Annotating the Rust Standard Library with Specifications for Use in a Rust Verifier

Master's Thesis

Julian Dunskus

Supervised by Aurel Bílý, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zürich
Zürich, Switzerland

February 2023

Abstract

Prusti is a verifier for the Rust programming language aiming to improve accessibility of program verification by being close to a Rust model. A major hindrance in adapting an existing project for verification with Prusti was that, in order for Prusti to understand calls to library code, they had to be manually specified in an ad hoc fashion. We improve this situation by offering specifications for the most-used parts of the Rust Standard Library, allowing users to rely on and contribute to a central repository of specifications rather than sourcing or writing them manually. To determine which methods to specify, we gathered and evaluated data from the large amount of code in Rust's central package repository, `crates.io`. All told, the new specifications cover around one fifth of all method calls across this dataset, and we further categorized the vast majority of currently unspecified calls to inform future specification efforts.

Contents

Chapter 1

Introduction

Avoiding bugs in code has long been a goal of any software development process. To this end, programming languages are becoming ever more expressive, allowing users to express complex relationships between variables and gain compile-time guarantees of their code's safety. Rust [1] is a modern systems programming language that prizes memory safety above all else, and leans heavily into these static safety guarantees.

Nevertheless, there remain many properties of code that cannot be expressed in the type system or otherwise checked at compile time. This is where deductive verifiers come in, building on top of the language's facilities to allow users to express additional properties of their code, which can be verified exhaustively without needing to run the code or otherwise test it.

One such verifier is Prusti [2], developed by the Programming Methodology Group at ETH Zurich. It is based on the Viper [3] verification infrastructure, translating Rust code input to Viper code and verifying that using the Viper verifier, then relating the results to the original Rust source. It can verify e.g. `assert!(...)` assertions in code as well as pre-/postconditions on functions (declared via Rust's attribute system).

In order for Prusti to understand a piece of code, it has to know how any functions called within it behave, which it can gather from annotations users place on their declared functions. However, real-world Rust code depends heavily on external methods and types that are not part of the user's codebase, so specifications for their behavior need to be available.

The most central such repository of external code is the Rust Standard Library, providing essential features like optionals (`Option`), error handling (`Result`), and functional primitives (`map`, `filter`, etc.). Being able to verify programs using this functionality with minimal setup would lower the barrier to entry for using Prusti and make it easier to apply to existing codebases, proliferating verification of Rust code.

Currently, if users want to verify programs using this code, they must search for or write these specifications themselves. A motivating example of such a specification for (part of) `std::option::Option` might look as follows:

```rust
#[extern_spec]
impl<T> std::option::Option<T> {
    #[pure]
    #[ensures(result == matches!(*self, Some(_)))]
    pub fn is_some(&self) -> bool;

    #[requires(self.is_some())]
    pub fn unwrap(self) -> T;
}
```

Rather than expecting users to write such specifications ad hoc whenever they need them, likely copying them over from other sources, we aim to provide a central repository of specifications as part of the Prusti project. These specifications can be added by declaring a single dependency, with the most fundamental ones shipping with Prusti directly. This repository also acts as a hub for users to contribute their own specifications to, expanding the usefulness of Prusti and reducing the effort required to integrate it into an existing project and start verifying.

This report begins by providing background knowledge for the involved concepts, describing the initial state of Prusti (**Chapter 2**). It then outlines the approach we settled on to focus our efforts on the right targets and to write extensible specifications (**Chapter 3**). The bulk of the report falls into **Chapter 4**, describing the process of implementing this approach, going into detail on groups of related methods, along with the difficulties encountered and insights gained along the way. Next, we evaluate the progress we have made in retrospect, visualizing the coverage achieved by our specifications (**Chapter 5**). Finally, we conclude by summarizing our findings and their relation to Prusti's goals, as well as laying out opportunities for future work that we discovered (**Chapter 6**). For completeness' sake, we also include a list of the issues opened and code contributions made as part of this project (**Appendix A**).

Chapter 2

Background

This chapter lays out the necessary background knowledge to understand the later chapters. It is grouped into sections, so that readers may skim through or skip over sections describing concepts they already understand. Extern specs and type-conditional spec refinements have received major changes as part of this thesis, so this chapter describes their initial state before said changes. A notable exception is syntax—in an effort to improve clarity, all examples here and throughout the report consistently use the final syntax. Later chapters will elaborate on the specific changes made.

2.1 Extern Specs

When using functions from external libraries (like the Rust Standard Library, or any dependencies of the project), it is not possible to simply write a specification at the function definition. To work around this, Prusti offers a feature called extern specs. These allow defining specifications for any external trait, `impl`, or free-standing function. They are written as the item in question with an `#[extern_spec]` attribute attached, e.g.:

```
#[extern_spec]
trait Default {
    #[pure]
    fn default() -> Self;
}
```

Trait specifications like this will be checked for all implementations in user code, and assumed to hold when using a type implementing the trait.

We can also specify the behavior of a specific implementation of a trait. In this particular case, we can describe the exact value that `i32::default()` returns:

```
#[extern_spec]
impl Default for i32 {
    #[pure]
    #[ensures(result == 0)]
    fn default() -> Self;
}
```

Trait methods already do not allow bodies in standard Rust, and in `impls` or free-standing functions, where a body would usually be required, extern specs explicitly disallow it. They only serve to specify the behavior of a function, not its source, so instead methods are only declared and not defined, followed by a semicolon instead of a body in braces.

Free-standing/top-level functions are written as part of their module, as importing it into scope would cause a name conflict. This requires re-importing the Prusti macros and makes it awkward to interface with code outside the module scope, e.g. a supporting trait for conditional specifications (illustrated in Subsection 4.2.5).

```
#[extern_spec]
mod core {
    mod mem {
        use prusti_contracts::*;

        #[ensures(*x === old(snap(y)) && *y === old(snap(x)))]
        fn swap<T>(x: &mut T, y: &mut T);
    }
}
```

2.2 Type-Conditional Spec Refinements

When specifying a function, we are often able to express its specification more precisely given certain conditions on the involved types. This is achievable through type-conditional spec refinements, implemented under the name "ghost constraints" [4] for a previous thesis.

For example, it is not possible to specify the exact values of `size_of` for different type arguments without baking support for it directly into Prusti logic, treating it as a built-in operation. However, we can define sizes for the types on which people most commonly use this function, e.g. the built-in numeric types. For this, we can define a new trait `KnownSize` whose requirements encode this information as follows:

```
pub trait KnownSize {
    #[pure]
    fn size() -> usize;
}
```

```
impl KnownSize for i32 {
    #[pure]
    fn size() -> usize { 4 }
}
```

We could already specify the behavior of `core::mem::size_of` as `pure` in the general case, since its output is the same when called repeatedly. But thanks to this new trait, we can conditionally further refine this specification when `T` is known to implement `KnownSize`, specifying exactly what the returned size is:

```
#[extern_spec(core::mem)]
#[pure]
#[refine_spec(where T: KnownSize, [
    ensures(result == T::size()),
])]
fn size_of<T>() -> usize;
```

2.3 Shipping Specifications with Prusti

There has been some prior work on publishing specifications with Prusti, which this project benefits from. For one, extern specs were extended to apply across crate boundaries, enabling the concept of "specification library" crates providing spec support for other crates to use in dependents' code [5]. In this process, Prusti was already set up to ship specifications for the Rust Standard Library, specifying a few methods[1] as proof of concept. Specifications were split into a crate for `core`, the core of the Rust Standard Library that is necessary for the language to function, and one for `std`, the more expansive standard library. This allows projects to eschew the latter, minimizing executable size to run in environments with tight space constraints [6].

2.4 Qrates

Qrates [7] lets us extract semantic data from crates published on crates.io, the primary repository of crates (packages) in the Rust ecosystem. Qrates fetches crates from said repository, compiles them using Docker, and builds a database from information it outputs during the build process. Notably, this allows us to answer questions like which methods or types are used the most across all published crates.

More specifically, Qrates works by wrapping `rustc` to gather data during the compilation process. In a separate step, this raw data is stored into a database for further processing. This database is represented as Datalog relations, allowing complex fixed-point queries (though these are not necessary for our purposes). Finally, there is a variety of queries implemented in Rust that use this database (and a Datalog engine[2]) to evaluate relations and output CSV files.

2.5 Related Work

There has been similar work done for other languages that provides valuable experience in how to approach these problems. In Python, users can add type annotations to their code to achieve some level of static typing, and there is a large repository providing types for different packages called typeshed [8]. Similarly, TypeScript extends JavaScript with a type system, and there is a de-facto standard repository of type annotations for different libraries called Definitely Typed [9]. Our goal of providing functional specifications for existing libraries is fundamentally analogous to these projects' type specifications.

