\textsc{Seq}\ \dfrac{\Gamma \vdash s_1 \Rightarrow \Gamma' \qquad \Gamma' \vdash s_2 \Rightarrow \Gamma''}{\Gamma \vdash s_1; s_2 \Rightarrow \Gamma''} \quad \textsc{Relax}\ \dfrac{\Gamma \preceq \Gamma'}{\Gamma \vdash \texttt{relax\_ctx!}\{\Gamma'\} \Rightarrow \Gamma'}$$

$$\textsc{Decl}\ \dfrac{\Gamma \vdash e : \{\beta : b \mid \varphi\}}{\Gamma \vdash \texttt{let}\ \ x = e \Rightarrow \Gamma[x \mapsto \beta], \varphi} \qquad \textsc{Assign}\ \dfrac{\Gamma \vdash e : \{\beta : b \mid \varphi\}}{\Gamma \vdash x = e \Rightarrow \Gamma[x \mapsto \beta], \varphi}$$

$$\textsc{Assign-Strong}\ \dfrac{\Gamma(z) = \beta \qquad \Gamma \vdash *x \in \{y\} \qquad \Gamma \vdash \gamma \text{ fresh}}{\Gamma \vdash *x = z \Rightarrow \Gamma[y \mapsto \gamma], \gamma \doteq \beta}$$

$$\textsc{Assign-Weak}\ \dfrac{\begin{array}{cc} \Gamma \vdash e : \tau & \Gamma \vdash *x \in \{y_1, \dots, y_n\} \\ \overline{\Gamma \vdash y_i : \{\beta_i : b_i \mid \varphi_i\}} & \overline{\Gamma \vdash \tau \preceq \{\beta_i : b_i \mid \varphi_i\}} \end{array}}{\Gamma \vdash *x = e \Rightarrow \Gamma}$$

$$\textsc{Fn-Call}\ \dfrac{\begin{array}{c} (\mu, \Phi) \preceq (\mu, \Phi \wedge (\overline{\varphi_i} \wedge \overline{\psi_j}) \overline{[\alpha_i \triangleright \mu(x_i)]}\ \overline{[\beta_j \triangleright \mu(y_j)]} \\ \Sigma \vdash f : \texttt{fn}(\overline{u_i : \{\alpha_i : b_i \mid \varphi_i\}}, \overline{v_j : \{\beta_j : b_j \mid \psi_j \Rightarrow \beta_j' \mid \psi_j'\}}) \rightarrow \{\gamma : b \mid \rho\} \end{array}}{\begin{array}{l} (\mu, \Phi) \vdash \texttt{let}\ \ r = f(\overline{x_i}, \overline{\&\texttt{mut}\ \ y_j}) \Rightarrow \\ \qquad (\mu[\overline{y_j \mapsto \beta_j'}, r \mapsto \gamma], \Phi \wedge (\overline{\varphi_i} \wedge \overline{\psi_j} \wedge \overline{\psi_i} \wedge \rho) \overline{[\alpha_i \triangleright \mu(x_i)]}) \end{array}}$$

The rule If models the path sensitivity for branches of the statement and requires that both branches have the same return context, which forces the branches to express their effects using the same context. If an inference system was added to Corten, it would add context relaxation statements to the end of the branches to automate the process.

The While implements a loop invariant rule: Type checking of a while loop requires that the body invariant Γ_I is preserved by the body, meaning the body end execution with a context Γ_I' that does not conflict with Γ_I and ensures that the context after execution is Γ_I with the addition of the path condition.

Relax allows the user to introduce a sub-context constraint; Decl and Assign append the context with the predicate of the value and update (or append for the declaration) the logic variable association for the program variable x.

For handling references, Corten offers the rules Assign-Strong and Assign-Weak with the first implementing strong updates, changing the type of the reference target y, if it can be shown that x must refer to y. The second rule allows updates to the value only if the type is not affected by it. This is ensured by demanding that the assigned type τ is at least as specific as the type of every possible reference destination (in the current context Γ).

Fn-Call handles function calls by requiring the current context to be a sub-context of the function entry predicates φ_i for values and immutable references and ψ_j for mutable references, where the logic variables of the parameters are substituted by argument's logic variables. The rule states that resulting context will be the starting context with updated the association for the mutated arguments y_j and new association for r and the addition of the predicates the function ensures to be true after execution. Because these predicates can use the variables from the immutable values, we also need to substitute the logic variable.

For clarity, Fn-Call omits a renaming step done directly after extracting the function signature from the declaration context Σ. It ensures that the logic variable names from the function call signature do not conflict with the any variable in the current context by renaming them if necessary. An an example take section 7.2: If the variables were not renamed, the second call to `fib(n2)` would conflict with the first.

```
...                                               fn receiver(
let x = 4;                                          a: ty!{ a1:i32 | a1 > 0 },
let mut y = 5;                                      b : &mut ty!{ b1 : i32 | true
                                                    => b2 | b2 > b1 + a1 })
                                                  {
// Γ_1 = ({x ↦ v_1, y ↦ v_2}, v_1 ≐ 4 ∧ v_2 ≐ 5)      // Γ_init = ({a ↦ a_1, b ↦ b_1}, a_1 > 0 ∧ true)
relax_ctx!{ /* Γ_init */ } /*----------->*/         assume_ctx!{ /* Γ_init*/ }

fn receiver(x, &mut y);                             ...

                                                    // Γ_r = ({a ↦ a_1, b ↦ b_2}, a_2 > 0 ∧ true)
assume_ctx!{ /* Γ_r */ } /*<-----------*/           relax_ctx! { /* Γ_r */ }
...                                               }
```

Listing 11: Example Showing the Correspondence between Function Calls and Context Assumptions

A different intuition for function calls handling in Corten may also gained by understanding calls in terms of assumptions `assume_ctx!` and assertions `relax_ctx!` that Corten replaces a function call with. Listing 11 shows how this might look like: The caller abstracts the concrete call by requiring the current context Γ_1 to satisfy the initial context Γ_{init} of the callee (modulo renaming). In turn, the callee assumes its initial context to be satisfied and proceeds to modify it, with the end of the function requiring the function body to satisfy the specified post context Γ_r. Consequently, the caller can assume the final context to be valid (with the addition of the old context of course). As a whole, this creates an unbroken chain of assumes and requires and thusly a sensible design.

Definition 5.2.9 (Reference Destination Judgement $\Gamma \vdash *x \in \{\ldots\}$). To complete the type system, two more judgements need to be defined. In the statement and expression typing rules, it is necessary to gain some information about the reference target. The rule May-Ref ensures that the set $\{y_1, \dots, y_n\}$ contains all possible reference targets. The special case $\{y\}$ implies, that x must reference exactly y, because a reference without a target is impossible.

$$\textsc{May-Ref}\ \frac{\mu(x) = \alpha \qquad (\beta \doteq \&y_1 \vee \dots \vee \beta \doteq \&y_n) \in \Phi \Downarrow \alpha}{(\mu, \Phi) \vdash *x \in \{y_1, \dots, y_n\}}$$

$\Phi \Downarrow \alpha$ computes the set of equivalent values in Γ's constraint set. The function satisfies the three properties of equivalence classes: $\alpha \in \Phi \Downarrow \alpha$, $\alpha \in \Phi \Downarrow \beta \Leftrightarrow \beta \in \Phi \Downarrow \alpha$ and $\alpha \in \Phi \Downarrow \beta \wedge \beta \in \Phi \Downarrow \gamma \implies \gamma \in \Phi \Downarrow \gamma$. This property holds because of the limited expressiveness of reference predicates, which can only directly specify equality with another logic variable or a reference target constraint, which ends the chain. Reference predicates can not be nested or further constrained by other predicates. The equivalence set is constructed by iteratively computing the fixpoint $\beta \in \Phi \Downarrow \alpha \wedge (\dots \wedge \beta \doteq \gamma \wedge \dots) = \Gamma \implies \gamma \in \Phi \Downarrow \alpha$. This is safe because each iteration monotonically appends to the set and the set of formulae and variables is finite. Well-formedness requires that there to be exactly one reference target constraint. Thus the reference constraint is unique and can be inferred.

5.3. Soundness of the Type System

Next, we will to justify why Corten type system is safe.

We will write substitution as $\Phi[x \triangleright a]$ meaning x is replaced by a in Φ and $\mu[x \mapsto \alpha]$ for the function update meaning $\mu[x \mapsto \alpha](x) = \alpha$ and $\mu(y)$ everywhere else.

To reduce the verbosity of the notation, for $x_1 \dots x_n \in \mathrm{dom}(\mu)$ we will abbreviate we shorten $\Phi[\mu(x_1) \triangleright g(x_1), \dots, \mu(x_n) \triangleright g(x_n)]$ to $\Phi[f(x_i) \triangleright g(x_i) \mid x]$ and $\Phi[\mu(x_i) \triangleright g(x_i) \mid x \neq y]$ if $x \in \{x \mid x \in \mathrm{dom}(\mu), x \neq y\}$

Definition 5.3.1 (State Conformance). A state σ is conformant with respect to a typing context $\Gamma = (\mu, \Phi)$ (written as $\sigma : \Gamma$), iff:

$$\Phi[\mu(x) \triangleright [\![\sigma(x)]\!] \mid x \in \mathrm{dom}(\mu)] \text{ is satisfiable}$$

I.e. a conformant type context does not contradict the execution state.

