Exposé Masterarbeit

Carsten Csiky

1 Abstract

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

2 Defining the Goal

The goal of the thesis is to show that Refinement Types can be idiomatically adapted to languages with unique mutable references. It aims to enable gradual adoption of lightweight verification methods in mutable languages.

I aim for automated parsing of refinement types and verification. In case implementing these translations could not be completed, the use cases will be transformed manually. In addition, a description of the syntax and semantics of the constructs, that will extend Rust's type system, will be provided as well as justification for their soundness. A feasibility study on minimal examples of representative use cases will show how useful the proposed verification system is. For use cases that could not be covered by the proposed additions, the reason will be investigated.

Specifying or verifying complete Rust modules or the entire Rust language is not the goal of the thesis. In particular `unsafe` Rust will not be

taken into account in specification nor implementation. Implementing Liquid Type inference in also not a goal of the thesis.

3 Exposition

3.1 Why?

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

3.2 What?

The thesis will enable automated verification of programs with mutability against refinement type specifications by taking advantage of Rust's strong Ownership Model.

The verification should be sound, decidable and feasible in the identified use cases, which will be tested on minimal – yet representative – examples. The examples will include full refinement type annotations, consequently making type inference optional. Listing 1 shows how an example program might look like: The `ty!` macros specify Refinement Types, which are used in the type checking of the `client` function for detect an index out of bounds error.

```
fn push_all(
  a: &mut ty!({ o: Vec<i32> } ~~> { v : Vec<i32> | v.len() == o.len() + b.len()  }),
  b: &Vec<i32>) {
    ...
}
fn client() -> i32 {
    let mut a : ty!{ v: Vec<i32> | v.len() == 2 } = vec![1, 2];
    let     b : ty!{ w: Vec<i32> | w.len() == 2 } = vec![2, 3];
    push_all(a, b);
    // a : ty!{ u: Vec<i32> | u.len() == 4 }
    return a.get(5);
    //     ^^^^^^^^  type error!
    // Function `get` expected: `index` of type `{ v: usize | v <= 4 }`
    //                but got:   `{ v: usize | v == 5}`
}
```

Listing 1: Example Use-Case: `push_all` specifies how the `Vec` length changes by calling it; as a result `client` should not compile.

Additionally justification for why the refinement types system is sound will be provided. The specification will be written in an extension of the Rust type language, inspired by refinement types.

3.3 How?

This section starts with outlining the fundamental ideas (denoted "T1" to "T5") the thesis will be based on.

- T1 Identify the relevant concepts (like mutable, immutable and owned types) and their semantics. The deliverable is a set of minimal – yet representative – examples, with explanations for why they are representative.
- T2 Devise a Refinement Type system that covers the relevant concepts, in particular mutable but unique references. Refinement types are inherently limited in expressiveness, thusly in cases where use cases cannot be covered, the reason will be explained. The result is a concrete description of the syntax and type judgment semantics.
- T3 Justify why the type system is sound and investigate its limitations. If necessary for the use cases, a definition of suitable axioms for language build-in types will be added. The deliverables is a set of refinement type coercions.

T4 Develop a prototype for translating refinement type judgments to a verification backend. The deliverable is a parser for Refinement Types and a tool for transformation of to a verification backend or alternatively a description of the process.

T5 Conduct a feasibility study on different categories of common use cases. The result is a set of proven examples. If some examples could not be proven, the reason will be investigated and explained.

Instead of conducting these tasks once for all use cases the set of tasks will be performed incrementally on the identified use cases in order of increasing complexity. These steps are denoted "S1" to "S7". These steps are connected to specific time in Section 5.

S1 **Fundamentals**: Cover the mutable, immutable and owned variables, which are specified with (non-abstract) refinement types. In terms of the value-level language, this step will cover assignments, if statements and tuples or product types.

S2 **Modular Verification**: Extend the system to cover modular verification. This entails propagating type information between caller and callee. Consequently, the value-level language will be extended with function calls.

S3 **Loops**: This step adds loops to the value-level language necessitating the handling of loop invariants (or an alternative thereof). The invariants will not be automatically inferred.

S4 **Predicate Generics**: Improve the expressivity of the type system by adding abstract refinement

S5 **ADTs**: This step would consist of extending the type system with sum types and the language with pattern matching, as well as extending the specification with means of expressing restrictions on sum type variants.

S6 **Higher-Order Functions**: Extend the system with non-capturing lambdas and their refinement type specification. Completing this step is optional.

S7 **Concurrency**: Explore how this approach can be expanded to cover concurrent programs. Starting this step is optional.

4 State of the Art

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

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

For example, Lanzinger [2] successfully adapted refinement types to Java, which allows the user to check, that property types described by java annotations hold true throughout the program. At this point in time, specification and verification is limited to immutable (`final`) data.