[1]Specifically, previous specifications were limited to `Result::is_ok`, `Result::is_err`, `Result::unwrap`, `HashMap::get`, and `HashMap::contains_key`.

[2]https://github.com/rust-lang/datafrog

One insight from these projects is that they manage the complexity of maintaining annotations for the large collection of libraries available by providing separate packages for each such library, rather than requiring users to pull in code for libraries they are not using. As this thesis only concerns itself with the standard library, such scalability considerations will only come up in future work. Regardless, these projects make for useful references, offering existing solutions for problems that may arise in work on Prusti specifications as well, e.g. supporting multiple versions of a crate spanning breaking changes.

Chapter 3

Approach

3.1 Prioritizing Most-Used Specifications

Specifying every single item in the Rust Standard Library is an enormous task, which would not fit within the time constraints of this project—instead, we elected to specify only the most-used parts. Further specifications will build up over time as people contribute specifications they find useful or other student projects explicitly set out to write more.

Figuring out what those most-used parts are, however, is far from trivial. Of course, any Rust user will have developed a certain sense of what the most fundamental types are, like `Option<T>` or `Result<T, E>`, and those are clear primary targets for specification. However, even these types already have a vast collection of methods, many of which are only rarely used.

As such, we base our decisions on an analysis of usage data from published Rust code. We gather data from public crates on crates.io, the primary package repository in the Rust ecosystem. The Qrates framework [7] provides the tools for this exploration, allowing us to gather usage data into a database and formulate queries on it. Qrates does not have a query built in that provides the exact data we need, requiring extensions for our purposes. With these results in hand, we can perform data science and use the gained insights to inform decisions about what to prioritize.

Qrates lets us extract more specific information than just which methods are called from where, most importantly allowing us to resolve (some) calls to trait methods to their concrete receiver type. For example, a central part of Rust's error handling story is the `Try` trait, implementing the postfix `?` operator for early-exit control flow, akin to traditional exceptions propagating up the stack. In this case, the trait itself does not provide the necessary information to express whether execution will continue along the current path or abort with an early return[1]. Thankfully, we can identify not only how many calls occur to the trait methods that this syntax desugars to, but even what fraction of the calls use `Result`, `Option`, or another type as receiver. Thus, we have an accurate idea of the coverage of our specifications, revealing any important missed cases.

Further, we can find out what generic arguments are involved in method calls, both to the function itself and to the receiver (if applicable). This is important for e.g. `core::mem::size_of<T>()`, a method that returns the memory footprint of a given type in bytes. Barring specific support for this method as a Prusti builtin[2], it is infeasible to specify the behavior for every single type. Hence, it is important to know the generic arguments involved in calls to this method, in order to identify the types whose size is most important to specify.

As of January 2023, crates.io has around 100,000 crates, which is too large a number to quickly test Qrates changes against. Instead, our analysis initially used the top 1000 most downloaded crates as a presumed-representative sample—these only take around an hour to build and extract data from. The insights gained from these turned out to hold true for the full dataset, so to avoid confusion we will only refer to the latter in this report.

3.1.1 Key Insights

For starters, we can limit our analysis to calls across different crates, since within a crate, specifications are written by the user directly at the definition site, rather than relying on external ones shipped with Prusti. Further, most intra-crate calls are to methods that are not public and thus could not be specified externally. Thus, in all our evaluation, we will implicitly ignore calls within a single crate. Further, note the distinction between *methods* and *call targets*—the latter being the combination of a method and its receiver, which is relevant for trait methods, which may be implemented by many types.

[1]This depends on the specific value in question, with e.g. `Result::Err` causing an exit while `Result::Ok` continues execution.

[2]This may make sense as future work, but our current implementation requires no additional support and already provides a lot of value to users, covering the vast majority of concrete calls.

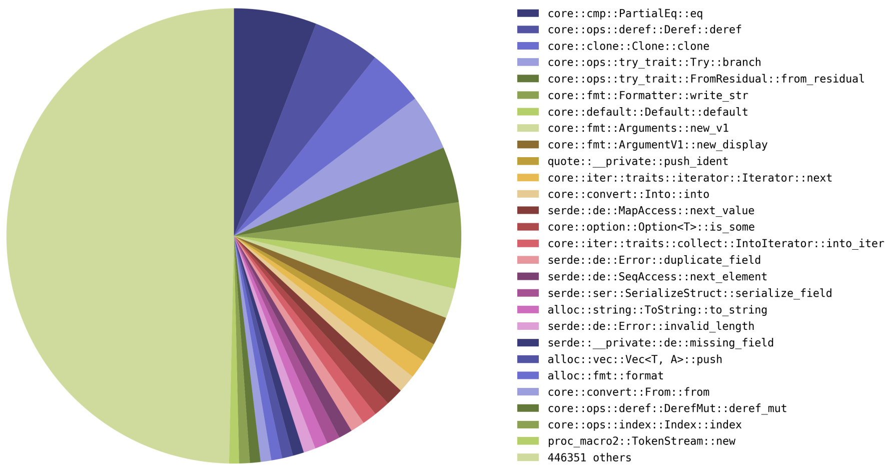

Figure 3.1: The frequency of the top 27 most-called methods, along with the sum of all other methods for comparison.

Of course, not all methods are called with the same frequency. In fact, the disparity is very large (Figure 3.1)—so much so that specifying just the top 27 methods ($\approx$ 0.0060% of 446,378 total) would already cover the majority of all call sites in our dataset. This trend remains when additionally considering the method's receiver: here, the top 87 call targets ($\approx$ 0.0035% of 2,512,426 total) would require specifications for the same effect. As a result of this, just a few specifications will already greatly enhance Prusti's usefulness in verifying everyday code (Figure 3.2).

As demonstrated here, in order to achieve good coverage of call sites and thus be helpful for users, it is essential that we prioritize the right methods for inclusion in Prusti's library of specifications. To this end, we can use the frequency data to answer questions not only about which methods are called most, but also what the most important receivers are for trait methods, and what the most common generic arguments are when those are relevant.

3.2 Flexible Trait Specifications

For traits in the Rust Standard Library, there is often a set of contracts that is not fully expressible in the type system but expected to be satisfied by implementers. These contracts can often be expressed and verified using Prusti. However, it is not necessarily desirable to simply impose these contracts on every implementer, since some might deliberately violate them. This would trigger failures when verifying an

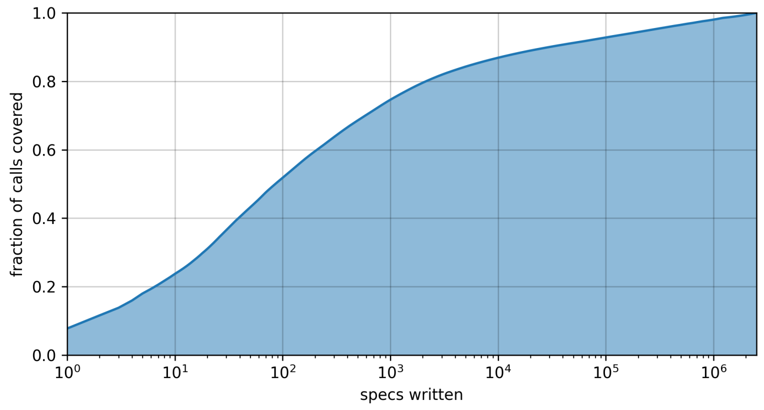

Figure 3.2: The theoretical coverage achieved by specifying the top x call targets—note the logarithmic scale on the x-axis.

implementation that violates the contracts, as well as potentially creating unsound behavior when they are assumed to hold for an unspecified implementation. As such, a way to opt into or out of such specs allows for more precise contracts that are more restrictive but still satisfied in most implementations.

As an example, consider the trait `Default`, which defines a single static method creating a new instance of the given type as a default value, e.g. 0 for integers. We would like to use this trait to specify `Option<T>::unwrap_or_default`, which unwraps the optional, falling back on the type's default value if it is `None`. Ideally, we could describe that in the latter case, `<T as Default>::default()` is called and the result returned, but that would require future Prusti features like call descriptions [10]. Failing that, we can assume that `default()` is pure[3], always returning the same value (or one that is indistinguishable under Prusti's snapshot encoding [11]). However, this might not hold for all implementations of `Default`, so there has to be a way to opt out of this expectation.

The fundamental tool to express this opt-in/opt-out mechanism is type-conditional specification refinement. This allows us to conditionally refine a specification with additional pre- or postconditions (or a purity annotation) when certain type constraints hold. In the running example, we can introduce a new trait `PureDefault` to express the additional constraint that `Default::default()` is pure:

[3]If it were not pure, we simply could not use it in pre- or postconditions currently.

```
pub auto trait PureDefault {}

#[extern_spec]
trait Default {
    #[refine_spec(where Self: PureDefault, [pure])]
    fn default() -> Self;
}
```

By marking it as an `auto trait`[4], we have created an opt-out mechanism, where every implementation is assumed to satisfy these more specific contracts unless it explicitly opts out by adding a so-called negative `impl`, syntactically e.g. `impl !PureDefault for Foo {}`.

