Contract Checking at Runtime in Rust

Master's Thesis Project Description

Cedric Hegglin
Supervisors: Aurel Bílý, Jonáš Fiala, Prof. Dr. Peter Müller

Start: April 5, 2023
End: October 5, 2023

1 Introduction

Prusti[1] is a formal verifier for the Rust programming language[2]. It can statically prove the absence of panics and certain functional correctness properties. Users can annotate functions with contracts in the form of pre- and postconditions, and Prusti will attempt to proof their validity. In theory, this should guarantee that these contracts are valid in all possible executions of the verified program.

In practice, however, these guarantees are often limited. A user running a Prusti verified Rust library in Python might unknowingly not comply with a function's precondition. In that case our Rust code might still panic or produce a wrong result. Similarly, when verifying any real-world program with Prusti, users usually rely on having so called *trusted* methods. A method being trusted means that whatever specification is attached to it is simply assumed to be correct. Examples of situations where one requires trusted methods are calls to foreign libraries, but also other Rust dependencies that can not be verified. Again, when running our program, if the trusted method does not uphold the specified contract, all of the guarantees in the verified parts of our code are lost too. In turn, users might lose trust in the value of verification, which motivates the idea of checking these contracts at runtime. When verified code is called from a non-verified context, we have to make sure it is called correctly, and when calling trusted methods from within verified code, we want to make sure it behaves as specified.

Additionally, runtime checks can also be helpful as a debugging tool in the process of verifying code. When a program is only partially annotated and can not be verified yet, the verifier itself offers very little help on determining a specification's correctness. Runtime checks on the other hand could already warn a user if a specification is incorrect.

The main challenges for doing this automatically are to translate Prusti's specification language to runtime checks, and to insert these checks into the generated executables automatically.

2 Approach

The road-map for this project can be split into a theoretical part and a practical part. The theoretical part is defining the translation of Prusti's specification language to runtime checks in Rust. The practical challenge is to then generate and automatically insert these checks automatically into the executables produced by Prusti.

In general, it makes sense to insert runtime checks in the following cases:

- When calling trusted functions from verified code, to check their postcondition after the call. This could be a method from a library that we annotated with `extern_spec`, or a wrapper for some foreign function.
- When calling verified functions from unverified code, to check that the precondition is satisfied on entry of the called function. For example, when publishing a library verified with Prusti, insert runtime checks for the precondition at the beginning of each public method.
- If a trusted function has a pledge attached, to check the Boolean expression as soon as the receiver reference of that pledge is dropped.
- If the verification of a method fails, check all its specifications to help a user to identify wrong annotations. In this case, postconditions can also be inserted into a method's body instead of at the call site.

In the other cases, Prusti's proof already guarantees the correctness of the contracts. In this case, inserting runtime checks could still be useful for detecting potential bugs in Prusti. Since not having to distinguish all of the above cases simplifies things, our initial approach will be to check as many specifications as possible. In some cases however (as will be shown in Section 2.1), runtime checks also put additional constraints on the used types in specifications. Therefore, allowing the user to configure which specifications should be checked could still be useful.

2.1 Translation

To design a translation of Prusti contracts to runtime checks, we need to create rules for the various components of its specification language. A large portion of the specification language is simply Rust syntax for Boolean expressions and some syntactic sugar for common operations such as implications. In these cases, the translation to runtime checks in the form of assertions is trivial as seen in Listing 1.

```
#[requires(i < self.length)]
fn lookup(&self, i: usize) -> i32 {
+   assert!(i < self.length);
    ..
}

#[requires(a ==> b)] // syntactic sugar
fn bar(a: bool, b: bool) {
+   assert!(!a || b);
    ..
}
```

Listing 1: Example of assertions to be added (green) to check the simple given preconditions.

However, for the more powerful features, the translation can quickly get a lot more complicated. To demonstrate this, we will consider the method shown in Listing 2, containing an `old` expression and a `forall` quantifier in its specification. It contains a method `push`, that adds an element to a collection. The specification expresses that all original elements remain unchanged. Since this method is trusted, we want to check its postcondition at runtime whenever it is called. This time, however, we can not perform the check by only adding assertions. To check this contract, we need to store all the `old` expressions that need to be evaluated to check the quantifier. Listing 3 shows how a call to this method would have to be extended.

```rust
#[trusted]
#[ensures(forall(|i: usize| i < old(self.len()) ==>
        old(self.lookup(i)) == self.lookup(i)
))]
fn push(&mut self, el: i32) {
    ..
}
```

Listing 2: Example of a more complex specification using the `old` keyword and a quantifier.

```rust
// storing "old" values
let old_self_len = v.len().clone();
let old_self_lookup_i = Vec::new();
for i in 0..old_self_len {
    old_self_lookup_i.push(v.lookup(i).clone())
}
// the actual call:
v.push();
// the runtime checks:
for i in 0..old_self_len {
    assert!(v.lookup(i) == old_self_lookup_i[i])
}
```

Listing 3: The saved values and runtime checks that are necessary to check the contract of Listing 2 whenever `push` is called.