Kloos et al. [3] extended refinement types to mutable and asynchronous programs. The paper explores how changes to possibly aliased memory cells can be tracked throughout a OCaml program. For that purpose the types are extended by a set of requirements on memory objects, which track distinctness and refined types of these memory cells. In contrast to OCaml, Rust already guarantees that mutable memory is not aliased and in particular *all mutable memory locations must be accessible by a variable name in the current context*, which offers substantial advantages in terms of simplicity to specification and verification of Rust programs.

Refinement types are also used in other applications. For example Graf et al. [4] use refinement types to check the exhaustiveness of pattern matching rules over complex (G)ADT types in Haskell. To check the exhaustiveness of patterns in Liquid Rust with ADTs may require similar approaches.

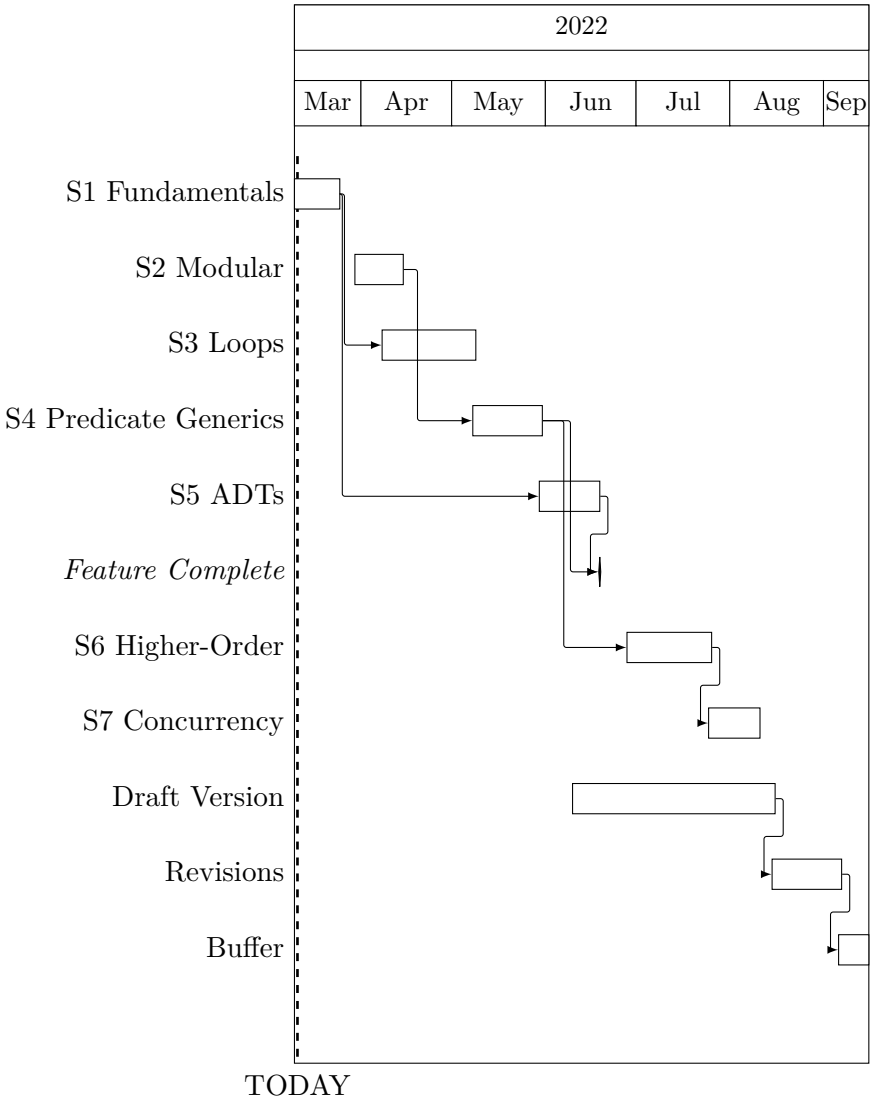
In terms of alternative verification approaches, Prusti[5] is notable, because of their work on formalizing the full Rust semantics, including `unsafe`. Prusti is a heavy-weight functional verification framework for Rust; based on separation logic.

Alternative verification approaches also exists: For example RustHorn[6] employs constrained horn clauses based verification to Rust. Particularity relevant for this thesis is the novel formalization for mutable references used in the paper. The authors stipulate that mutable references should be specified by a pre- and post-state from before a reference is borrowed to after it is returned.

The notion of Abstract Refinement Types that this thesis is based on is defined by Vazou et al. [7]. The basic idea is to allow the programmer to ”refine” language types with predicates from a decidable logic. The type system has a notion of subtyping for refined types, where one type is a subtype of another if one predicate implies the other.