Better yet, Rust only applies auto traits automatically when all the type's fields implement the trait, so nesting a non-`PureDefault` type within an unspecified one will opt the latter out of the trait by default. Users can still decide to opt back into the trait, e.g. if they have customized the implementation to ensure purity despite the impure nested implementation.

We can combine this addition with another refinement on `unwrap_or_default` to specify it:

```
#[ensures(old(self.is_none()) || old(self) === Some(result))]
#[refine_spec(where T: PureDefault, [
    ensures(result === match old(self) {
        Some(v) => v,
        None => Default::default(),
    })
])]
fn unwrap_or_default(self) -> T where T: Default;
```

This finally gives us the desired specification outlined above, along with a simple fallback when the more precise contract is not known to hold.

[4]https://lang-team.rust-lang.org/design_notes/auto_traits.html

Chapter 4

Implementation

As outlined before, we started out by gathering data to determine which methods to prioritize specifying. We then moved on to specifying methods identified to be important. This chapter lays out the process of gathering this data (Section 4.1) and writing the specifications (Section 4.2), delving into the complexities these goals entailed, including limitations in Prusti along with their workarounds or remedies (Section 4.3).

4.1 Data Gathering

Qrates comes with an assortment of predefined queries, Rust code querying the database and gathering rows of data into a CSV file. The database already contained relationships encoding method calls, but this data proved insufficient for our use case. Indeed, figuring out what data is relevant to identify a method call and how to encode it meaningfully is far from trivial. For each call, we gather descriptions of: the receiver (if any) and its generics, the path to the called function, generic arguments to the function and the type it is defined on (if any), as well as the crate of the call site and of the called function (the latter technically being repeated from the function path).

For all this to work, we also had to create a mechanism to meaningfully represent arbitrary types as text. This task again turned out to be rather involved, as we could not simply rely on Rust's built-in implementations of `Display` or `Debug`. These implementations include generic arguments, which would have to be re-parsed and separated out for meaningful data processing. They also have the issue of working relatively to the current crate, only prepending the crate name for foreign paths, which would make the data less uniform and harder to process. In the end, we implemented methods from scratch that would traverse type paths, gathering uniform descriptions of them and, separately, any involved generics.

4.1.1 Macros

If a call was written as part of a macro expansion, we also store the macro that was invoked. Otherwise, these might heavily skew results towards functions people do not interact with directly, since macros often expand into many individual method calls. However, we could not just take the top macro from the call stack, because when users define their own macros calling methods, we want to catalog these calls as if made directly.

Consider the following example:

```rust
macro_rules! my_macro {
  ($name:ident) => {
    fn $name() {
      println!(
        stringify!($name)
      );
    }
  };
}
my_macro!(example);
```

```rust
// expands to:
fn example() {
  {
    ::std::io::_print(
      ::core::fmt::Arguments::new_v1(
        &["example\n"], &[],
      )
    );
  };
}
```

The call to `Arguments::new_v1` here originates from an internal macro named `format_args_nl` used by `println!()`. Thus, its macro backtrace, from bottom to top, is `format_args_nl`, `println`, `my_macro`. The most salient of these is `println`, being the lowest-level macro in the user's code. We adapted this intuition into a general rule: look at the crates the macros are defined in and choose the topmost one defined outside the current crate. This treats the user's own macros transparently, while avoiding surfacing implementation details of any called macros.

4.1.2 Scaling

Another issue was scaling this logic to the (as of the time of writing) over 100,000 crates published on crates.io, as we wanted a balanced perspective on real-world Rust code. This involved optimizations at every step of the way, as well as some rethinking of the monolithic approach built for smaller datasets.

When running the query for method calls, the gathered rows are deduplicated before being stored in the CSV, instead additionally storing each one's number of occurrences. For data from the top 1000 crates, this takes the final CSV from 524 MiB to 39 MiB. Further, the calls are sorted by their target path to optimize for compression using gzip. The aforementioned 39 MiB CSV gzips down to 10.6 MiB without sorting, reduced to just 4.8 MiB with sorting, a compression factor of around $8\times$.

There are other difficulties in the process of compiling the crates to gather the data in the first place. This process was not designed for such large numbers of crates, and simply running it over all crates inevitably broke down as memory usage increased. This amount of crates also takes on the order of weeks to compile, making it dangerous to rely on the machine running stably for the duration. Instead, we implemented a batching system that subdivides the list of crates to compile into batches of configurable size, after each of which the generated data is extracted and the workspace is reset. Larger batch sizes incur less overhead per compilation but increase memory requirements and increase potential data loss on failure, and 1000 crates per batch turned out to be a good compromise. This system is more fault-tolerant, only requiring a single batch to be redone if the machine crashes at any point, which reduces lost time to a few hours.

Following this step, the data is processed to form a database, which has to fit into memory at once with the current setup. As it turns out, the database ends up at around the same size as the original extracted data, coming in at around 10 GiB[1] for a batch of 1000 crates. This unfortunately makes it infeasible to build a database over all 100,000 crates, so we had to build the database and run the query on a per-batch basis as well. Merging the query output proved a little tricky, because the data also includes dependencies, which would have been represented many times if naively concatenating the CSVs. Instead, based on the "caller crate" field, we iterate through the individual batches' rows and only append data for a crate from the batch where it is first seen, ignoring it in later batches.

[1]This varies a lot by crate. It also fortunately compresses by a factor of around $5\times$ as a ZIP, taking the data for all crates from over 1 TiB to a more manageable 250 GiB.

Combining all these optimizations, we end up with a final CSV for all 100,000 crates of 2.6 GiB (277 MiB gzipped). We can also strip out intra-crate calls (where caller and target crate are the same), taking it down to 2.0 GiB (234 MiB gzipped). A small caveat is that, of the 100,000 crates, only 73,181 successfully compiled and produced any entries, but this dataset is still absolutely sufficient for our purposes.

4.1.3 Evaluation

In order to evaluate this data, we employed a Jupyter notebook and the `pandas` library, as has previously been done in Qrates. Thanks to the above optimizations, the data comfortably fits into memory and processing it does not impose special hardware requirements.

The most obvious question to ask of this data is which methods are called the most, but this is a deceptively simple question. For example, for methods like `Option::map`, the concrete generics (of the type and the function) do not matter—any specification would apply to all of them. On the other hand, `From<T>::from` requires a separate specification for each combination of type argument `T` and receiver type. In the end, we filtered the data in various ways to identify important targets, relying on manual inspection to pick out worthwhile targets from these lists.

We have elected to publish the data we gathered, the scripts used to orchestrate the process, and the Jupyter notebook used to analyze it, on GitHub at https://github.com/juliand665/qrates-eval, in the hope that this will aid future efforts along the same lines and provide a jumping-off point for other research.

4.2 Specification Writing

In choosing methods to specify, we did not strictly adhere to the popularity ordering—when specifying some methods in a group (e.g. on the same type or with similar signatures), the bar to specify further methods was lowered, inviting specifications for the remaining methods. For example, it would not make much sense to specify `Option::is_some` without also specifying `is_none`, even if the former were used many times as much. (In reality, `is_some` is used only around twice as much as `is_none`, but they make for a nice example.)

The argument for this is twofold: From a usability perspective, it would feel strange for these closely related methods to have different levels of support, perhaps having to resort to negating `is_some` in place of writing `is_none` if only the former were specified. From the perspective of specification writing, the specs these methods require are almost identical, apart from flipping the two cases, so specifying both requires only minimally more effort than specifying one. While this is a contrived example, the logic scales to other groups of methods, e.g. `core::mem::size_of` and `core::mem::align_of`, or to the `Default` values or other details of all numeric types.

The rest of this section looks at interesting groups of methods, highlighting the chosen approach to specifying them, as well as any difficulties or Prusti limitations encountered along the way. Many of these lead down rabbit holes of necessary improvements to specify them the right way, and these examples should illuminate the processes and motivate the changes we made to Prusti.

4.2.1 `Option::unwrap`

This function unwraps the value wrapped in an `Option`, panicking if there is none. It is very simple, but there is already an important trap worth pointing out here:

```
#[ensures(match old(self) {
    Some(v) => result === v,
    None => unreachable!(),
})]
fn unwrap(self) -> T;
```

There is a bug in this code: the `unreachable!()` is actually reachable. This is because we never state the precondition that the value cannot be `None`. Writing `false` instead would have the same effect. If we instead wrote `true` here, we would still not be validating that the method is callable, but we would solve the main issue:

The big problem is that this is effectively an `ensures(false)`, entailed by any call to this method where the receiver could be `None`, leaving the verifier in a state where *anything* verifies, even patently false things like `assert!(0 == 1)`. Ordinarily, this specification would be verified against the implementation of the method—for which, like most code, the postcondition `false` would not hold—catching the bug. However, this is part of an extern spec, and so the method body is not available to verify against (the problem would be the same if the method were marked `#[trusted]`).

