Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Reasoning about Complexities in a Rust Verifier

Master Thesis

Lowis Engel

October 9, 2021

Advisors: Prof. Dr. Peter Müller, Dr. Christoph Matheja and Aurel Bílý

Department of Computer Science, ETH Zürich

Abstract

In addition to safety and correctness properties, the efficiency of programs is gaining more attention in recent years. There is a demand to reason about the resource usage of software in terms of runtime, memory space, energy consumption and other resources.

In this thesis we present a method to prove polynomial resource bounds in a procedure-modular verifier. We focus on verifying asymptotic bounds, but the approach is also suited to reason in terms of concrete quantities. The polynomial bounds can be defined in terms of multiple variables.

Similar to heap access permissions in separation logic or implicit dynamic frames, we use so-called resource credits to represent the right to consume some unit of a resource quantity. The reasoning about such credits is generic, the semantics of a resource are defined separately by a cost model. Polynomial amounts of such credits are represented by their coefficients in a normal form to simplify the proof obligations.

The verification is implemented as an extension to the Prusti verifier which encodes Rust programs for the Viper verification infrastructure.

Acknowledgments

I would like to thank my supervisors Dr. Christoph Matheja and Aurel Bílý for their support and patience. Our weekly meetings kept me motivated, provided helpful feedback and always left me with a positive feeling. This motivation was especially important due to the isolation in the pandemic. I am also grateful that they came up with this interesting research topic.

I would like to extend my thanks to Dr. Alexander Summers for providing us with insights about Viper and reminding us about permission amounts above one, which was crucial for our Viper encoding. He also gave a great introduction to program verification in a lecture with the same name, that amplified my interest in the topic and equipped me with the background knowledge for this thesis.

Furthermore, I thank Federico Poli for introducing me to the Prusti code base and giving me the opportunity to contribute to the project as a preparation for working on this thesis.

I am also grateful to Prof. Peter Müller for the opportunity to work on both of my theses with the great people of his research group. His lectures sparked my excitement for programming languages and software quality, which lead to the choice of my research topics.

Finally, I would like to thank my family for their support and especially, for helping me to stay determined and motivated towards the end of this project.

Contents

Chapter 1

Introduction

Computers and software running on them have become omnipresent in our lives. The more we depend on them, the more important it is that they work reliably. At the same time, the size and complexity of software systems are increasing, which makes ensuring their reliability even harder. Different measures have been applied in the past to improve reliability.

Programming languages can prevent certain errors by imposing restrictions on programmers. For example, the Rust programming language [21] has a strict type system that provides strong safety guarantees. The so-called ownership type system statically prevents common memory bugs such as data races, dangling pointers and undesirable aliasing side effects.

Nevertheless, programming languages cannot prevent programmers from writing functionally incorrect code, i.e. code that does not compute the desired results. Empirical methods like testing can be used to increase the confidence in a program's correctness. However, they can often only cover a subset of all possible inputs and program states. Higher reliability can be achieved by using a static verifier, that is able to statically reason about all possible program states.

The Prusti project built one such verifier for Rust [2]. As usual in program verification, the Prusti verifier is mainly aiming at verifying functional requirements. Additionally, it is able to prove the absence of errors like overflows or panics (unrecoverable failures). However, functional correctness and safety guarantees do not imply efficient implementations.

Especially for a programming language like Rust, 'targeting systems-level applications' [21], verifying assumptions about the resource usage of a program to ensure efficiency might be equally important. For example, bounds on the stack space usage of programs provide safety guarantees for potentially safety-critical applications in embedded systems [23]. Furthermore, reasoning about energy consumption becomes increasingly important with

the rise of handheld devices and cloud computing [10]. Finally, proving constant runtime bounds on cryptographic algorithms helps to prevent timing side-channel attacks [20, 5].

The goal of this thesis is to extend the Prusti verifier to be able to prove asymptotic upper bounds on the resource usage of Rust programs. As usual in Prusti, the proof is performed by an encoding in the intermediate language of the Viper verification infrastructure [22]. Combining this encoding with the existing encoding of functional properties enables us to exploit functional properties for more precise resource bounds. Our resource model can also be used for amortized analyses.

1.1 Contributions

This thesis contains the following main contributions:

- The definition of a coefficient-based representation of multivariate polynomials that simplifies the proof obligations,
- Methods to propagate this representation through the program,
- A formalization of asymptotic bounds in this representation and a way how to prove them including a cost inference,
- An encoding of this approach in the intermediate language of the Viper verification infrastructure,
- A prototype implementation as an extension of the Prusti verifier.

1.2 Outline

The following chapter presents necessary background knowledge. This includes relevant methodology and related work as well as an introduction to Viper and Prusti. Chapter 3 introduces the basic principles of our reasoning about resource usage in mathematical terms. In Chapter 4 we continue this part on abstract methodology by describing how we verify asymptotic bounds. Chapter 5 presents how we encode the concepts introduced before into Viper and other implementation details. It follows an evaluation on examples in Chapter 6. Finally, we conclude in Chapter 7.

Chapter 2

Background

This chapter introduces some background knowledge that helps understanding the following chapters. It is subdivided into a description of related work and the concepts occurring there, in Section 2.1, and an introduction of the languages and tools that constitute the environment of our implementation, in Section 2.2.

2.1 Methodology and Related Work

In this section we will first introduce two ways to model amortized complexity analysis, since many formalizations for reasoning about complexities are based on these models. A short overview over such formalizations follows. Finally, we will present some existing tools that reason about program complexities.

