Exposé Masterarbeit

Carsten Csiky

1 Exposition

1.1 Why?

In systems programming, there has always been an interest in both performance and correctness. As these pieces of software are often the foundation of critical infrastructure.

The classical approaches to ensure correctness uncluse extensive testing, long experience and bmc, because these approaches are approachable to programmers as well as understandable in the programming context (i.e. no seperate language)

Rust – a relative newcomer to the systems programming field – has the goal of guaraneeing memory and thread safety at runtime by extending the type system. This approach works surprisingly well.

User-defined functional requirements however cannot be checked by rusts type system and require users to use testing / fuzzing to check for correctness.

Testing is inadequate in terms of accuracy, while heavy verification systems are inadequate in terms of accesability to developers: Both workflow and the specification languages are hard to adapt to existing programming workflows.

In contrast developers are used to interacting with statically typed languages and their type warnings. It is therefore desirable to embed functional verification systems in type checking phase a compiler.

1.2 What?

The thesis will enable automated verification of rust programming against refinement type specifications.

The verification should be sound and feasible in the identified use cases, which will be tested on minimal – yet representative – examples. The ex-

amples will include full refinement type annotations, consequently making type inference optional.

The specification will be written in an extension of the rust type language, inspired by refinement types.

1.3 How?

This section starts with outlining the fundamental ideas (denoted "I1" to "I4") the thesis will be based on.

- I1 adapt Refinement Type semantics to Rust, especially for mutability in Rust's Ownership Model
- I2 Treat owner and immutable types as logical variables (i.e. referncially transparent, unchanging)
- I3 Treat mutable references as state transitions from start of end of borrow
- I2 develop a prototype for translating refinement type judgments to a verification backend (e.g. z3, prusti)
- I3 conduct a feasibility study on different categories of common use cases

The following enumeration lists the concrete tasks (denoted "T1" to "T6") with their respective deliverable. These tasks are connected to specific time in Section 4.

- T1 identify common use cases for refinement types and categorize them, in particular use cases with mutable references. The deliverable is a set of minimal – yet representative – examples, with explanations for why they are representative.
- T2 devise a system for type specifications, which would ideally cover the identified use cases. 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 implement a parser for the refinement language defined above. The product is a parser that can parse the refinement language.

T4 transform refinement types and Rust AST to a verification backend. The product is an implementation, or alternatively a description of the process.

T5 if necessary for the use cases, a definition of suitable axioms for language build-in types. The deliverable is a set of refinement type coersions.

T6 conduct a feasibility study on the examples of the use cases in T1. The result is a set of proven examples. If some examples could not be proven, the reason will be investigated and explained.

2 Defining the Goal

The goal of the thesis is to show that Refinement Types can be idiomatically adapted to mutable languages when leveraging a strong Ownership Model. It aims to enable gradual adoption of leightweight verification methods to Rust.

I aim for automated parsing of refinement types and source code translation. As well as giving a description of the syntax and semantics of the constructs, that will be added to Rust.

A feasibility study on the previously defined use cases will show how usefull the proposed additions are. For use cases that could not be covered by the porposed additions, the reason will be investigated. In case implementing these translations could not be completed, the use cases will be transformed manually.

Specifying or verifying complete Rust modules or the entire Rust languge is not the goal of the thesis.

3 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, asynchroneous execution etc. and secondly other verification frameworks for Rust.

For example, Lanzinger [1] 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.