One possible solution would be to fail or at least emit a warning when the verifier reaches this `false` state. Perhaps the `unreachable!()` could be interpreted differently in this context, causing an error when it is in fact reached. It would also be valuable in general to provide the option to verify extern specs against their source with a separate invocation. This would not help users unless they went the extra mile to perform this check, but it would create a way to gain some confidence that one's extern specs are provably correct, especially valuable for us as creators of a central repository of them.

4.2.2 `Clone::clone`

`Clone`, one of Rust's most fundamental traits, provides a way to explicitly create a copy of a value. It is a reasonable expectation that any implementation of `Clone` also satisfies snapshot equality[2] between the copy and the original. However, there are bound to be exceptions, so we do not want to unconditionally impose this constraint on all implementers. This is where the flexible trait specifications designed in Section 3.2 come in. We define an `auto trait SnapshotEqualClone` that is used to express this additional constraint as a type-conditional spec refinement:

```
#[extern_spec]
trait Clone {
    #[refine_spec(where Self: SnapshotEqualClone, [
        ensures(result === self),
    ])]
    fn clone(&self) -> Self;
}
```

With this specification in place, any usage of `clone()` is known to satisfy snapshot equality unless the type explicitly opted out of the automatic implementation of `SnapshotEqualClone`.

One caveat to this approach is its interaction with Rust's orphan rule[3], which forbids implementing foreign traits for foreign types to make it impossible for multiple crates to provide overlapping implementations. This means that a Prusti user writing extern specs for a crate they do not own is unable to opt a type out of this auto trait.

[2]Snapshot equality relies on Prusti's snapshot encoding [11] to check deep structural equality.
[3]https://rust-lang.github.io/chalk/book/clauses/coherence.html

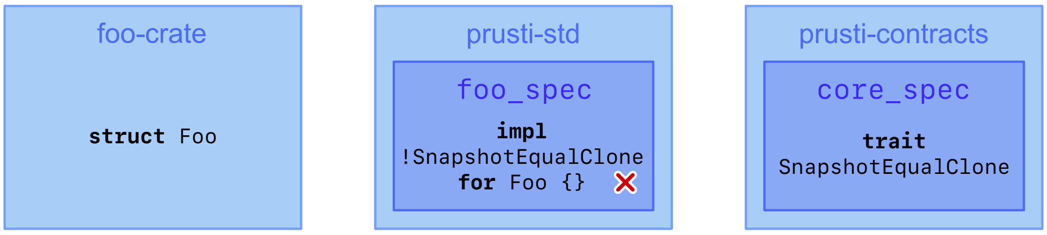

Figure 4.1: An example configuration of the crates involved in trying to opt an external type out of a marker trait, violating the orphan rule.

Even contributing these specifications to Prusti's collection does not always work with the current architecture, as illustrated in Figure 4.1. Traits that only interact with Rust's `core` crate (like `SnapshotEqualClone`) are defined directly in the `prusti-contracts` crate that ships with Prusti, as they can be included unconditionally. All other specifications live in `prusti-std`, a separate crate that people need to explicitly add as a dependency, allowing for configuration with Cargo's "features". As a consequence, the specifications for a third-party crate should also be added to `prusti-std`, where they cannot create links between the types from `prusti-contracts` and the third-party library.

In future, Prusti could be extended to either offer an alternative opt-in/-out mechanism or provide a way to circumvent this rule in limited cases like this. Until then, rather than being provided externally, such specifications can only be written part of the library itself, contributed to its source repository.

There is also an important tradeoff here compared to using a regular, non-`auto` trait. This approach is technically unsound, because it assumes the stricter constraints for external types, whose bodies are not available to verify they hold. This is a general problem with extern specs, but it is exacerbated by the automatic nature of this solution, whereas having to opt in explicitly would result in each specific case being given more careful thought as to whether it satisfies the specification. It should be noted that any client code implementing `Clone` will be verified against this constraint, failing if a user's implementation does not satisfy snapshot equality, so it is only a problem for external types. All things considered, we deemed `SnapshotEqualClone` (and other auto traits refining specifications) likely enough to hold for any type that this downside is clearly outweighed by the additional effort necessary to make an opt-in system useful.

4.2.3 `Default::default`

`Default` is a trait that defines a default value for a type, which can be constructed at any time without arguments via the `Default::default()` method. We can specify the default values for a variety of commonly used types. In order to avoid excessively verbose code, we used a macro to list default values concisely:

```
macro_rules! default_spec {
    ($t:ty, $v:expr) => {
        #[extern_spec]
        impl Default for $t {
            #[pure]
            #[ensures(result == $v)]
            fn default() -> Self;
        }
    };
}

// e.g.:
default_spec! { bool, false }
default_spec! { i32, 0 }
```

Note that the function is also considered pure, since it always returns the same value. As for `Clone`, this is likely to hold for almost all types implementing `Default`, but in order to maintain support for the exceptions, we cannot unconditionally mark `default()` as pure. Instead, we can once again employ a marker trait, introducing an `auto trait PureDefault` that expresses this additional constraint. This allows clients that are explicitly impure to state as much, while preserving the sensible default for the vast majority of use cases. The specification for the `Default` trait then looks as follows:

```
#[extern_spec]
trait Default {
    #[refine_spec(where Self: Copy + PureDefault, [pure])]
    fn default() -> Self;
}
```

As an aside, Prusti did not yet support conditionally marking methods as pure via type-conditional spec refinements, so we had to extend it. This ended up being a rather involved change, but it enabled more complete specifications for many methods.

One example that benefited from this change is `Option::unwrap_or_default`. This method returns the wrapped value if present, otherwise calling `default()` and returning its result. It would make sense to specify it as follows:

```
#[ensures(result === match old(self) {
    Some(v) => v,
    None => Default::default(),
})]
fn unwrap_or_default(self) -> T where T: Default;
```

However, `default()` is not necessarily pure, so it cannot be used in the context of a pre- or postcondition. Without it, we can only state that if there is indeed a value, it is returned from `unwrap_or_default()`. We described our approach to this problem in Section 3.2, and that approach indeed bore out, though there is an extra detail that we skipped earlier:

Since a type that is not `Copy` cannot implement `default()` purely[4], we would like to avoid assuming `PureDefault` for non-`Copy` types. Ideally, we would be able to spell this as a supertrait (`auto trait PureDefault: Copy`), but Rust does not support this. Instead, wherever we use the trait, we form a conjunction with `Copy`, i.e. `Copy + PureDefault`. Thanks to this setup, non-`Copy` types are effectively automatically opted out of this trait which they cannot implement.

All told, the final specification looks as follows:

```
#[ensures(old(self.is_none()) || old(self) === Some(result))]
#[refine_spec(where T: Copy + PureDefault, [
    ensures(result === match old(self) {
        Some(v) => v,
        None => Default::default(),
    })
])]
fn unwrap_or_default(self) -> T where T: Default;
```

In future, the `PureDefault` trait may be avoidable by using a call description [10] instead, which enables specifying calls to functions regardless of purity in a less direct way. Then again, depending on the syntax and performance, this may be

[4]Pure functions require their argument and return types to be `Copy`—Subsection 4.2.10 elaborates on this.

more burdensome, especially for users trying to specify their own functions calling `default()`, so it's possible this `PureDefault` approach will still be useful.

4.2.4 `From<T>` and `Into<U>`

These traits form a generalized way to describe conversions from one type to another. Users simply implement `From<T>` for a type `U`, defining a method `U::from` to construct it from a `T`. In turn, Rust defines a blanket implementation that implements `Into<U>` for the source type `T`, providing a method `T::into()` that allows users to leverage type inference to determine what to convert to (commonly used to provide arguments to a function).

In order to specify this behavior, we need to describe the call to `From::from` in the spec for `into()`, which once again means it has to be pure. To this end, we introduced an auto trait `PureFrom` imposing the additional purity constraint on `from()`, allowing us to write the following specification:

```rust
#[extern_spec]
impl<T, U> Into<U> for T where U: From<T> {
    #[refine_spec(where U: Copy + PureFrom, T: Copy, [
        pure,
        ensures(result === U::from(self))
    ])]
    fn into(self) -> U;
}
```

Note how we mimic `Copy` constraints on the types involved in `PureFrom` by expressing them at the usage site, since this is not possible at the trait definition. In this case, we actually need to constrain both `U`, the type implementing `PureFrom`, and `T`, the type from which we are converting. Type-conditional spec refinements with multiple bounds like this were not supported in Prusti; this is another feature we added that improves it for all users. Prusti also did not support extern specs on generic traits due to an overly cautious assertion which we simply removed.