One limitation of this approach is the requirement that whatever type is used within an `old` expression has to implement the `Clone` trait. Moreover, the quantifier in this example has a very convenient form. It contains an implication whose left-hand side limits the number of values we need to loop through. Without this limitation, these kinds of runtime checks can be practically uncheckable or are at least undesirable due to the timing overhead they introduce. This raises some design questions that will need to be addressed. A user might want to limit the number of loop iterations that are introduced by a single quantifier. Non-exhaustive runtime checking might still provide some value, for example using simple heuristics such as checking the boundaries only.

While we only showed examples of translations so far, the result of this part of the project will be a set of rules on how to translate the various features. This must also support arbitrary combinations and nesting of the supported expressions. Nevertheless, for a few specification features we are not certain yet, whether runtime checks will be in scope for this thesis. These include snapshot equality and closure specifications.

2.2 Instrumenting Executables

Assuming that we have determined the appropriate runtime checks to include, our next challenge is to incorporate them into executables. Prusti already uses the Rust compiler's interface to read various internal data structures and can use it to compile programs after verifying them. Our goal is to modify the generated executables by adjusting the control flow graph of the MIR (mid-level intermediate representation). When the MIR is first built, before any optimizations are performed, we want to identify all method declarations and calls of interest and instrument them with runtime checks. There are multiple reasons for why we want to operate on the MIR level. First of all, the type information on this level is explicit, which means all function calls can be resolved and we can insert checks depending on the type of the involved variables (recall the requirement of `Clone` for `old`). Additionally, this is the level where Prusti already performs most of its analysis, which should simplify our integration. One downside of this approach is that, since the MIR is control flow graph based and low-level, it is quite verbose and hard to work with.

Unfortunately, the interface of the Rust compiler does not provide a lot of functionality to support such modifications, which is why this will require a fair bit of engineering. Although this section is rather short, this will be the main part of the project in terms of effort. It will require a deep understanding of the compiler, the MIR and the inner workings of Prusti.

3 Goals

While the previous section should have provided an idea of the technical challenges, the following sections will provide a more structured overview of our goals for this project.

3.1 Core Goals

- **Design specification translation**
  Design the theoretical translation of Prusti features into runtime checks. This translation will consist of a set of rules on how to translate the Prusti specification syntax, for example quantifiers or old expressions, to executable Rust code. Additionally we also need to design rules for when and where to insert these checks for the various types of specification items such as preconditions, postconditions, pledges and various others. In general, one goal of this project is to include runtime checks for as many specifications as possible.

- **Instrumentation of generated executables**
  Within Prusti, use the Rust compiler's interface to to modify the generated executables in a sensible way. It has to allows us to insert the desired runtime checks automatically, but optimally it can also be used in different settings. One example of a different purpose are the optimizations based on verification listed in the extension goals.

- **Implement the translation and insertion**
  Use this instrumentation of the Rust compiler and implement the automatic translation and insertion of runtime checks based on Prusti's contracts.

- **Qualitative Evaluation**
  We will demonstrate the functionality of our implementation on a set of examples. This may include examples of calls to foreign libraries from Rust and calls to verified Rust libraries from other languages.

3.2 Extension Goals

- **Support for "hard" specification features**
  For some features of Prusti and its specification language, it is rather unclear how we could support runtime checks. Examples include snapshot equality and closure specifications.

- **Optimizations based on the verification results**
  During verification, Prusti gains additional information about a program. This could allows us to perform optimizations a traditional compiler would not be able to find, such as eliminating an unreachable match arm for example. The added instrumentation of the compiler could allow us to perform such optimizations.

- **Combining contract checking with a testcase generator**
  Since runtime checks are a way of automated testing, it could be interesting to combine them with a testcase generator. Runtime violations of preconditions could be used to eliminate testcases, and the remaining assertions can be used as oracles to find errors.

- **Error reporting**
  Failing runtime checks, as we have described them so far, would simply result in failing assertions. A user will see the expression that failed, but it would be better if the resulting panic provided some information about which specification was violated to cause this error.

- **Conversion of equivalent specifications for efficiency**
  Certain specifications including quantifiers can express equivalent properties in multiple ways, but result in runtime checks with vastly different performance. For example, sortedness of a collection can be expressed using a quantifier over two variables conditioned on one being larger than the other, or only one variable and its successor. Assuming transitivity, both express the same property. But the resulting runtime check's complexity is quadratic in one case and linear in the other. With a set of lemmas one could recognize and convert certain types of these equivalences, and allow for more thorough runtime checks.

References

[1] V. Astrauskas et al. "Leveraging Rust Types for Modular Specification and Verification". In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Vol. 3. OOPSLA. ACM, 2019, 147:1–147:30. DOI: `10.1145/3360573`. URL: `http://doi.acm.org/10.1145/3360573`.

[2] Nicholas D Matsakis and Felix S Klock II. "The rust language". In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.