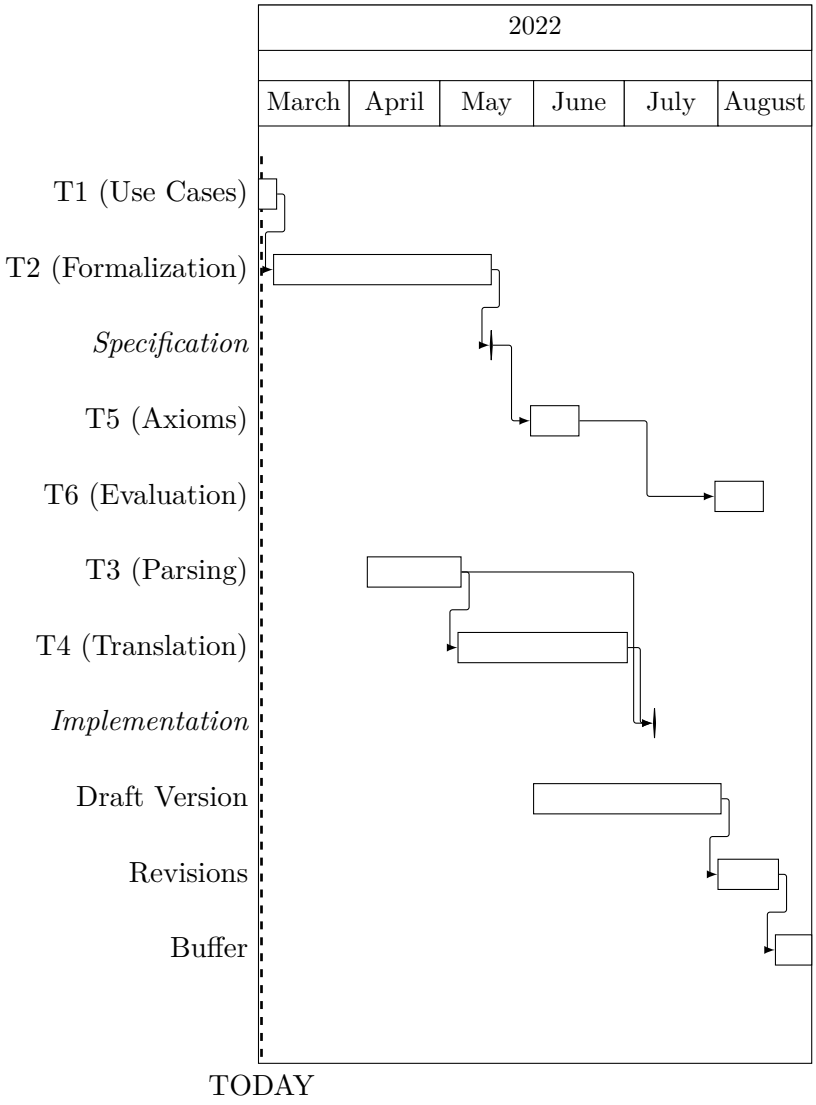
Koos et al. [2] extended refinement types to mutable and asynchronous programs. The paper explores how changes to possibly aliased memory cells can be tracked throughout a OCaml program. For that purpose the types are extended by a set of requirements on memory objects, which track distinktness and refinemented 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.

In terms of alternative verification approaches, Prusti[3] 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 seperation logic.

Alternative verification approaches also exists: For example RustHorn[4] employs constrained horn clauses based verification to Rust. Particularily 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 poststate from before a reference is borrowed to after it is returned.

4 Schedule

This sections describes the relation of the tasks from Section 1.3 to their timeslot and the dependencies between them in form of a gantt diagram. There are two milestones: Completion of Specification (denoted *Specification*) and the completion of implementation (denoted *Implementation*).


2022
March
April
May
June
July
August
T1 (Use Cases)
T2 (Formalization)
Specification
T5 (Axioms)
T6 (Evaluation)
T3 (Parsing)
T4 (Translation)
Implementation
Draft Version
Revisions
Buffer
TODAY

Appendices

A Use Case Analysis

To gain some understanding of how lambdas are used in java, the 1000 most starred java repositories on GitHub were scanned for lambda expressions. In total 151 thousand lambda expressions were found in 37 thousand files in a combined code-base size of over 51 million lines of java code (without comments and white space lines), in about five million methods. Excluding tests, 73 thousand lambda expressions were found in 22 thousand files.

First the general context – meaning the parent node in the java AST – of these lambda expressions was determined (shown in Figure 1). The context is categorized into "Invocation of a Method by Field" (call of a Method on an Object by reference); "other Invocation" (e.g. call of a local method by name); "return statement"; a "constructor call"; "assignment" and finally a catch-all category: "any other Context".

More than 80% of all uses are in an invocation context. Thus making it the primary use case.

Figure 1: Lambda Expressions by Context

To get a better understanding of what kinds of invocations can be expected the data was analysed by the method's name (visualized in Figure 2). The x-axis labels are the names of the Methods; RETURN being an exception and standing for a return statement context.

`map`, `forEach` and `filter` make up about 25% of all uses, but the rest of all uses quickly drops below 2%.

For this thesis that means, that the classical use case of `map`, `forEach` and `filter` is an important one, but certainly not the only. Consequently the solution should be general and expandable to cover the 75% of other use cases.

Figure 2: Lambda Expressions by fine Context

A.1 Comparison to Cok's Observations

Comparing the observation made here and the ones make by Cok et al. : In this data set from GitHub, functions as arguments made up about 85% of lambda uses; `Collection` uses about 18%; while functions as variables only

accounted for about 6% and return statements about 5%. In the data set from GitHub `Collection` uses are three times higher than Cok et al. saw.

In contrast to Cok et al. in the data set from GitHub, only 2% of Methods had FP features, while Cok saw 20%. This difference might originate from different definitions what constitutes a FP feature. In this analysis, only lambda expressions and function references are considered FP features; Cok's exact definition could not be found in the paper. Another reason could be the different code bases. Method length is likely not the reason, since in the data set from GitHub, method length did not correlate with having FP features.

References

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

[2] 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).

[3] 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).

[4] 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.