The state conformance rule is quite different to a standard state conformance rule. It is owed to the fact that firstly, the context contains formulae and secondly, the context may contain free variables, which are not bound by any variable in σ. To get an idea of the behaviour of this definition consider the following properties:

- If $\sigma : (\emptyset, \Phi)$ then Φ is satisfiable
- If $\sigma : (\mu, \Phi_1 \wedge \Phi_2)$ then $\sigma : (\mu, \Phi_1)$ and $\sigma : (\mu, \Phi_1)$.
- If $\sigma : (\mu, \Phi)$ and $\mathrm{FV}(\Phi) \subseteq \mathrm{dom}(\mu)$, then $\vDash \Phi[\mu(x) \triangleright [\![\sigma(x)]\!] \mid x \in \mathrm{dom}(\mu)]$

Especially interesting is the last one, which follows from the fact, by definition a Φ without any free variables, will have all variables replaced by constants, making the entire predicate constant. Because the definition of state conformance was satisfied for some m it does not depend on, Φ is also valid.

With the notion of conformance established, next we will introduce our definition of safety. As usual for a small-step semantic, we will consider progress and preservation based on Wright and Felleisen [34]. We will assume, that the base language (e.g. Rust) satisfies the safety properties and show that relative to the base language Corten also satisfies the type safety properties.

Definition 5.3.2 (Progress). If $\Gamma \vdash s_1, \sigma : \Gamma \Rightarrow \Gamma_2$ and $s_1 \neq \texttt{unit}$, then there is a s_2 and σ_2 with $\langle s_1 \mid \sigma_1 \rangle \rightsquigarrow \langle s_2 \mid \sigma_2 \rangle$.

Definition 5.3.3 (Preservation). If $\Gamma \vdash s \Rightarrow \Gamma_2, \sigma : \Gamma$ and $\langle s \mid \sigma \rangle \rightsquigarrow \langle s_1 \mid \sigma_1 \rangle$, then there is a Γ_1 with $\Gamma_1 \vdash s_1 \Rightarrow \Gamma_2$ and $\sigma_2 : \Gamma_2$

Since Corten's types do not influence the execution semantics of the program and conformance for the base types is assumed, progress not interesting.

In contrast, preservation requires us to show that conformance of σ to Corten's (refined) type context is preserved by execution. The proof sketch is structured as follows: We start by showing in lemma 5.3.4, that the evaluation of expressions preserves state conformance (although expressed slightly unusual way). Next lemma 5.3.5 we argue for why reference tracking is conservative, followed by lemma 5.3.6 which demonstrates that our notion of sub-contexts is in deed conservative. Using these arguments, lemma 5.3.7 will show, that preservation holds for the type system.

Lemma 5.3.4 (Conformance of Symbolic Execution). *If* $\sigma : \Gamma, \Gamma \vdash \alpha$ *fresh then* $\sigma[x \mapsto [\![e]\!]\sigma] : \Gamma[x \mapsto \alpha], (\alpha \simeq [\![e]\!]\Gamma)$

Proof. Let $\Gamma = (\Phi, \mu)$. Given $\Phi[\mu(x) \triangleright [\![\sigma(x)]\!] \mid x]$ SAT show $(\Phi \land \alpha \simeq [\![e]\!]\Gamma)[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!]]$. Let $\sigma' = \sigma[x \triangleright [\![e]\!]\sigma]$. As a model we select $m' = m[\beta \triangleright \sigma(x)]$ where $\mu(x) = \beta$ and show that the proof obligation is satisfied for m'.

Split the goal on the conjunction; if both sides are valid then the conjunction is valid (for the same model).

- Part I Show $m' \models \Phi[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y]$. In essence, we show that the satisfiability of Φ is not changed by the addition of fresh variable α. We start by splitting the substitution into the part of concerned with x and a part that is not $(x \neq y)$. We then introduce a useless substitution for α that can not change the formula because it does not contain α (and the substitution on the left explicitly excludes x). After some bookkeeping, we use the fact that substitution in expression can be rephrased as a function update on the model m to arrive at our goal.

$$\begin{aligned} &\Phi[\mu(y) \triangleright [\![\sigma(y)]\!] \mid y] \\ &= \Phi[\mu(y) \triangleright [\![\sigma(y)]\!] \mid y \neq x][\mu(x) \triangleright [\![\sigma(x)]\!]] \\ &= \Phi[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y \neq x][\alpha \triangleright [\![e]\!]\sigma][\mu(x) \triangleright [\![\sigma(x)]\!]] \\ &= \Phi[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y \neq x][\mu[x \mapsto \alpha](x) \triangleright [\![\sigma'(x)]\!]][\mu(x) \triangleright [\![\sigma(x)]\!]] \\ &= \Phi[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y][\mu(x) \triangleright [\![\sigma(x)]\!]] \end{aligned}$$

$$\begin{aligned} m &\vDash \Phi[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y][\mu(x) \triangleright [\![\sigma(x)]\!]] \\ &\text{implies} \\ \underbrace{m[\mu(x) \mapsto [\![\sigma(x)]\!]]}_{=m'} &\vDash \Phi[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y] \end{aligned}$$

- Part II show $m' \vDash (\alpha \simeq [\![e]\!]\Gamma)[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y]$ In part two we need to show that our symbolic execution does not conflict with σ. The problematic case is handling of reassignments to a variable. In that case, the old logic variable becomes unbound, forcing us to show that we can choose a appropriate model. A multitude of case distinctions ensures none of these cases are missed.

  Case distinction on $\alpha \simeq [\![e]\!]\Gamma$

  - Case true show $m' \vDash \text{true}[\dots]$; trivial
  - Case constraint show:

    $$m' \vDash \underbrace{\alpha[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y]}_{[\![\sigma(x)]\!] = [\![e]\!]\sigma} \doteq [\![e]\!]\Gamma[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y]$$

    - Case Constant $e = v$ trivial
    - Case Variable $e = z$
      - Case $e = z = x$

        $$\begin{aligned} &\Gamma[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y] \\ &= \mu(x)[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y] \\ &= \mu(x)[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y \neq x][\alpha \triangleright [\![\sigma'(x)]\!]] \\ &\overset{\alpha \text{ fresh}}{=} \mu(x) \overset{\text{def. } m'}{=} [\![\sigma(x)]\!] \end{aligned}$$

      - Case $e = z \neq x$

        $$\begin{aligned} &[\![z]\!]\Gamma[\mu[x \mapsto \alpha](y) \triangleright [\![\sigma'(y)]\!] \mid y] \\ &= \mu(z)[\mu[x \mapsto \alpha](z) \triangleright [\![\sigma'(z)]\!]] \\ &\overset{\text{def. } z}{=} \mu(z)[\mu(z) \triangleright [\![\sigma'(z)]\!]] \\ &= [\![\sigma(z)]\!] = [\![z]\!]\sigma \end{aligned}$$

- Case Binary Operation $e = x_1 \odot x_2$
  - Case $x_1 = x$, $x_2 \neq x$ (without loss of generality)

$$\begin{aligned}
&\llbracket x \odot x_2 \rrbracket [\mu[x \mapsto \alpha](y) \triangleright \llbracket \sigma'(y) \rrbracket \mid y] \\
&= \llbracket x \odot x_2 \rrbracket \Gamma [\mu(x_2) \triangleright \llbracket \sigma'(x_2) \rrbracket] [\mu[x \mapsto \alpha](x) \triangleright \llbracket \sigma'(x) \rrbracket] \\
&= \llbracket \mu(x) \odot \mu(x_2) \rrbracket [\mu(x_2) \triangleright \llbracket \sigma'(x_2) \rrbracket] [\alpha \triangleright \llbracket \sigma'(x) \rrbracket] \\
&= \llbracket \mu(x) \odot \mu(x_2) \rrbracket [\mu(x_2) \triangleright \llbracket \sigma(x_2) \rrbracket] \\
&= \llbracket \sigma(x) \odot \sigma(x_2) \rrbracket \underset{\mathit{Inv.}}{\overset{\mathit{Rule}}{=}} \llbracket e \rrbracket \sigma
\end{aligned}$$

  - Case $x_1 \neq x, x_2 \neq x$, **Case** $x_1 = x, x_2 = x$ and **Case** $x_1 \neq x, x_2 = x$ analogous

□

The next lemma not only shows that predicate types can not have no existential bindings, but also – crucially – that the typing assigned to mutable references is conservative, in the sense that it covers all possible reference destinations.

Lemma 5.3.5 (Reference Predicate are Conservative). *If* $\sigma : \Gamma$ *and* $\Gamma \vdash *x \in \{y_1, \ldots, y_n\}$ *then* $\llbracket \sigma(x) \rrbracket = \&y_i$ *for some* $i \in 1, \ldots, n$

Proof Sketch. Rule inversion of the May-Ref rule yields $\mu(x) = \alpha$ (I) and $\Gamma \Downarrow \alpha \ni \{y_1, \ldots, y_n\}$. Let $\varphi_\alpha = (\alpha \doteq \&y_1 \vee \cdots \vee \alpha \doteq \&y_n)$. By using the definition of the $\Gamma \Downarrow \alpha$ we can show that $\varphi_\alpha[\mu(x) \triangleright \llbracket \sigma(x) \rrbracket \mid x \in \mathrm{dom}(\mu)]$ (II) is satisfiable, because φ_α is less restrictive than Φ (being directly constructed from a subset of terms in a conjunction) and Φ already satisfies the state conformance property.