There is also a blanket implementation for identity conversions, which we can specify fully even without additional infrastructure. However, this does not apply to the corresponding `Into` implementation, since it results from the general blanket implementation. In order to support this, we would need to refine the above specification conditionally when `T = U`, but this is currently not a supported constraint in Rust and thus also not by Prusti. Until that changes, the best we can do is to apply the same `PureFrom` conditional refinement as above, and limit applicability to that scenario:

```
#[extern_spec]
impl<T> From<T> for T {
    #[ensures(result === other)]
    #[refine_spec(where T: Copy, [pure])]
    fn from(other: T) -> Self;
}
```

Apart from identity conversions, another common use case of `From`/`Into` conversions is exact conversions between different numeric types. This does not apply to every pair of types, e.g. a `u32` is not guaranteed to fit into an `i32` or vice versa, floats cannot exactly represent integers of the same bit width, and there are very few guarantees about the size of `usize`/`isize`. Still, there is a good amount of conversions implemented, and we can cross-reference Rust's own implementations to make sure they are all specified. Once again, we leverage a macro to limit boilerplate and concisely list supported conversions. Our specification expresses the conversion using `as`, which Prusti supports intrinsically:

```
macro_rules! specify_numeric_from {
    ($from:ty => $($to:ty)*) => ($(
        #[extern_spec]
        impl From<$from> for $to {
            #[pure]
            #[ensures(result == num as Self)]
            fn from(num: $from) -> Self;
        }
    )*)
}

// e.g.:
specify_numeric_from!(u16 => u32 i32 u64 i64);
specify_numeric_from!(i16 => f32 f64);
```

4.2.5 `core::mem::size_of`

`size_of<T>()` is another interesting function to specify, since its return value depends entirely on the generic argument `T`. In order to specify the concrete values, we once again employ a helper trait, `KnownSize`, as outlined in Section 2.2, providing a pure method `size()` that we then refer to in a conditional refinement:

```
#[pure]
#[refine_spec(where T: KnownSize, [ensures(result == T::size())])]
fn size_of<T>() -> usize;
```

However, this glosses over some important details. Previously, one could not attach the `#[extern_spec]` attribute to free-standing functions, instead it would have to be applied to a module containing the function—with nesting to reflect the original module structure—so in this case a `mod core` nesting a `mod mem` containing the `size_of` function. This was worsened by the fact that the extern spec transformation process maintained this module structure, giving it a separate namespace from the rest of the file, as Rust modules do. To work around this, one had to re-import modules like `prusti_contracts::*` for the specification macros. In our case, to refer to the `KnownSize` trait defined in the same file, we would have had to import that into scope as well:

```
#[extern_spec]
mod core {
    mod mem {
        use prusti_contracts::*;

        #[pure]
        #[refine_spec(where T: super::super::KnownSize, [
            ensures(result == T::size()),
        ])]
        fn size_of<T>() -> usize;
    }
}
```

We made several changes to improve this situation. For one, we modified the transformation to remove the modules and only keep their contents. This required mangling the function names to avoid name collisions with the actual functions being specified. To reduce the need to nest, `extern_spec` now also optionally takes a module path as argument, e.g. `core::mem`. So the previous example could now be expressed as `#[extern_spec(core)] mod mem { ... }`.

```
#[extern_spec(core)]
mod mem {
    #[pure]
    #[refine_spec(where T: KnownSize, [ensures(result == T::size())])]
    fn size_of<T>() -> usize;
}
```

Better yet, this change paved the way for another extension to allow specifying free-standing functions without an enclosing module:

```
#[extern_spec(core::mem)]
#[pure]
#[refine_spec(where T: KnownSize, [ensures(result == T::size())])]
fn size_of<T>() -> usize;
```

All these changes reduce boilerplate needed to specify singular free-standing functions not just for us but for any Prusti user. For completeness' sake, the module path argument applies not just to modules and functions but also to traits. It does not apply to `impl`s because those already accept qualified type paths.

4.2.6 `Deref`, `Index`, etc.

These traits center around a method that returns a reference to an associated type. These methods should be marked pure (conditionally, with a marker trait), since their output is always the same as long as the receiver is not mutated. However, Prusti currently crashes when specifying these methods, presumably from the attempt to use a pure function that returns a reference. We have filed an issue for this, **issue #1221**[5], but unfortunately could not resolve it within the timeline of this thesis.

This has turned out to be a rather big issue, blocking a wide array of fundamental specifications, such as interacting with or constructing a smart pointer or indexing into collections. Resolving it will greatly open up the space for new specifications of smart pointers (though `Box` already has support built into Prusti) and collections (like `Vec` and `HashMap`).

[5] https://github.com/viperproject/prusti-dev/issues/1221

4.2.7 Vec, String

These suffer from issue #1221 described above, but we would still like to describe our planned approach to specifying them. When users interact with these types, they are usually implicitly making use of their `Deref` implementation, forwarding to `as_slice()` or `as_str()`, respectively. As such, marking these methods as pure already suffices for everything not involving mutation. They also repeat some methods from the underlying type, which can simply forward to the respective underlying method e.g. for `Vec::len`:

```
#[pure]
#[ensures(result == self.as_slice().len())]
fn len(&self) -> usize;
```

`[T]::len`, a member function on slices returning their length, is an intrinsic operation built into Prusti, so this approach can preserve the information all the way from initialization with a slice literal in the `vec!` macro to any later operations on the `Vec`.

Another valid approach would be to simply mark `Vec::len` as pure and express the connection to `as_slice` as a postcondition on the latter instead:

```
#[pure]
#[ensures(result.len() == self.len())]
fn as_slice(&self) -> &[T];
```

This same connection would also be expressed for the implementations of `Deref`/`DerefMut` and similar operations. The benefit of this approach is that it reduces indirection for the methods unique to `Vec`, shifting the complexity to its slice view accessors.

Whichever approach ends up being chosen, the other missing piece for this end-to-end support is that Prusti needs to understand the “unsizing” operation that turns a literal of fixed size (e.g. `[i32; 5]`) into a slice (e.g. `[i32]`), whose size is not known to the compiler. `String` is not quite as far along, since `str` currently only has preliminary built-in support, but future work could treat it analogously.

Mutating operations would be described using quantifiers, e.g. the specification for `Vec::push` might look as follows:

```
#[ensures(self.len() == old(self.len()) + 1)]
#[ensures(self[self.len() - 1] === value)]
#[ensures(forall(|i: usize|
    i < old(self.len()) ==> self[i] === old(self[i])
))]
fn push(&mut self, value: T);
```

This relies on the `Index` implementation being specified via `as_slice()`, with slice indexing being another built-in.

4.2.8 Ranges & Comparisons

Ranges turn out to not be used particularly much outside of iteration, but are clear candidates for specification due to their usefulness as algorithmic building blocks. One might imagine a spec on `core::ops::Range<Idx>::contains` as follows:

```
#[pure]
#[ensures(result == (self.start <= *item && *item < self.end))]
fn contains<U>(&self, item: &U) -> bool; // where clause omitted
```

Unfortunately, this does not currently work, due to its usage of `PartialOrd` comparison operators, which are only built into Prusti for integers. We would like to provide general specifications for them, requiring purity and spelling out the relationships between the various methods in detail. Notably, we want to declare them as pure unconditionally, since impure implementations would violate guarantees necessary for sorting and other uses.

However, simply marking them pure would currently result in very poor user experience. The reason is that users most commonly implement these functions through `derive` macros, rather than writing the boilerplate manually. This, however, causes a verification failure because the functions marked pure in the trait are not marked pure in the synthesized implementation:

```
// other derive macros omitted
    for brevity
#[derive(PartialOrd)]
struct IntContainer {
    x: i32,
}
```

```
#[automatically_derived]
impl PartialOrd for IntContainer {
    #[inline]
    // need #[pure] here
    fn partial_cmp(
        &self, other: &Self
    ) -> Option<Ordering> {
        PartialOrd::partial_cmp(
            &self.x, &other.x)
    }
}
```

This can be worked around by either implementing `PartialOrd` manually or using extern specs to attach the specifications, but such a workaround would be unreasonably burdensome for users, requiring exactly the kind of boilerplate that `derive` macros were designed to avoid.

On Prusti's side, the macro expansion could be intercepted somehow to attach the necessary attributes. Perhaps more generally, methods produced through derive macros could inherit not only the pre-/postconditions but also the purity. Inheritance of pre-/postconditions is already the case for trait implementations in general, so also inheriting the `pure` attribute would follow these conventions.