5 Schedule

This sections describes the relation of the steps from Section 3.3 to their timeslot and the dependencies between them in form of a gantt diagram. There are one milestones, namely completion of the critical stages, which is denoted *Feature Complete*.



Appendices

A Use Case Analysis

To gain some understanding of how relevant mutability is in Rust, all published Rust crates (Rusts version packages) published on crates.io with at least 10 versions were analysed[1]. The analysis uses the syntactical structure to infer mutability information about the following various AST items:

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

A total of around 52 million items were found in 228263 files in a combined code-base size of over 64 million lines of Rust code (without comments and white space lines)[2]

Figure 1 shows the ratio of mutable to immutable items. Note that about 30% of local variables are defined mutable while just 10% to 20% of parameters are mutable and less than 10% of functions have mutable parameters at all.

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

[2] Calculated with `cloc`

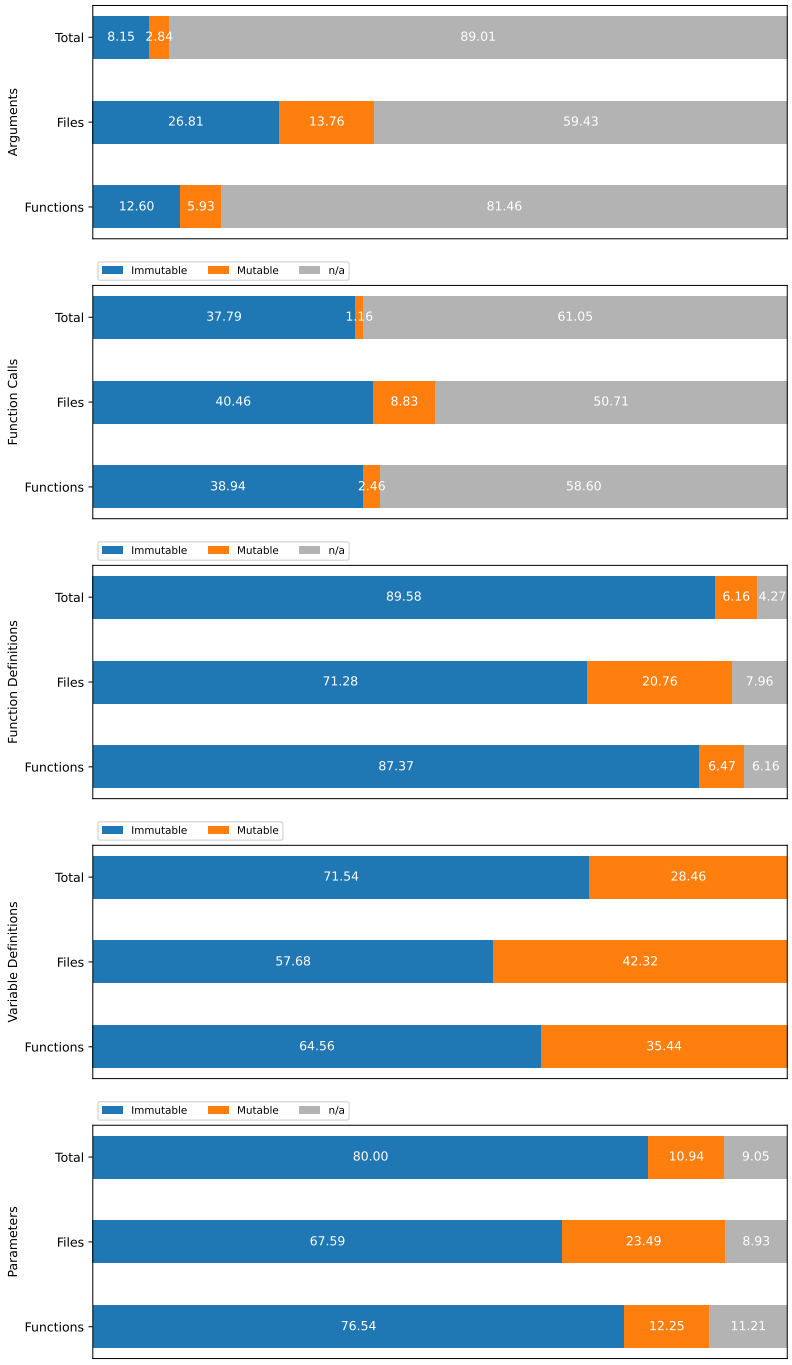


Figure 1: Ratio of Immutable to Mutable Versions of Different AST Items. Items are Counted by Unique Occurrences

References

[1] 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.

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

[3] 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 27/01/2022).

[4] 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 2nd Aug. 2020), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3408989. URL: https://dl.acm.org/doi/10.1145/3408989 (visited on 15/03/2022).

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

[6] 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.

[7] 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.