Keep in mind that φ_α has no free variables wrt. μ as it only refers to the logic variable α, which by assumption (I) is not free and program variables y_i, which as – as far as the logic is concerned – mere constants (III). The argument from the state conformance examples also applies here: If we can show a formula to be independent from the rest and the model, then showing satisfiability for a model that φ_α does not depends on, yields validity (IV).

Therefore, for some m:

$$\begin{aligned}
&\quad m \vDash \Phi[\mu(z) \triangleright \llbracket \sigma(z) \rrbracket \mid z \in \mathrm{dom}(\mu)] \\
&\overset{\text{(II)}}{\Longrightarrow} m \vDash \varphi_\alpha[\mu(z) \triangleright \llbracket \sigma(z) \rrbracket \mid z \in \mathrm{dom}(\mu)] \\
&\Longrightarrow m \vDash \varphi_\alpha[\mu(x) \triangleright \llbracket \sigma(x) \rrbracket][\mu(z) \triangleright \llbracket \sigma(z) \rrbracket \mid z \in \mathrm{dom}(\mu) \setminus \{x\}] \\
&\overset{\text{(III)}}{\Longrightarrow} m \vDash \varphi_\alpha[\mu(x) \triangleright \llbracket \sigma(x) \rrbracket] \\
&\Longrightarrow m \vDash (\alpha \doteq \&y_1 \vee \cdots \vee \alpha \doteq \&y_n)[\mu(x) \triangleright \llbracket \sigma(x) \rrbracket] \\
&\overset{\text{(I)}}{\Longrightarrow} m \vDash (\alpha \doteq \&y_1 \vee \cdots \vee \alpha \doteq \&y_n)[\alpha \triangleright \llbracket \sigma(x) \rrbracket] \\
&\overset{\text{(IV)}}{\Longrightarrow} \vDash (\llbracket \sigma(x) \rrbracket \doteq \&y_1 \vee \cdots \vee \llbracket \sigma(x) \rrbracket \doteq \&y_n) \\
&\Leftrightarrow \llbracket \sigma(x) \rrbracket \doteq \&y_i \text{ for some } i \in 1, \ldots, n
\end{aligned}$$

□

Lemma 5.3.6 (Sub-Context Relation is Conservative). *If* $\Gamma \preceq \Gamma'$ *and* $\sigma : \Gamma$ *then* $\sigma : \Gamma'$

Proof. Given $\Gamma \preceq \Gamma'$, $\sigma : \Gamma$ show $\sigma : \Gamma'$. Let $x \in \mu$ denote $x \in \mathrm{dom}(\mu)$ and v_x denote $[\![\sigma(x)]\!]$.

We need to show that for a given model of $\sigma : \Gamma$ we can find a model, that satisfies $\sigma : \Gamma'$. We use the preconditions of the Sub-Ctx rule, which yields an implication relation between the type contexts (5.1) and the fact that only some variables, namely $\mathrm{dom}(\mu') \setminus \mathrm{dom}(\mu)$, are fresh and therefore unconstrained by Γ and therefore these variables can be chosen to suit our needs: The logic variables that become unbound in Γ' relative to Γ are instead constrained to the required value by the chosen context m. Also recall that μ is injective.

Rule inversion on $\Gamma \preceq \Gamma'$ yields:

$$m \vDash \Phi'[\mu'(x) \triangleright \mu(x) \mid x \in \mu] \longrightarrow \Phi \quad \text{forall } m \tag{5.1}$$

$$\mathrm{dom}(\mu') \subseteq \mathrm{dom}(\mu) \tag{5.2}$$

$$\sigma : \Gamma \implies m^\star \vDash \Phi[\mu(x) \triangleright v_x \mid x \in \mu] \quad \text{for some } m^\star \tag{5.3}$$

$\mathrm{dom}(\mu') \setminus \mathrm{dom}(\mu)$ is fresh wrt. Γ

$$\implies m^\star[\mu'(x) \mapsto v_x \mid x \in \mu' \setminus \mu] \vDash \Phi[\mu(x) \triangleright v_x \mid x \in \mu] \tag{5.4}$$

$$\implies m^\star[\mu'(x) \mapsto v_x \mid x \in \mu' \setminus \mu][\mu(x) \mapsto v_x \mid x \in \mu] \vDash \Phi \tag{5.5}$$

Let $m = m^\star[\mu'(x) \mapsto v_x \mid x \in \mu' \setminus \mu]$

$$\implies m[\mu(x) \mapsto v_x \mid x \in \mu] \vDash \Phi \tag{5.6}$$

eq. (5.1) for $m[\mu(x) \mapsto v_x \mid x \in \mu]$

$$\implies m[\mu(x) \mapsto v_x \mid x \in \mu] \vDash \Phi'[\mu'(x) \triangleright \mu(x) \mid x \in \mu] \longrightarrow \Phi \tag{5.7}$$

Using eq. (5.6):

$$\implies m[\mu(x) \mapsto v_x \mid x \in \mu] \vDash \Phi'[\mu'(x) \triangleright \mu(x) \mid x \in \mu] \tag{5.8}$$

$$\implies m \vDash \Phi'[\mu'(x) \triangleright \mu(x) \mid x \in \mu][\mu(x) \triangleright v_x \mid x \in \mu] \tag{5.9}$$

Composition of the substitutions[4]: $\ldots [a \triangleright b][b \triangleright c]$ to $\ldots [a \triangleright c]$

$$\implies m \vDash \Phi'[\mu'(x) \triangleright v_x \mid x \in \mu] \tag{5.10}$$

$$\overset{Def.m}{\implies} m^\star[\mu'(x) \mapsto v_x \mid x \in \mu' \setminus \mu] \vDash \Phi'[\mu'(x) \triangleright v_x \mid x \in \mu] \tag{5.11}$$

$$\implies m^\star \vDash \Phi'[\mu'(x) \triangleright v_x \mid x \in \mu][\mu'(x) \triangleright v_x \mid x \in \mu' \setminus \mu] \tag{5.12}$$

$$\implies m^\star \vDash \Phi'[\mu'(x) \triangleright v_x \mid x \in \mu'] \tag{5.13}$$

$$\Leftrightarrow \sigma : \Gamma' \tag{5.14}$$

□

Lemma 5.3.7 (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

To ensure that the proof structure is correct, this proof as well as the type rules and some other lemmata were also partially formalized in the theorem prover Lean[22]. Most branches were also proven with Lean[5].

Proof Sketch. Rule Induction over $\langle s \mid \sigma\rangle \rightsquigarrow \langle s_1 \mid \sigma_1\rangle$

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

  Rule inversion over $\Gamma \vdash e : \{\alpha : b \mid \varphi\}$:

  - Cases with Symbolic Execution (Lit , Add , Var , Ref): rule inversion yields $\Gamma \vdash \alpha$ fresh and $\Gamma_2 = \Gamma[x \mapsto \alpha], \alpha \simeq [\![e]\!]\Gamma$

    With $\Gamma_1 = \Gamma_2$, the goal is: $\sigma[x \mapsto [\![e]\!]\sigma] : (\Gamma[x \mapsto \alpha], \alpha \simeq [\![e]\!]\Gamma)$; lemma 5.3.4 implies the goal.

  - Case Var-Deref $e = *y$: rule inversion yields $\Gamma \vdash *y : \{z\}, \Gamma \vdash z : \{\gamma : b \mid \varphi_\gamma\}$ and $\Gamma_2 = \Gamma[x \mapsto \gamma], \varphi_\gamma$

    Rule inversion over $\Gamma \vdash z : \{\gamma : b \mid \varphi_\gamma\}$ yields $\Gamma \vdash \gamma$ fresh.

    Lemma 5.3.4 for γ implies $\sigma[x \mapsto [\![y]\!]\sigma] : \Gamma[x \mapsto \gamma], \gamma \simeq [\![y]\!]\Gamma$.

    Choose $\Gamma_1 = \Gamma_2$ which implies the goal

  - Case Intro-Sub gives $\Gamma \vdash e' : \tau'$ and $\Gamma \vdash \tau' : \tau$ as well as the induction hypothesis for the inner expression e'.

    The argument for why this should hold should be similar to the Sub-Context a few cases further down.

- Case SS-Seq-Inner : Given: $\langle s \mid \sigma\rangle \rightsquigarrow \langle s_1 \mid \sigma_1\rangle, \Gamma \vdash s; r \Rightarrow \Gamma_2$,
  $\forall \Gamma_2, \Gamma \vdash s \Rightarrow \Gamma_2 \land \sigma : \Gamma \longrightarrow \exists \Gamma_1, \sigma_1 : \Gamma_1 \land \Gamma_1 \vdash s_1 \Rightarrow \Gamma_2$
  show $\exists \Gamma_1, \sigma_1 : \Gamma_1 \land \Gamma_1 \vdash s_1; r \Rightarrow \Gamma_2$.