Without making changes to Prusti, it would also be possible to add a marker trait `PurePartialOrd` that is *not* implemented by default (unlike the others), and conditionalize the purity on that. We would then implement it for and attach the necessary specs to basic types like `i32`, requiring users to manually implement it for their own types if they want to benefit from the specifications. However, this would add a noise and complexity to the specifications, and ultimately still need to be overhauled with a more permanent solution.

The problem is the same for `PartialEq`, which we would also like to mark pure for use in specifications. We have opened an issue[6] to discuss possible approaches to this problem. Until it is resolved, we have chosen to leave these methods unspecified.

[6]https://github.com/viperproject/prusti-dev/issues/1311

4.2.9 `Option::and`

This method returns its argument if the receiver is some, otherwise returning `None`. A possible specification might look like the following:

```
#[ensures(match result === old(self) {
    Some(_) => old(other),
    None => None,
})]
fn and<U>(self, other: Option<U>) -> Option<U>;
```

However, this is not yet supported by Prusti, because algebraic data type literals[7] without arguments (like `None` here) are represented differently than those with arguments (like `Some(foo)`) in Rust's MIR intermediate representation, and the former's representation is not yet supported[8]. Instead, the `result ===` needs to be pulled into the match, resulting in the following specification:

```
#[ensures(match old(self) {
    Some(_) => result === old(other),
    None => result.is_none(),
})]
fn and<U>(self, other: Option<U>) -> Option<U>;
```

Once Prusti does support these literals, it would make sense to adjust specifications like this to leverage the feature.

4.2.10 Other Limitations

Prusti has a concept of purity, where pure functions cannot access or modify external state, even through references. In Rust's standard library, the vast majority of functions that don't take mutable references as argument are pure—this makes for handy building blocks. However, for a function to be marked pure in Prusti, its arguments and return type need to implement `Copy`. Among other things, this allows Prusti to express that their output is always the same given the same inputs.

[7]Note that this does not apply to the usage of `None` as a pattern to match against (preceding `=>`, which Prusti does support.

[8]Issue #1268: https://github.com/viperproject/prusti-dev/issues/1268

This unfortunately poses a problem for generic types like `Option`, `Result`, etc. when considering functions consuming or returning a value of that type, requiring a type-conditional spec refinement to express this purity only when the wrapped value (and thus the container) is `Copy`. It would make sense to investigate ways to relax these restrictions, perhaps adding a variant of `pure` or an argument to it (something like `pure(if_generics_valid)`) that makes it only apply when the concrete generic arguments implement the necessary traits, having no effect otherwise. We have opened a GitHub issue[9] for Prusti to explore this space.

4.3 Prusti Improvements

In the process of writing the specifications, we ran into a few shortcomings of Prusti, several of which we addressed with new features or enhanced support. Not only did these improvements make it easier for us to write the specifications we wanted, but they will also make it easier for Prusti users to write their own specifications.

4.3.1 Type-Conditional Spec Refinements

Type-conditional spec refinements gave us the power to meaningfully specify behavior and thus were central to our specifications. Initially, these were considered an unstable feature and gated behind a feature flag. Furthermore, there was also no way to ignore this attribute when the flag is not set[10], so either users would have had to set the flag to use any of our specifications, or we would have had to leave out the ones making use of these refinements for now. In the interest of advancing Prusti as a whole, we decided to finalize and stabilize them.

They were initially referred to as ghost constraints, and used a slightly different syntax. Our stabilization consists of the new naming as well as a slightly adjusted syntax to form a connection to Rust's existing `where`-constraints:

```
#[ghost_constraint(Foo: Bar, [ensures(foo::bar())])]
#[refine_spec(where Foo: Bar, [ensures(foo::bar())])]
```

[9]https://github.com/viperproject/prusti-dev/issues/1313

[10]They would have had to be ignored only in upstream crates like our specs, while still producing diagnostics when users attempt to use them in their own code without setting the flag, and even this behavior may not always be desired.

As previously mentioned, these refinements also did not support conditionalizing the `pure` attribute, which we remedied. Further, we also extended them to fully support any constraints Rust itself supports, rather than relying on a custom parser. This was a problem with constraints involving commas, e.g. `where Foo: Bar<A, B>, [...]`, as it interpreted commas within constraints as separating them. And of course, as previously mentioned, we added support for multiple constraints rather than limiting to just one.

4.3.2 Extern Specs

We ran into many shortcomings of extern specs, both in functionality and in user experience. Some of these, such as taking a module path as argument and treating modules as transparent, have already been discussed above. The user experience improvements, however, are more subtle and merit some highlighting.

Extern specs expect to be applied to method stubs, ending in a semicolon rather than a body in braces, but this was not explicitly enforced. This was potentially confusing and allowed different spellings like `{}`, so we now detect such cases and diagnose an error.

Traits can define predicates, which are functions that are transparent to Prusti, typically used as part of pre-/postconditions. These predicates did not allow defining a default body for implementations to inherit. This change opens the door to powerful trait-based specification with reduced boilerplate, like a possible future approach to `Fn` traits with e.g. a precondition and invariant defaulting to `true`.

The parsing of extern specs[11] only handled application to simple type paths. This meant that e.g. specifying the implementation of `[i32]` involved first defining a `type Slice_i32 = [i32]` or similar to work around this limitation. This workaround was not obvious and the limitation added friction to the user experience. We have extended it to allow any valid Rust type spelling, simply delegating to the built-in parser and refactoring some of the logic that relied on types being simple paths.

The desugaring of extern specs created new functions calling out to the original function, along with a defensive assertion that the code was unreachable, as shown below. The call itself lets Prusti identify from the desugared form which function is being specified.

[11]And `#[refine_trait_spec]`, a related attribute necessary to apply specifications to a trait implementation.

```
// attributes omitted
fn prusti_extern_spec_size_of<T>() -> usize {
    core::mem::size_of();
    unreachable!()
}
```

However, this assertion made it so that the function call was not type checked against the extern spec's declared context. This allowed e.g. incorrectly declared return types to slip through, creating confusing mismatches between the specification and the original function. We deemed the assertion unnecessary and removed it to restore this type-checker connection.

The desugaring also did not forward generic arguments (as shown in the snippet above), instead relying on the concrete argument values to suffice for their inference. This could have caused all manner of issues, but most pressingly for us, it made it impossible to specify `size_of`, since the only argument it takes is a generic parameter. We adjusted the generated code to explicitly forward generics, in the process uncovering several instances in existing specs where the declared generics did not match the original ones.

Writing an extern spec for a trait used to require annotating each type parameter with `#[generic]` or `#[concrete]`, with the envisioned design that the behavior for a specific set of type arguments could be specified separately from the rest. Type-conditional spec refinements are a more general and complete approach that also solves this problem. Additionally, concrete generics for traits were not functional, so we were not touching functionality that users could have been relying on. In the process of stabilizing them, we obsoleted these attributes, resulting in less noisy trait declarations.

Chapter 5

Evaluation

5.1 Specification Progress

Using Qrates and the techniques outlined previously, we gathered a dataset of 87,060,395 calls across 73,181 crates[1], with 18,885 of those defining functions called by other crates. We then manually categorized groups of these calls by their specification status, determining for each combination of call target and receiver whether we had specified it and, if not, the rationale (missing features/not useful to specify/etc.). With all that information, we were able to chart the distribution of these categories (Figure 5.1).

The largest chunk of calls is marked as `macro`. This status conveys that the call has not been specified, but it is likely not as important to specify, as users did not perform this call directly. Macros often expand into a large amount of calls, especially `quote::quote!` (iterating over tokens) or the various derive macros (iterating over fields) As such, the macros have been separated out to avoid skewing the remaining data towards methods people are unlikely to interact with directly.

The next big chunk is `unknown`, which are calls which we did not yet triage to determine their status, most likely currently unspecified. Next is `complete`, for calls that we have specified as much as we deemed to be useful. A little later but topical here, another large chunk is marked `generic`. These are calls that were not resolved to concrete types, either being generic parameters or `impl` existentials. This categorization technically overlaps with other statuses, and in our evaluation

[1]Out of the 100,156 crates on `crates.io` (at the time of writing), these were the ones that we could compile successfully.

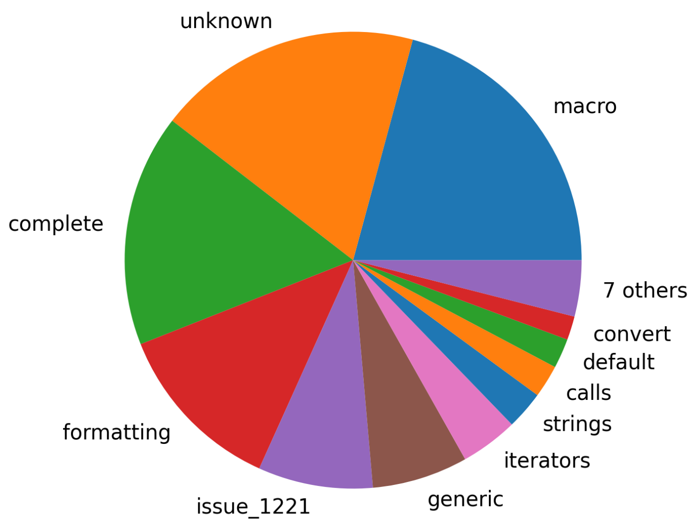

Figure 5.1: The frequency of each specification status, i.e. how many of the 87 million calls fall into each category.

we only categorize calls as this if they would otherwise have been deemed unknown, so e.g. unresolved generics use in a macro would simply be categorized as `macro`.

The other categories concern themselves with semantic grouping:

- `formatting` comprises the calls involved in formatting strings, e.g. due to the `format!()` macro or `println!()` etc. These calls almost always originate from macros, which is why the slice is so large—otherwise those calls would largely be part of the `macro` slice. They are shown here because we noticed that macro involvement is not detected reliably for these methods, so we instead decided to categorize them directly.

- `issue_1221` represents specifications blocked by issue #1221, most notably `Deref` and `Index`, as laid out in Subsection 4.2.6. The size of this slice illustrates the importance of resolving that issue.

- `iterators` consists of the various methods defined on the `Iterator` trait, as well as the ways to construct iterators from collections and vice versa (`into_iter`, `collect`, `[T]::iter`, etc.). There has been prior work to establish an approach for these [12], but it has yet to be added to Prusti's own specification library.

- `strings` contains methods like string equality and mutation, which are likely to be handled holistically by a future effort to improve Prusti's support for strings.
- `calls` is for methods that require call descriptions [10] to specify meaningfully, e.g. `Option::unwrap_or_else`, which currently has a basic specification, but whose usage of the passed-in function cannot yet be described.
- `convert` groups `From::from()` and `Into::into()`, which we will look at in more detail in Subsection 5.1.2, since the group consists of a variety of different conversions, not all of which are specified.
- `async` describes Rust's futures and `async`/`await` system. Specifying this area is likely to be the subject of future research.
- `interior_mutability` groups methods that will only be specifiable once Prusti gains support for the notion of interior mutability. It comprises smart pointer types like `Arc` and `Pin`.

Aside from the following subsection, the rest of this section goes into more detail about some targets that cannot be adequately categorized without considering generic parameters and more.

5.1.1 Macros

There were 41,554,923 calls written through macro expansion, 25,980,458 of which are not currently specified or more specifically categorized. Out of the former, we further investigated which specific macros were most popular—Figure 5.2 shows these results.

The `serde` crate accounts for the biggest macro in terms of calls generated. It provides facilities for serializing to and deserializing from data formats, notably JSON. Fully specifying this crate would be difficult, requiring a powerful model of the data format that is being read from or written to in order to model end-to-end properties.

Other big macros include derivations of the `Debug`, `Clone`, and `PartialEq` traits. `Debug` is unlikely to be relevant for the functional correctness of any program, as it simply represents a type for printing to understand a program's behavior. The `Clone` expansion is correctly verified against the snapshot equality postcondition (unless opted out of the `SnapshotEqualClone` trait). The `PartialEq` trait is not yet specified, for reasons outlined in Subsection 4.2.8.

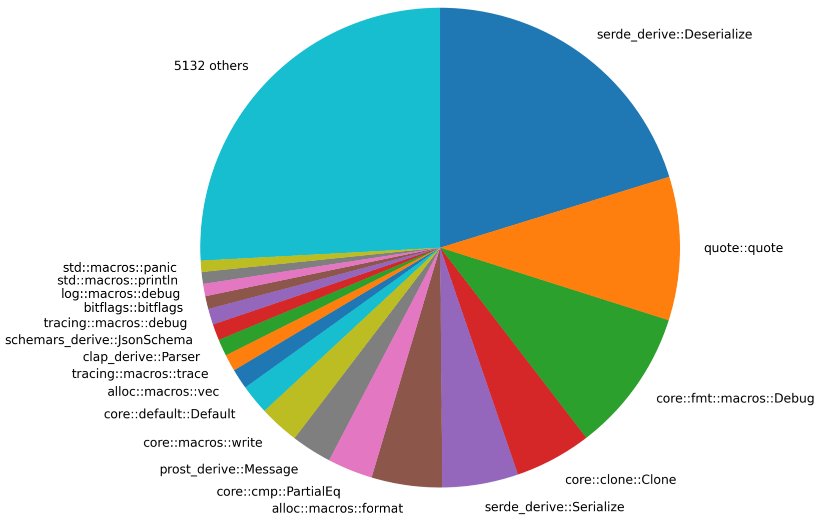

Figure 5.2: The frequency of each macro-induced call, i.e. how many calls resulted from an expansion of each. Macros whose names match traits are `derive` macros for said trait.

5.1.2 Conversions

The `From` and `Into` traits together account for around 2% (1.77 / 87.06 million) of all calls in the dataset. Ignoring calls whose generics were not resolved to concrete types leaves 1.46 million, and further ignoring calls originating from macros trims it down to 1.11 million. The rest of this section will work with this last subset, as it is most directly relevant to users.

Apart from identity conversions, with the same type as source and target, these calls need to be specified individually for each pair of involved types. At this point, we have only specified all the numeric conversions, as outlined in Subsection 4.2.4. The most popular conversion is `str` to `String`, alone accounting for 13.3% of specifiable calls, but this is blocked due to issue #1221. Taking all this into account, our progress is shown in Figure 5.3.

The results are sobering: Even with the issue resolved, we would have less than a quarter specified. The unfortunate truth of conversions is that they are more spread out and thus would need proportionally more specifications to reach the same fraction of calls specified than for the dataset as a whole (recall the analogous area graph in Figure 3.1). On closer investigation, only for 28.3% of

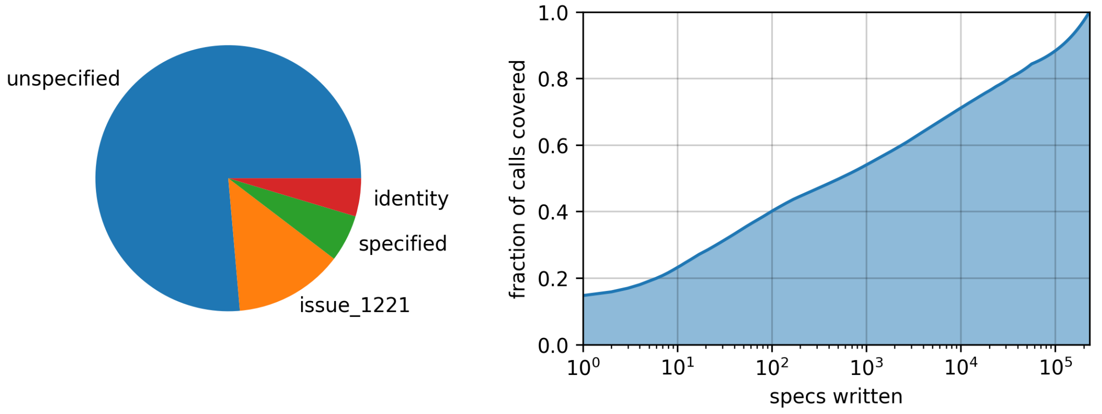

Figure 5.3: L: Distribution of specification status for calls to `from` and `into`.
R: What fraction (y) of these calls would be covered by specifying the top x.

conversions are both their source and target type part of the standard library, with the rest involving third-party crates. With this in mind, our specifications (would) achieve a much higher level of coverage, though it remains an important insight that third-party crates are so often involved in these conversions.

5.1.3 `Default`

Calls to the `Default` trait's only method, `default`, also account for around 2% (1.90 / 87.06 million) of all calls, just slightly more than conversions. Of those, 1.82 million were resolved to concrete types and thus specifiable, with our current specifications covering 1.11 million or 60.8% of them. Considering only calls involving types from the standard library brings the total down to 1.19 million, with our 1.11 million equating to 92.9% coverage.

5.1.4 `size_of`

There are 100,635 calls to `core::mem::size_of`, a much smaller part than the previous methods. Out of these, 89,218 were resolved to concrete types, of which 66,601 or 74.8% are currently covered. Our specifications cover primitive numeric types and `bool`, as well as arrays thereof, but this clearly makes up the vast majority of calls to this method. We were slightly surprised by this, as in the original dataset of only the top 1000 crates, these types accounted for almost 90% of the `size_of` calls, but the end result is still reasonably close.