  Rule inversion over $\Gamma \vdash s; r \Rightarrow \Gamma_2$ yields $\Gamma \vdash s \Rightarrow \Gamma_1, \Gamma_1 \vdash r \Rightarrow \Gamma_2$

  The preconditions for the third induction hypothesis are satisfied for Γ_1 and yields $\exists \Gamma_1', \sigma_1 : \Gamma_1' \land \Gamma_1' \vdash s_1 \Rightarrow \Gamma_1$

  State conformance $\sigma_1 : \Gamma_1'$ follows directly from this.

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

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

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

- Case SS-IF-T : Given $[\![x]\!]\sigma \doteq \texttt{true}$, $\Gamma \vdash \texttt{if}\ x\{s_t\}\texttt{else}\{s_e\} \Rightarrow \Gamma_2$, $\sigma : \Gamma$, show $\exists \Gamma_1, \sigma : \Gamma_1 \land \Gamma_1 \vdash s_t \Rightarrow \Gamma_2$

  Rule inversion over $\Gamma \vdash \texttt{..} \Rightarrow \Gamma_2$ yields $\Gamma \vdash s_t \Rightarrow \Gamma_2$. With $\Gamma_1 = \Gamma$ the goal is satisfied.

---

[5] The Lean Code can be found here `https://gitlab.com/csicar/liquidrust/-/tree/main/docs/lean-proof`

- Case SS-IF-F : analogous to Case SS-IF-T

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

- Case SS-Assign-Ref Given $\sigma(x) = \&y$, $\Gamma \vdash *x = z \Rightarrow \Gamma_2$, $\sigma : \Gamma$, show $\exists \Gamma_1, \sigma[y \mapsto \llbracket \sigma(z) \rrbracket] : \Gamma_1 \wedge \Gamma_1 \vdash \texttt{unit} \Rightarrow \Gamma_2$

  Cases of $\Gamma \vdash *x = z \Rightarrow \Gamma_2$:

  - Case Assign-Strong gives: $\Gamma(z) = \beta$, $\Gamma \vdash *x \in \{y\}$, $\Gamma \vdash \gamma$ fresh.

    Choose $\Gamma_2 = \Gamma_1 = \Gamma[y \mapsto \gamma], \gamma \doteq \beta$. With $\Gamma \vdash \alpha$ fresh using lemma 5.3.5 yields $\llbracket \sigma(x) \rrbracket = \&y$. lemma 5.3.4 yields $\sigma[y \mapsto \llbracket \sigma(z) \rrbracket] : \Gamma[y \mapsto \gamma], \gamma \doteq \beta$

  - Case Assign-Weak gives: $\Gamma \vdash z : \tau$, $\Gamma \vdash *x \in \{\bar{y_i}\}$ $\Gamma \vdash y_i : \{\beta_i : b_i \mid \varphi_i\}$ $\Gamma \vdash \tau \preceq \left\{\beta_j : b_j \mid \varphi_j\right\}$ for $j \in [1, n]$.

    Lemma 5.3.5 gives $\llbracket \sigma(x) \rrbracket = \&y_i$ for some $i \in [1, n]$.

    At this point it remains to be shown that the assigned type does not change conformance for the post state. We will show this in two steps: First we show that the assigned value preserves conformance, but for a different context. Then we will show that this context is a subcontext of Γ.

    Using lemma 5.3.4 we get $\sigma[y_j \mapsto \llbracket z \rrbracket \sigma] : \Gamma[y_j \mapsto \beta_j], (\alpha \simeq \llbracket z \rrbracket \Gamma)$ for all $j \in [1, n]$.

    By assumption $\Gamma \vdash \tau \preceq \left\{\beta_j : b_j \mid \varphi_j\right\}$ for every $j \in [1, n]$. Using lemma 5.3.6 we get $\sigma[y_j \mapsto \llbracket \sigma(z) \rrbracket] : \Gamma$. For $\Gamma = \Gamma_1 = \Gamma_2$ the proof obligation is satisfied.

- Case SS-While Given $\Gamma \vdash \texttt{while}(x)\{s\} \longrightarrow \Gamma_2$, $\sigma : \Gamma$ show $\exists \Gamma_1, \sigma : \Gamma_1 \wedge \Gamma_1 \vdash \texttt{if}(x)\{\texttt{while}(x)\{s\}\}\texttt{else}\{\texttt{unit}\} \Rightarrow \Gamma_2$

  Rule inversion over $\Gamma \vdash \texttt{while}(x)\{s\} \longrightarrow \Gamma_2$ yields $\Gamma \preceq \Gamma'$, $\Gamma(x) = \alpha$, $\Gamma \vdash c \longrightarrow \Gamma'$

  Thus $\Gamma \vdash \texttt{while}(x)\{s\} \Rightarrow \Gamma$. Which completes the precondition for $\Gamma \vdash \texttt{if}(x)\{\texttt{while}(x)\{s\}\}\texttt{else}\{\texttt{unit}\} \Rightarrow \Gamma_2$ and together with the second hypothesis implies the goal.

- Case SS-Relax Given $\Gamma \vdash \texttt{relax\_ctx!}\{\Gamma'\} \Rightarrow \Gamma_2$, $\sigma : \Gamma$ show $\exists \Gamma_1, \sigma : \Gamma_1 \wedge \Gamma_1 \vdash \texttt{unit} \Rightarrow \Gamma_2$.

  Rule inversion over $\Gamma \vdash \texttt{relax\_ctx!}\{\Gamma'\} \Rightarrow \Gamma_2$ yields $\Gamma_2 \preceq \Gamma$ For $\Gamma_1 = \Gamma_2$ and lemma 5.3.6 implies the goal.

- Case SS-Call function calls are executed by renaming and inlining and adding a context relaxation at the start and end of the function pertaining to the entry and exit context specifications. Definition 5.2.8 further justifies this.

□

5.4. Extensions

These are some ways to further improve the usability and expressiveness of the type system. Even though these extensions are not part of the implementation, they 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 could to be added to the type system.

5.4.1. 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 should seamlessly extend to `structs`: The variable mapping in Γ is generalized: $\mu : \text{Path} \to \text{LVar}$, which tracks the logic variable assignments for each field of an owned `struct`. The relevant typing rules should work without major changes.

```
struct Point { x: i32, y: i32 }
fn incr(i : &mut ty!{ i1 =>  i2 | i2 == i1 + 1 }) { .. }
fn move_up(p : &mut ty!{ p1 : Point => p2 | p2.x > p1.x && p1.y == p2.y }) {
  // $\Gamma_{init} = (\{p.x \mapsto v_1, p.y \mapsto v_2\}, true)$
  ...
  // $\Gamma_{final} \preceq (\{p.x \mapsto v_3, p.y \mapsto v_4\}, v_3 > v_1 \land v_4 \doteq v_2)$
}

fn client() {
    let p = Point {x:1, y:2}
    // $\Gamma_1 = (\{p.x \mapsto \alpha, p.y \mapsto \beta\}, \alpha \doteq 1 \land \beta \doteq 2)$
    incr(&mut p.x);
    // $\Gamma_2 = (\{p.x \mapsto i_2, p.y \mapsto \beta\}, \alpha \doteq 1 \land \beta \doteq 2 \land i_2 \doteq \alpha + 1)$
    move_up(&mut p);
    // $\Gamma_3 = (\{p.x \mapsto v_3, p.y \mapsto v_4, \ldots \land v_3 > i_2 \land v_4 \doteq \beta\})$
}
```

Listing 12: Example for how `structs` could be handled in Corten

Listing 12 shows how this could work in practice: When passing records as argument, we employ the same technique that we used for references: For any reference target without a direct name, we generate one: Referencing $p.x$ in `move_up` will reference a newly generated name associated with $p.x$. The predicates in the signature are immediately desugared to their corresponding logic variable.

5.4.2. 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 in Rust 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(403)` if the provided password was incorrect. This requires the type language to assert that a specific variant is expected as well as granting access to the variant's fields.

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

5.4.3. 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. [25] 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.[6]

5.4.4. Predicate Generics

Vazou et al. [32] 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.

5.4.5. Convenience Improvements

For practical use, it would make sense to relax the distinction between logic variables and program variables for immutable variables. Because these variables can not change, there is no need to introduce or replace their types. This also means that relaxing the predicate for immutable variables is never necessary.

In some cases, the programmer might want to choose an invariant for a variable that any assigned value has to satisfy. In terms of the type system, this would not be challenging: A set of variable's invariants are added the context and after each update, a subcontext relation between it and newly updated context are checked. If the subtyping relation can not be shown, then the program does not type check.

---

[6] It turns out that exactly is what Lehmann et al. [18] did. See section 8.1 for details

6. Implementation: CortenC

The type system described in the previous chapter was also implemented for Rust to test the practical feasibility of our approach[1]. There are a few differences and details between the type system described above and the implementation CortenC (short for Corten Type 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 amenable in practice.

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 as its base language. 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 meaning the types of expressions 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, which would have reduced the implementation effort. 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. MIR construction only occurs after HIR construction and rust type checking is completed. This means that just a single (unrelated) type error may keep the refinement system from starting to produce any errors, which can be quite annoying in practise, as can be seen by Rust borrow checking system, which already displays this kind of behaviour. 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 5.4)

---

[1] The implementation can be found at `https://gitlab.com/csicar/liquidrust`

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 13 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.

The second macro `relax_ctx!` takes the place of a statement and expands into a function call that contains the specified types as arguments. CortenC uses the arguments to reconstruct the described context for the type system.

When CortenC reads the type aliases, the predicates are simple strings that need to be parsed before CortenC can use them. Serendipitously, Rust’s tooling for procedural macros can also help here: The Rust crates `syn` and `quote` allow the programmer to easily parse and generate Rust source code. We use `syn` to parse the predicates, which not only lets the user express the predicates in a familiar syntax, but also makes the implementation of CortenC cleaner. Using Rustc’s parser for this purpose would probably be too involved.

Architecture CortenC follows the structure implied by the type system: All concepts and type checking rules have a correspondence in CortenC: For instance `RContext` is the representation of Γ, which only differs by the fact that predicates stay associated with a logic variable (and potentially a program variable). The purpose is 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.

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 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 14 shows the core functions signatures involved in type checking[2].

---

[2] Lifetimes were removed for clarity

```
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! { $i2 }}, {stringify! { $pred2 }}>
  };
  // -- snip --
}
```

Listing 13: Definition of CortenC's macros

`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` (corresponding to Sub-Ctx) and `require_is_subtype_of` (corresponding to Sub-Ty) 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 To better Consider the example program 15, 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 $\Gamma = (a : \{n \mid n > 0\})$ 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

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

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

// Γ ⊢ e ⇒ Γ′
fn type_expr(
  expr: &Expr,
  tcx: &TyCtxt,
  ctx: &RContext,
  local_ctx: &TypeckResults,
  solver: &mut SmtSolver,
  fresh: &mut Fresh,
) -> Result<(RefinementType, RContext)> { .. }
```

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

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

Listing 14: Overview of the Central Functions CortenC is Built from

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

Listing 15: Simple Example Program used for demonstrating the operation of CortenC

to $\Gamma_1 = (a : \{n \mid n > 0\}, b \mapsto \{v_1 \mid v_1 \doteq 1\})$ (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 $\{v_2 : i32 \mid v_2 \doteq n + v_1\}$. 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 16. Because the SMT call returned `unsat`, the subtype relation is valid and the function type checking was successful.

```
(declare-datatypes () ((Unit unit)))
; <Context>
  ; decl for <fud.rs>:79:73: 79:74 (#0) local b
  (declare-const v_0 Int)

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

  ; predicate for v_0: local b     ty!{ v_0 : i32 | v_0 == 1 }
  (assert (= |v_0| 1))

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

(declare-const v_1 Int)
(assert (= (+ |n| |v_0|) |v_1|))

(assert (not (> |v_1| 1)))

; checking: ty!{ v_1 : i32 | n + v_0 == _1 } <=  ty!{ v : i32 | v > 1 }
(check-sat)
; done checking is_subtype_of! is sat: false
```

Listing 16: SMT Requests dispatched by CortenC for checking that the returned type matches the specified type

7. Evaluation

In this chapter will conduct an evaluation on CortenC to test its practicality. Most of the example were defined at the start of writing the thesis and are accompanied by some examples from other papers in the field as well as some benchmarks to show the extend of what is possible in CortenC.

7.1. Maximum using Path Conditions

The first example is based by Rondon et al. [25]. It demonstrates that the `max` function can be implemented and fully verified using path sensitivity. The mathematical maximum is defined as $r = \max(a, b)$ iff $r \geq a \land r \geq b \land (r = a \lor r = b)$. Listing 17 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 17: Example demonstrating a fully specified `max` function using Corten's path sensitivity

CortenC will automatically check the type specification matches the implementation, which succeeds in this case. 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 18. The exact program line where the error occurred is also by CortenC. To help the user, CortenC also extract the counter example generated by the SMT solver and displays it to the user understand what when wrong. It is a fortunate coincidence, that when generating counter example Z3 will often leave out variable, that are irrelevant to contradiction, which automatically creates a kind of minimal working counter example.

Path sensitivity is not limited to the return value: Effects on the type context can also be path sensitive as demonstrated by listing 19, which implements a clamping function. The function `clamp` ensures that the reference passed to it will be at most `max` or stay the same. The `client` function uses `clamp` and can use the facts from `clamp`'s mutation specification when proving its own return type specification.

Note that when typing `client`, CortenC matches `&mut x` with `a` and thusly `x` with `s` to correctly handle the effects of calling `clamp`. Because of this reason, function calls

```
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 }
}
For example this assignment, satisfies the sub type, but not the super type:
  { _1 = 0, b = 1, a = 0 }
```

Listing 18: Example of an error message created by CortenC

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 19: Example demonstrating optional mutation of an external location

7.2. Recursion: Fibonacci-Numbers

Thanks to Corten's modular approach, handling recursive functions is straightforward in CortenC. Listing 20 demonstrates this by implementing a function that returns the nth fibonacci number F_n and asserting that $10 \cdot F_n \geq n^2$. Because n is arbitrary, this proves[1] that the Fibonacci sequence grows faster than $\frac{n^2}{10}$.

---

[1] The specification is not technically in linear arithmetic, but CortenC can allow some leeway, as long as SMT-Solver are still able to solve it.

```
fn fib(n: ty!{ nv: i32 | nv >= 0}) -> ty!{ v: i32 | 10 * 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 | 10 * r >= nv * nv }
  } else { 1 }
}
```

Listing 20: 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 21 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 returns the same result as 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's solution of treating predicates as immutable while retaining unassociated predicates shows its practicability here.

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 is established atomically.

For this purpose, CortenC's `relax_ctx!` macro is used, which the programmer (or an inference system) can use to change the type of multiple variables at a time.

7.4. Complex Mutable References

Listing 22 shows that CortenC can correctly track mutation patterns across function boundaries in a modular way. The `swap` function replaces the value of `x` with `y` and vice versa. The specification states that for any values `x1` and `y1` at the reference target of `x` and `y`, swap will update their values to satisfy the predicates `x2 == y1` and `y2 == x1`. Notice that CortenC can use these facts in the `client` function: For `*a` to resolve to the required type, we have to infer that `b` is a reference to `j` with is associated with logic variable `j1` and `swap` ensures that the value referenced by `a` now references the value of `j1`. Therefore `a` must now refer to `j`, which is associated with logic variable `j1`.

```
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 21: Example loops with complex loop invariants and value updates affecting the invariant

```
fn swap(
  x: &mut ty!{ x1: i32 => x2 | x2 == y1 },
  y: &mut ty!{ y1: i32 => y2 | y2 == x1 }
) -> ty!{ v: () }  {
  let tmp = *x;
  *x = *y;
  *y = tmp;
  ()
}
```

```
fn client(
  mut i: ty!{ i1 : i32 },
  mut j: ty!{ j1 : i32 }
) -> ty!{ v: i32 | v == j1-i1 } {
  let mut a = &mut i;
  let mut b = &mut j;
  swap(a, b);
  *a - *b
}
```

Listing 22: Example demonstrating modularity and ease of specification for complex mutation patterns

7.5. Pseudo Vectors

The next example will demonstrate that CortenC would be able to specify and verify the absence of index-out-of-bounds errors on vectors. To this end, listing 23 the type alias `IntVec` is introduced, where its value represents the vector length. This example is inspired by a example for the Flux paper by Lehmann et al. [18], where the authors specify a `Vec<T>` with elements of type `T` and show, that the index is kept in bounds. Similarly, the functions in listing 23 are part of the interface of vectors and ensure no index outside the bounds is read or written to. The example also demonstrates that CortenC can also check that mutable reference are in fact unchanged. For instance the function `len` and `get` take the vector `v` as a mutable reference, but CortenC can still show that these functions do not change the value and therefore all properties from before the function call carry over to after the function call.

CortenC will automatically typecheck the program. If `pop` was called on a vector that could not be shown to have sufficient length, CortenC would reject the program.

7.6. 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 24 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.

7.7. Interoperability with Other Tools

Corten can only verify safe Rust, but as we discovered in chapter 3, unfortunately `unsafe` Rust is used at times. For use-cases where the unsafe code exposes a safe interface, CortenC could be used to verify the safe part and delegating the verification of the rest to a more complex verification systems.