Interestingly, the closely-related method `core::mem::align_of` is not only used much less—only 8301 times (6510 specifiable)—but also much less with these primitive types. Instead, the vast majority of its queries involve third-party types from crates like `i-slint-core` and `sixty-fps-corelib` (a previous branding of the former).

Despite the comparatively low usage, these functions would make sense to support directly in Prusti, achieving full coverage even on third-party types.

Chapter 6

Conclusion

In this work, we have written specifications for a variety of types that users commonly interact with. We have also categorized a majority of the as-yet unspecified methods, identifying blocking issues or groups that should be approached holistically. Further, our pull requests and accessible data[1] make it easy to inform future specification efforts with frequency data from a large dataset. Though there are several major issues blocking specification of some methods or requiring awkward rewrites of existing code to integrate Prusti, these specifications already reduce the necessary setup to start verifying real-world code. This is illustrated by the fact that several unit tests for Prusti were adjusted to remove ad hoc specifications that Prusti now ships with.

Even in cases where we do not yet offer specifications, it is now easier for a user to add their own extern specs to a method they need. This is thanks to the various improvements we have made to extern specs and type-conditional spec refinements (Appendix A). The infrastructure we now have in place and code patterns we have established further open up the space for users to contribute their own specifications to Prusti's collection via Pull Requests.

[1]https://github.com/juliand665/qrates-eval

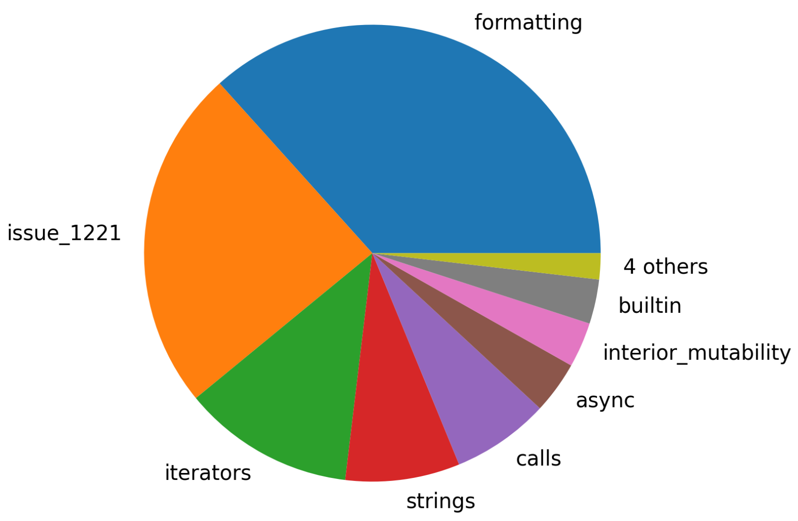

Figure 6.1: The frequency of each specification category identified for future work.

6.1 Future Work

6.1.1 Specifications

In our evaluation, we categorized the method calls in our dataset into various groups, previously described in Section 5.1. Figure 6.1 provides an overview of what coverage can be expected from various future efforts, by starting with Figure 5.1 and limiting it to the categories most relevant for guiding future work (excluding `unknown`, `complete`, `macro`, `generic`, `default`, and `convert`). For reference, this subset of categories adds up to 29.13 million calls, or 33.5% of the previous pie chart (i.e. all calls).

The biggest slice is `formatting`, but this is likely not to be as immediately relevant to Prusti's roadmap, instead focusing on the correctness of implementations of algorithms and data structures. As such, the main standout is issue #1221, unblocking a range of specifications for fundamental types like `Deref` and `Index`, the latter being especially relevant for data structures. The other issues also serve as starting points, notably issue #1311[2], which blocks usage of `PartialEq` and `PartialOrd` for arbitrary types in pure code (incl. pre-/postconditions), blocking specifications of e.g. the relationships between the `PartialOrd` methods or of the range types (important for loops and functional chains). All reported issues are listed in Appendix A.

Finally, as outlined in Subsection 5.1.4, Prusti should gain built-in support for `size_of` and `align_of`, rather than the current state of simply specifying the results for some widely-used types.

6.1.2 Qrates

Especially as more method groups get unblocked, the fact that the concrete types of generic parameters is not known for every call will increasingly limit insights, possibly also biasing results towards certain use cases. To remedy this, future work on Qrates could work analogously to monomorphization[3] (or even make use of it), starting from places where all concrete types are known and recursively descending into calls made from there, maintaining knowledge of the concrete generic substitutions to fully resolve every call.

[2] https://github.com/viperproject/prusti-dev/issues/1311

[3] Monomorphization is the process by which Rust inlines generic parameters at compile time to avoid incurring any overhead over specialized methods: https://rustc-dev-guide.rust-lang.org/backend/monomorph.html

Bibliography

[1] Nicholas D. Matsakis and Felix S. Klock. The rust language. Ada Lett., 34(3):103–104, oct 2014.

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging rust types for modular specification and verification. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), volume 3, pages 147:1–147:30. ACM, 2019.

[3] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, Verification, Model Checking, and Abstract Interpretation (VMCAI), volume 9583 of LNCS, pages 41–62. Springer-Verlag, 2016.

[4] J. Hansen. Specification and Verification of Iterators in a Rust Verifier. Master's thesis, ETH Zürich, 2022.

[5] J. Mičko and J. Fiala. Support for library contracts. https://github.com/viperproject/prusti-dev/pull/1086. Accessed 2023-01-16.

[6] Rust contributors. A `no_std` Rust Environment. https://docs.rust-embedded.org/book/intro/no-std.html. Accessed 2023-01-23.

[7] Prusti developers. The qrates framework. https://github.com/rust-corpus/qrates. Accessed 2022-07-11.

[8] Typeshed contributors. The typeshed repository. https://github.com/python/typeshed. Accessed 2022-07-18.

[9] Definitely Typed contributors. Definitely Typed: TypeScript type definitions. https://github.com/DefinitelyTyped/DefinitelyTyped. Accessed 2022-07-18.

[10] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. Modular Specification and Verification of Closures in Rust. Proc. ACM Program. Lang., 5(OOPSLA), 2021.

[11] Prusti developers. Snapshot-based type encoding. https://viperproject.github.io/prusti-dev/dev-guide/encoding/types-snap.html. Accessed 2023-01-28.

[12] Aurel Bílý, Jonas Hansen, Peter Müller, and Alexander J. Summers. Compositional Reasoning for Side-effectful Iterators and Iterator Adapters, 2022.

Appendix A

Open-Source Contributions

As part of this thesis, we created multiple Pull Requests on GitHub to merge our work into both Prusti and Qrates. We also opened several issues to describe and gather information on problems we ran into but could not solve as part of the thesis. For reference, these are outlined below, along with a quick summary of their main purpose.

Separately, we published the scripts written to orchestrate gathering our large dataset, as well as the Jupyter notebook in which we analyzed it, alongside the raw data, as a new GitHub repository: https://github.com/juliand665/qrates-eval

Prusti Pull Requests

#1138: Ghost Constraint Improvements

Fixes parsing issues with ghost constraints/type-conditional spec refinements and lets them specify the purity of a function.
https://github.com/viperproject/prusti-dev/pull/1138

#1188: Extern Spec Improvements

Makes extern specs applicable to more types and functions.
https://github.com/viperproject/prusti-dev/pull/1188

#1249: Provide Specs for the Standard Library

Contributes the specs we wrote to Prusti itself, making them available to users.
https://github.com/viperproject/prusti-dev/pull/1249

#1271: Ghost Constraint Stabilization

Finalizes the previously-unstable ghost constraints as type-conditional spec refinements, reworking the syntax and removing the flag gating them.
https://github.com/viperproject/prusti-dev/pull/1271

Qrates Pull Requests

#315: Add new query for resolved calls (with receivers & generics)

Introduces a new query to extract the data we needed to prioritize specifications and evaluate their coverage.
https://github.com/rust-corpus/qrates/pull/315

#353: Batched Compilation

Enables splitting the compilation into batches, for resumption after failures and avoid hitting memory limits for very large processes.
https://github.com/rust-corpus/qrates/pull/353

Prusti Issues

#1179: Type aliases treated opaquely when constructing ADTs

https://github.com/viperproject/prusti-dev/issues/1179

#1221: Error when using pure functions that return a reference

https://github.com/viperproject/prusti-dev/issues/1221

#1310: Missing Source Snippets in some circumstances

https://github.com/viperproject/prusti-dev/issues/1310

#1311: Inheritance of Purity from Trait Specs

https://github.com/viperproject/prusti-dev/issues/1311

#1313: Supertrait limitations of auto traits

https://github.com/viperproject/prusti-dev/issues/1313

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Annotating the Rust Standard Library with Specifications for Use in a Rust Verifier

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Dunskus	Julian

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 2023-01-25	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.