To facilitate this usage pattern, CortenC provides a builtin macro (shown in listing 25), that will instruct CortenC to assume a predicate is true without checking it, named `assume_corten`. CortenC will extract the path condition from the call and leave it up to an external tool to show, that the `unreachable!` macro is in fact unreachable. For instance, Prusti[3] is able to use such conditions to generate appropriate proof obligations to ensure the statement is unreachable.

```
type IntVec = i32;

fn new() -> ty!{v: IntVec | v == 0 } {
    0
}

fn len(v : &mut ty!{ b: IntVec => a | a == b }) -> ty!{ r: i32 | r == b } {
    *v
}

fn get(
    v : &mut ty!{ b: IntVec | true => a | a == b },
    index: ty!{i : i32 | 0 <= i && i < b}) -> ty!{ r: () } { ()
}

fn is_empty(v : &mut ty!{ v1 : i32 => v2 | v2 == v1 })
  -> ty!{ e : bool | e == (v1 == 0) } {
    let l = len(v); l == 0
}

fn push(v : &mut ty!{ v1 : i32 => v2 | v2 == v1 + 1 }) -> ty!{ r : () } {
    *v = *v + 1; ()
}

fn pop(v : &mut ty!{ v1 : i32 | v1 > 0 => v2 | v2 == v1 - 1 }) -> ty!{ r : () } {
    *v = *v - 1; ()
}

fn client() -> ty!{ r : i32 | r == 1 } {
  let mut v = new(); let mut i = 1;
  push(&mut v); push(&mut v); get(&mut v, i);
  pop(&mut v); push(&mut v); pop(&mut v);
  i = 0; get(&mut v, i);
  len(&mut v)
}
```

Listing 23: Example demonstrating modularity and ease of specification for complex mutation patterns

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

```
macro_rules! corten_assume {
    ( $pred:expr ) => {
        builtin_assume($pred, if(!$pred) { unreachable!(); } else {()});
    };
}
pub fn builtin_assume(formula: bool, proof: ()) {}

fn client() { ... corten_assume!(a > 0); ... }
```

Listing 25: Macro provided by CortenC to offload verification to other tools

8. Related Work

Relevant papers originate from two lines of work: On one hand additions to refinement types for mutability, asynchronous execution etc. and on the other verification frameworks for Rust.

Refinement Types Refinement Types were originally developed for verification in functional languages by Freeman and Pfenning [10], who define a subtyping relation for ML types on a lattice of possible union and intersection of types and check them using a system called tree automata. Xi and Pfenning [35] rephrase this notion subtype refinement as a restricted dependent type, where the constraints less expressive than the base language, making automatic verification attainable. They show that type checking is decidable (relative to a decidable domain), but inference is not.

Taking advantage of the advances in SMT solvers, Rondon et al. [25] devised a type system that combines the ideas of refinement subtyping and restricted dependent types with the novel inference system to create a path sensitive and decidable type inference system.

Vazou et al. [32] extend the refinement language to allow for uninterpreted predicates to be placed in the refinement predicate, which retain the decidability of type checking, while at the same time offering significant benefits to expressiveness.

Refinement types are also used in other applications. For example Graf et al. [11] 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.

Refinement Types and Mutability Rondon et al. [26] extend Liquid Types to a reduced subset of C featuring mutable aliased pointers. Their type system CSolve extends Refinement Types to enable verification of low-level program with pointer arithmetic. Bakst and Jhala [5] present a type system based on Refinement Types named Alias Refinement Types, which combines alias types with Refinement Types. Both approaches focus on a C-like target language that provides little guarantees, which is not conducive to reasoning about aliasing, necessitating ad-hoc mechanisms to control aliasing.

Recognizing the fact, Sammler et al. [28] devise an ownership type system for C used that is combined with a Refinement Types. Besides automatic verification, their type system can also translate their proofs to Coq, following the approach of RustBelt. RefinedC ownership model differs from Rust.

Kloos et al. [15] 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 offers extensive guarantees, which offers substantial advantages in terms of simplicity to specification and verification of Rust programs.

Also Lanzinger [16] successfully adapted refinement types to Java, which allows the user to check that property types described by java annotations hold true throughout the program, although only for immutable data. Bachmeier [4] later extended the approach to handle mutable data as well.

Rust verification Given Rust's ownership system and predestined position as language for writing future safety critical software, Rust has sparked interest in the software verification field. There are several papers with a variety of approaches.

Ullrich [31] translates Rust's MIR to the theorem prover / dependently typed language Lean ([22]), forgoing automation, but offering a clean modelling and extensive coverage of the Rust language.

A key observation made by Ullrich is that with Rust's ownership system, mutable references can be encoded as so called lenses[8]. Basically, mutable reference parameter are translated in to a immutable initial value and a returned value representing the mutated type. Lenses extend this encoding by allowing a caller to give a callee access to part of a structure letting the Lens mechanism handle the reassembly of the structure from the changed value. The insight of Ullrich is that Rust's mutable references can naturally be translated into a sequence of sequential, linear updates to a data structure. Corten's syntax for mutable refinement parameters is inspired by Ullrich's encoding. Denis et al. [7] also use a translation technique, but use Why3 as the target.

A different verification approach is taken by Astrauskas et al. [3] with Prusti, which is a heavy-weight functional verification framework for Rust, that translates its proof obligations into a separation logic based verification infrastructure named Viper.

There also exists a line of work that focuses on the generation of constrained Horn. For example Matsushita et al. [20] employ this technique for verification of Rust. Particularity relevant for this thesis is the novel formalization for mutable references used in the paper. The authors stipulate that lending of a variable should have a contract between the lender and the borrower: The borrower requires some precondition to be satisfied about the mutable reference and upon returning the borrow, the borrower ensures that a postcondition is met. Matsushita et al. [21] further extend this like of work to cover unsafe code.

Other tools also use (bounded) model checking techniques for verification, like Kani [1] or Crust, described by Toman et al. [30].

8.1. Comparison to Flux

During the writing of the thesis, Lehmann et al. [18] published a preprint for a paper that is especially relevant. Lehmann et al. describe a implementation of Liquid Types for Rust called Flux, covering the same solution and problem space to Corten. For the purpose of plurality in design and implementation of such a system, is was decided to only read the paper after finishing the Corten type system.

Flux distinguished between three different kinds of references: Shared (immutable) references, mutable weak (unique) references and strong references.

```
#[lr::sig(fn(i32<@n>)
  -> bool<0 < n>)]
fn is_pos(n: i32) -> bool {
  if 0 < n { true } else { false }
}

#[lr::sig(fn(i32<@x>)
  -> bool{v: x <= x && 0 <= v>)]
fn add(x: i32) -> i32 {
  if x < 0 { -x } else { x }
}
```

```
fn is_pos(n: ty!{ n : i32})
  -> ty!{ v: bool | v == 0 < n } {
  if 0 < n { true as ... } else { false }
}

fn add(x: ty!{ x: i32})
  -> ty!{ v: i32 | x <= x && 0 <= v } {
  if x < 0 { -x as ... } else { x }
}
```

Listing 26: Example demonstrating the Ownership System: `greet(a)` transfers ownership of `a` to `greet`

To demonstrate the difference in the handling of strong updates, consider the signatures of incrementing and decrementing functions of a datatype `Nat`. The functions are fully specifications and ensure the invariant that a natural number is a positive.

```
fn increment(&strg v : Nat<n>) -> ()
  ensures *self: Nat<n+1>;
fn decrement(&strg v : Nat<n>) -> ()
  requires n > 0
  ensures *self: Nat<n-1>;
```

```
fn increment(n: &mut ty!{
 n1: Nat => n1 | n1 == n1+1 }
) -> ();

fn decrement(n: &mut ty!{
 v1: Nat | v1 > 0 => v2 | v2 == v1-1 }
) -> ();
```

Listing 27: Comparison of specifying Type Changes Caused by a strong mutation. Flux on the left; Corten on the right

Flux type system has the function body judgement $\Sigma, \Delta \mid K; T \vdash F$, which ”checks if a function body is well typed under a global environment Σ” [18, p. 11]

Δ in Flux is analogous to to Corten's Φ. Both contain the constraints on refinement variables and are initialized with the function preconditions. Flux's T maps locations to types, and serves a similar purpose to μ in Corten: Keeping track of the current properties that are known for a location. Corten's type system does not associate constraints with locations with specific predicate, like Flux does, because in the typing rules of Corten, the origin of the constraint is irrelevant[1].

Flux has a separate type syntax for references: $\&_{\text{mut}}\tau$ that denotes that the referenced location has type τ. Corten uses a similar syntax for specifying mutable references, but in terms of the semantics delegates the type constraint to the referenced value and even disallows reference types to constrain the referenced type. Because the reference location can change during the program execution, constraining a reference can have many possible meanings and unintuitive consequences. Flux avoids this problem by restricting to weak references, which can not change their type. Presumably for the same reasons, Flux like Corten chose to forbid type specification on strong references, but their handling differs a bit: Even though strong reference destinations are specified with a similar syntax ptr (l) (equivalent to $\&l$ in Corten), Corten allows multiple possible destinations to be specified. In that case, the reference may only be update weakly. Importantly, Corten allows the type to change, meaning a reference can be unambiguous and therefore strongly updatable and later ambiguous in its target meaning only weakly updatable.

---

[1] interestingly our implementation, like Flux, does have that distinction to improve error messages

Function signatures are expressed quite similarly. Flux function signatures have the form $\forall v \mathrel{\bar{:}} \sigma.\mathrm{fn}\,(r; \bar{x}.T_i) \longrightarrow \rho_0.T_0$. Consider the function decrease from listing 27. In Flux the function signature would be encoded as:

$$\forall n : \mathrm{int}, \rho : \mathrm{loc}.\ \mathrm{fn}\,(n > 0, x.\{\rho \mapsto \mathrm{nat}_\tau\langle n\rangle, x \mapsto \mathrm{prt}(\rho)\} \longrightarrow \rho_0.\{\rho_0 \mapsto (), \rho \mapsto \mathrm{nat}_\tau\langle n-1\rangle\})$$

The rule T-Def initializes the context to $\Delta = (n > 0, v : \mathrm{loc}, n : \mathrm{int})$, $K = (x \mapsto \mathrm{prt}(\rho), \rho \mapsto \mathrm{nat}_\tau\langle n\rangle)$ and the expected return value $T_0 = \{\rho_0 \mapsto (), \rho \mapsto \mathrm{nat}_\tau\langle n-1\rangle\}$

Corten would generate an initial context $\Gamma = (\mu, \Phi)$ with $\mu = \{n \mapsto \&arg_0, arg_0 \mapsto v_1\}$, $\Phi = \{v_1 > 0\}$ and expected return type $\{v_r : () \mid \mathrm{true}\}$ and end context $\mu' = \{arg_0 \mapsto v_2\}$, $\Phi' = \{v_2 \doteq v_1 - 1, v_1 > 0\}$.

What Corten calls a sub-context relation, is referred to as context inclusion in Flux. Though Flux is significantly more conservative with its rules; only allowing permutations (C-Perm), removal (C-Weaken) of predicates and subtyping of just individual types (C-Sub) and only if they can be separated (C-Frame), meaning they do not depend on other predicates. Flux should thusly only be able to show inclusion for a subset of what the logic and theories allow. As an example consider the sets displayed in fig. 8.1, which are described by the constraints super-context $x > y - 1 \land x > y + 1$, which allows x to vary from y by at most 1. A sensible sub-context would be $x = y$, which satisfies the constraints, but there is no possible proof for this fact in Flux. Because Corten delegates sub-context checking to SMT, handling these dependencies are no problem.

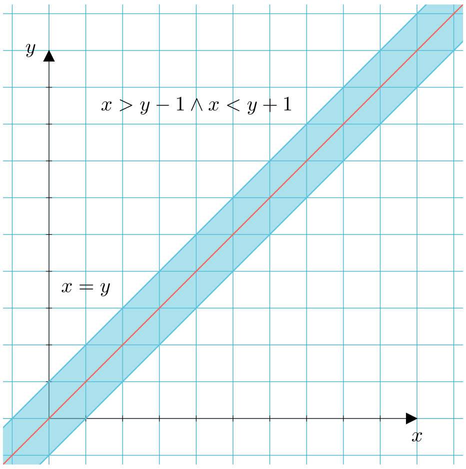

Figure 8.1.: Figure showing a value space, where inclusion can not be shown

The subtyping rules are quite similar. To the degree applicable, S-Exists is analogous to $\preceq$-Ty.

Looking at the choice of target language, we can see another difference: Flux chose λ_{Rust} by Jung et al. [14] as a basis for the formalization, which expresses the semantics in a continuation passing style over a language based on Rustc's MIR, which provides a more direct correspondence to the actual Rust. On the other hand Corten based the formalization on a language based on Rustc's HIR and semantics expressed in a small-spec semantic.

The corresponding implementations also use the MIR and HIR respectively. The advantages and disadvantages are discussed in chapter 6. Flux takes advantage of the MIR to cover a greater variety of control-flow scenarios. Lehmann et al. also found that "the refinement annotations [..] do not appear in Rust's MIR" [18, p. 17], which poses a challenge for the implementation.

Lastly Lehmann et al. evaluate Flux against Prusti proving properties like "the absence of index-overflow error in a suite of vector-manipulating programs." [18, p. 2] Lehmann et al. found that the liquid type inference system is powerful enough to infer necessary invariants for this use case, partly because "loop invariants express either simple inequalities or tedious bookkeeping." [18, p. 21] In these cases Prusti still required invariants to be specified, which Lehmann et al. party attribute to the fact, for every unchanged mutable reference, the invariant needs to state that the reference was not modified. By using `mut`, but not `strg` references, Flux could avoid that problem. Even though Corten does not share the the strong references mechanism with Flux, the argument still applies: Corten would also not require additional invariants for the unchanging mutable references. Because Corten does not support vectors (yet), a direct comparison is not possible, but listing 23 approximates it.

9. Conclusion & Future Work

The goal of this thesis is to show that Refinement Types can be idiomatically adapted to languages with unique mutable references. The thesis has show that CortenC can type-check Refinement Types in Rust, enforcing properties over mutable data and references and that these types fit idiomatically into Rust, if an inference system was added to CortenC. The type system was sufficiently expressive for verifying complex properties about mutable data, recursive functions and loop invariants. In addition, the thesis analyzed the use of mutability in Rust and reaffirmed the limited prevalence of `unsafe` in Rust code. For a subset of Rust (named MiniCorten), a formal description of the syntax and small-step semantics was described. A syntax for Refinement Types for Rust was devised and an accompanying type system was devised. The thesis also provides justification for its soundness in the form of preservation (and progress) properties. Based in the formal description, a proof-of-concept implementation shows that automatic type checking of this type system is feasible and practical. The implementation also shows that using the interface provided by Rustc is sensible and beneficial for user-friendliness.

The evaluations has shown that meaningful specifications can be automatically type checked. As planned and suspected, the lack of an inference system, calls for some avoidable specification, otherwise the evaluation has demonstrated that the required specification effort is limited. The breath of features implemented in the thesis is limited, mainly because getting the foundations right took some experimenting, though the foundations layed in the thesis are purposefully designed with these extensions in mind. Therefore covering more language features should not be hard to achieve.

In terms of future work, there are two axis to work on: On one hand covering more language features, like records or algebraic data types, would greatly improve the practicality of CortenC. Of course the extensions outlined in chapter 4 are also prime candidates for future improvements.

On the other hand expanding the expressiveness of the type system would also be interesting: Introducing abstract predicates would allow a whole new set of properties to be specified. In particular developing a model of concurrency patterns could be a great application for them in Rust.

In terms of the analysis, it would also be interesting to correlate unsafe code blocks with the frequency of changes using, for example the git history. If unsafe code changes often, that probably means that this `unsafe` use is more problematic as it clearly needs to change often.

Bibliography

[1] *Announcing the Kani Rust Verifier Project.* Kani Rust Verifier Blog. May 4, 2022. URL: `https://model-checking.github.io//kani-verifier-blog/2022/05/04/announcing-the-kani-rust-verifier-project.html` (visited on 10/03/2022).

[2] Vytautas Astrauskas et al. "How do programmers use unsafe rust?" In: *Proceedings of the ACM on Programming Languages* 4 (OOPSLA Nov. 13, 2020), pp. 1–27. ISSN: 2475-1421. DOI: `10.1145/3428204`. URL: `https://dl.acm.org/doi/10.1145/3428204` (visited on 02/23/2022).

[3] Vytautas Astrauskas et al. "Leveraging rust types for modular specification and verification". In: *Proceedings of the ACM on Programming Languages* 3 (OOPSLA Oct. 10, 2019), pp. 1–30. ISSN: 2475-1421. DOI: `10.1145/3360573`. URL: `https://dl.acm.org/doi/10.1145/3360573` (visited on 02/23/2022).

[4] Joshua Bachmeier. *Property Types for Mutable Data Structures in Java.* 2022. DOI: `10.5445/IR/1000150318`. URL: `https://publikationen.bibliothek.kit.edu/1000150318` (visited on 10/03/2022).

[5] Alexander Bakst and Ranjit Jhala. "Predicate Abstraction for Linked Data Structures". In: *Verification, Model Checking, and Abstract Interpretation.* Ed. by Barbara Jobstmann and K. Rustan M. Leino. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2016, pp. 65–84. ISBN: 978-3-662-49122-5. DOI: `10.1007/978-3-662-49122-5_3`.

[6] Travis Breaux and Jennifer Moritz. "The 2021 Software Developer Shortage is Coming". In: *Commun. ACM* 64.7 (June 2021). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 39–41. ISSN: 0001-0782. DOI: `10.1145/3440753`. URL: `https://doi.org/10.1145/3440753`.

[7] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. "The Creusot Environment for the Deductive Verification of Rust Programs". PhD thesis. Inria Saclay-Île de France, 2021.

[8] J. Nathan Foster et al. "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem". In: *ACM Transactions on Programming Languages and Systems* 29.3 (May 1, 2007), 17–es. ISSN: 0164-0925. DOI: `10.1145/1232420.1232424`. URL: `https://doi.org/10.1145/1232420.1232424` (visited on 10/03/2022).

[9] Johannes Foufas. *Why Volvo thinks you should have Rust in your car*. Volvo Cars Engineering. Sept. 23, 2022. URL: `https://medium.com/volvo-cars-engineering/why-volvo-thinks-you-should-have-rust-in-your-car-4320bd639e09` (visited on 09/26/2022).

[10] Tim Freeman and Frank Pfenning. "Refinement types for ML". In: *ACM SIGPLAN Notices* 26.6 (May 1, 1991), pp. 268–277. ISSN: 0362-1340. DOI: `10.1145/113446.113468`. URL: `https://doi.org/10.1145/113446.113468` (visited on 09/29/2022).

[11] Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. "Lower your guards: a compositional pattern-match coverage checker". In: *Proceedings of the ACM on Programming Languages* 4 (ICFP Aug. 2, 2020), pp. 1–30. ISSN: 2475-1421. DOI: `10.1145/3408989`. URL: `https://dl.acm.org/doi/10.1145/3408989` (visited on 03/15/2022).

[12] Dan Grossman et al. "Region-Based Memory Management in Cyclone". In: (), p. 12.

[13] Ralf Jung. *MiniRust*. original-date: 2022-05-21T15:30:40Z. Sept. 28, 2022. URL: `https://github.com/RalfJung/minirust` (visited on 09/28/2022).

[14] Ralf Jung et al. "RustBelt: securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages* 2 (POPL Jan. 2018), pp. 1–34. ISSN: 2475-1421. DOI: `10.1145/3158154`. URL: `https://dl.acm.org/doi/10.1145/3158154` (visited on 02/17/2022).

[15] Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. "Asynchronous Liquid Separation Types". In: (2015). In collab. with Marc Herbstritt. Artwork Size: 25 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 25 pages. DOI: `10.4230/LIPICS.ECOOP.2015.396`. URL: `http://drops.dagstuhl.de/opus/volltexte/2015/5223/` (visited on 01/27/2022).

[16] Florian Lanzinger. "Property Types in Java: Combining Type Systems and Deductive Verification". Master Thesis. Karlsruher Institut für Technologie, Feb. 2021.

[17] Michael Larabel. *LPC 2022: Rust Linux Drivers Capable Of Achieving Performance Comparable To C Code*. phoronix. Sept. 12, 2022. URL: `https://www.phoronix.com/news/LPC-2022-Rust-Linux` (visited on 09/26/2022).

[18] Nico Lehmann et al. *Flux: Liquid Types for Rust*. Number: arXiv:2207.04034. July 8, 2022. arXiv: `2207.04034[cs]`. URL: `http://arxiv.org/abs/2207.04034` (visited on 07/14/2022).

[19] Nicholas D. Matsakis and Felix S. Klock. "The rust language". In: *ACM SIGAda Ada Letters* 34.3 (Oct. 18, 2014), pp. 103–104. ISSN: 1094-3641. DOI: `10.1145/2692956.2663188`. URL: `https://doi.org/10.1145/2692956.2663188` (visited on 09/26/2022).

[20] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. "RustHorn: CHC-based verification for Rust programs". In: *European Symposium on Programming*. Springer, Cham, 2020, pp. 484–514.

[21] Yusuke Matsushita et al. “RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code”. In: *San Diego* (2022), p. 16.

[22] Leonardo de Moura et al. “The Lean Theorem Prover (System Description)”. In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21401-6. DOI: `10.1007/978-3-319-21401-6_26`.

[23] *Overview of the Compiler - Guide to Rustc Development*. URL: `https://rustc-dev-guide.rust-lang.org/overview.html` (visited on 09/19/2022).

[24] Benjamin C. Pierce. *Types and Programming Languages*. Google-Books-ID: ti6zoAC9Ph8C. MIT Press, Jan. 4, 2002. 656 pp. ISBN: 978-0-262-16209-8.

[25] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. “Liquid types”. In: *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI ’08*. the 2008 ACM SIGPLAN conference. Tucson, AZ, USA: ACM Press, 2008, p. 159. ISBN: 978-1-59593-860-2. DOI: `10.1145/1375581.1375602`. URL: `http://portal.acm.org/citation.cfm?doid=1375581.1375602` (visited on 06/30/2022).

[26] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. “Low-level liquid types”. In: *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’10. New York, NY, USA: Association for Computing Machinery, Jan. 17, 2010, pp. 131–144. ISBN: 978-1-60558-479-9. DOI: `10.1145/1706299.1706316`. URL: `https://doi.org/10.1145/1706299.1706316` (visited on 09/16/2022).

[27] Amr Sabry and Matthias Felleisen. “Reasoning about Programs in Continuation-Passing Style”. In: *Lisp and Symbolic Computation*. 1993, pp. 288–298.

[28] Michael Sammler et al. “RefinedC: automating the foundational verification of C code with refined ownership types”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. New York, NY, USA: Association for Computing Machinery, June 19, 2021, pp. 158–174. ISBN: 978-1-4503-8391-2. DOI: `10.1145/3453483.3454036`. URL: `https://doi.org/10.1145/3453483.3454036` (visited on 09/29/2022).

[29] *The THIR (Typed High-level IR) - Guide to Rustc Development*. URL: `https://rustc-dev-guide.rust-lang.org/thir.html` (visited on 09/19/2022).

[30] John Toman, Stuart Pernsteiner, and Emina Torlak. “Crust: A Bounded Verifier for Rust (N)”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). Nov. 2015, pp. 75–80. DOI: `10.1109/ASE.2015.77`.

[31] Sebastian Ullrich. “Simple Verification of Rust Programs via Functional Purification”. In: *6.12.2016* (), p. 65.

[32] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. "Abstract Refinement Types". In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer, 2013, pp. 209–228. ISBN: 978-3-642-37036-6. DOI: `10.1007/978-3-642-37036-6_13`.

[33] Philip Wadler. "Theorems for free!" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. FPCA '89. New York, NY, USA: Association for Computing Machinery, Nov. 1, 1989, pp. 347–359. ISBN: 978-0-89791-328-7. DOI: `10.1145/99370.99404`. URL: `https://doi.org/10.1145/99370.99404` (visited on 08/29/2022).

[34] A. K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness". In: *Information and Computation* 115.1 (Nov. 15, 1994), pp. 38–94. ISSN: 0890-5401. DOI: `10.1006/inco.1994.1093`. URL: `https://www.sciencedirect.com/science/article/pii/S0890540184710935` (visited on 10/03/2022).

[35] Hongwei Xi and Frank Pfenning. "Dependent types in practical programming". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '99. New York, NY, USA: Association for Computing Machinery, Jan. 1, 1999, pp. 214–227. ISBN: 978-1-58113-095-9. DOI: `10.1145/292540.292560`. URL: `https://doi.org/10.1145/292540.292560` (visited on 09/29/2022).

A. Additional Graphs from the Use-Case Analysis

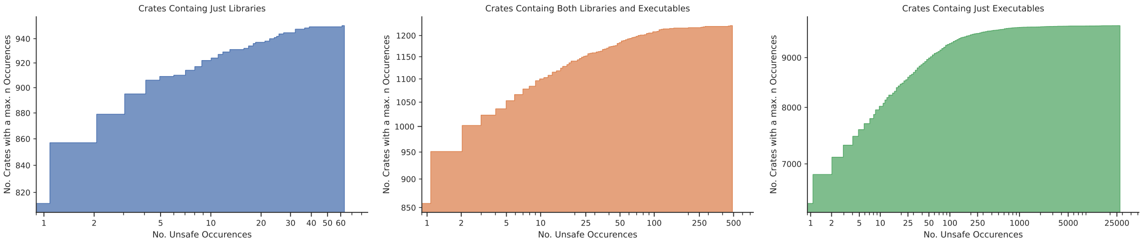

Figure A.1.: Cumulative, Logarithmic Histogram of the Amount of `unsafe` Uses in each Category

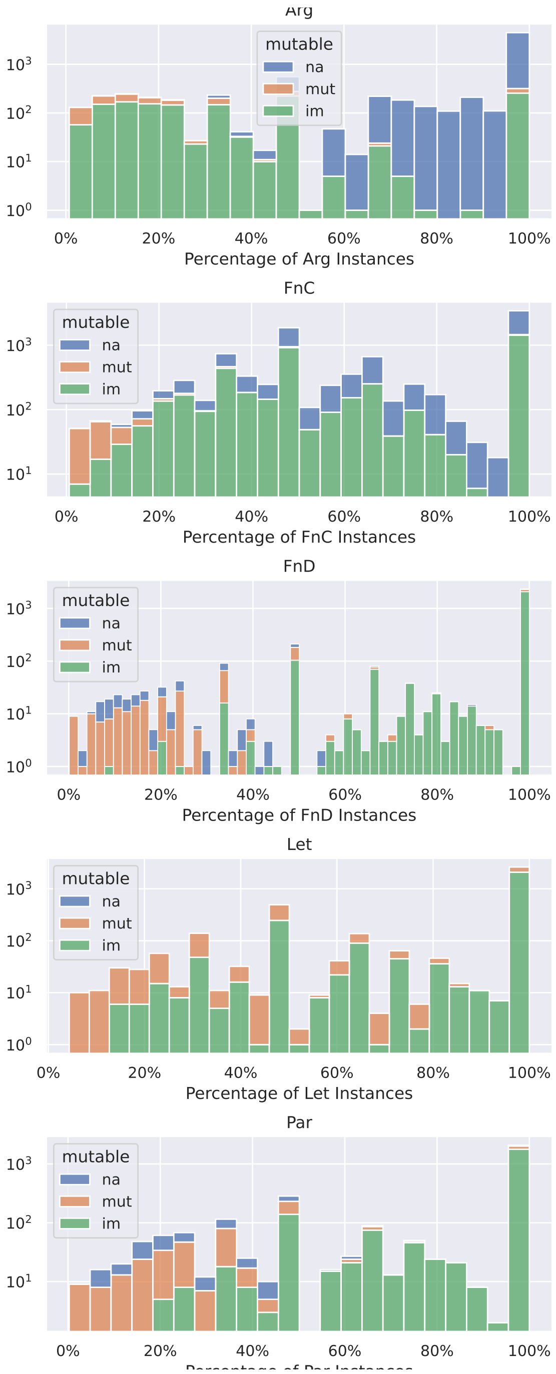

Figure A.2.: Logarithmic Histogram of the Number of Crates that contain n % immutable / mutable items