2.1.1 Amortized Complexity Analysis

As introduced by Tarjan [26], amortized analysis is a technique to analyze the performance of operations on a data structure. The resource usage of such operations may vary a lot depending on the previously executed operations — especially when the data structure is self-adjusting from time to time to keep an efficient structure, like splay trees are. Therefore, overapproximating a sequence of operations by the sum of their worst-case complexities might lead to a very pessimistic view of the overall performance. Instead, amortized analysis takes into account which operations can or must have been executed before. Hence, it gives a more realistic view of the operation's performance in a sequence of operations.

Tarjan describes two equivalent conceptual models of amortized analysis. We will mostly rely on the so-called banker's view, also known as accounting method. It uses credits which are required to consume a certain constant

amount of a resource, e.g. to run for 1 time step. The resource usage of each operation can then be translated into paying a certain amount of credits. For amortized analysis a different number of credits can be defined to be the cost of an operation which can be smaller or greater than the actual cost. If there are enough of those credits left over in every allowed sequence of operations to pay the gap for operations with a higher actual cost (i.e. the credit amount stays non-negative), the total actual cost of any such sequence will be upper bounded by the sum of the defined credit costs. In other words the operations will be storing or receiving credits from an at least conceptual storage of credits inside the data structure to average out costlier operations.

Figure 2.1 presents a typical example for the application of this approach. It shows how the state of a 5-bit binary counter evolves. The representation has the least significant bit at the bottom. Two separate incrementation steps are presented. Following the usual definition of binary addition, the incrementation flips each bit starting from the bottom until a zero is encountered. Therefore, the run time of each addition depends on the number of 1-bits at the tail of the counter. This number ultimately depends on how many incrementations happened before.

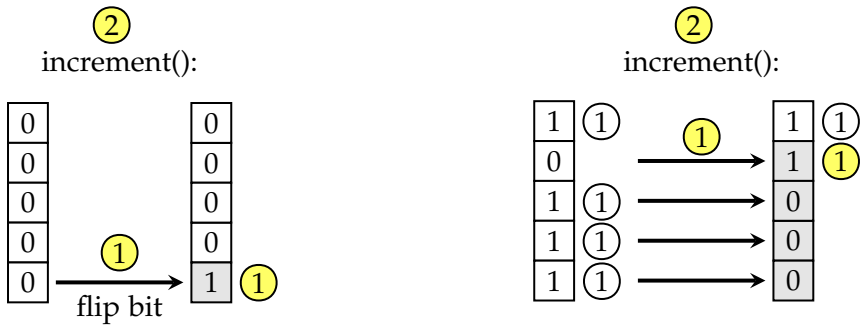

Figure 2.1: Amortized analysis of a binary counter increment using the banker's approach

We can model each bit flip to take one time step. This means, we need to pay one time credit in the banker's view. The first increment operation shown only needs one flip and hence has a actual cost of one time credit. To average out the costlier operations, we also store one time credit for the bit flip from zero to one. This stored credit will later pay for the flip back to zero. In total we have a conceptual cost of two time credits as depicted above the 'increment()'. After some increment operations we reach the state shown on the left of the second incrementation. Each bit value one has a credit stored with it. As mentioned, these can then be used to pay for the lower bit flips to zero. Therefore, we only need to pay for the last bit flip and the credit stored. Consequently, we also have a conceptual cost of two time credits for this operation, even though it would actually take four time

steps.

In some cases it might be easier to reason about amortized costs in terms of the completely equivalent physicist's or potential method. This means defining a potential function mapping the state of a data structure to a real number. The conceptual cost of an operation is then the actual cost plus the corresponding change of potential. Consequently, the total cost of a sequence of operations is the sum of their conceptual costs minus the overall potential change. If the potential was zero in the beginning and always non-negative, just the sum of conceptual costs is again an upper bound on the total actual cost.

The binary counter example can also be modeled with this approach, as shown in Figure 2.2. We define the potential function Φ to be the number of ones in the current state of the counter. This exactly matches the number of credits stored in the banker's approach. The conceptual cost of the increment operations is then defined by the sum of the number of bit flips and the change in potential. This means the cost of the first operations is:

$$1 + (\Phi(\sigma') - \Phi(\sigma)) = 1 + (1 - 0) = 2$$

And the second operation costs:

$$4 + (\Phi(\sigma''') - \Phi(\sigma'')) = 4 + (2 - 4) = 2$$

As expected, the resulting conceptual costs are the same as with the banker's approach.

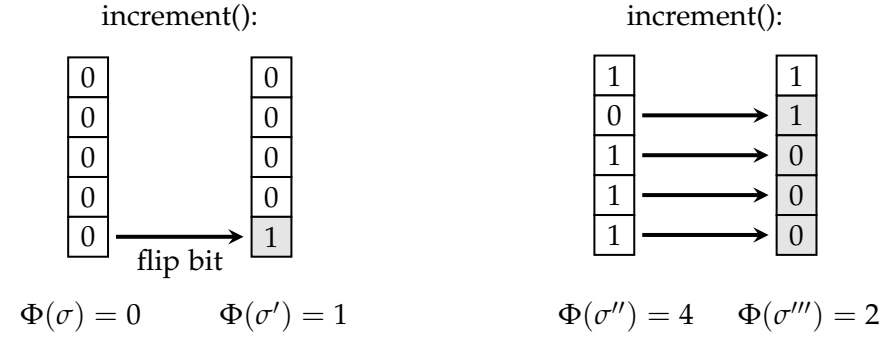

Figure 2.2: Amortized analysis of a binary counter increment using the potential method

2.1.2 Hoare Logics

In formal reasoning about the functional correctness of programs one often uses Hoare logic [15] or extensions of it. The main component of this logic

are so-called Hoare triples. They can be used to define pre- and postconditions of a program as follows:

$$\{P\} \texttt{ program } \{Q\} \tag{2.1}$$

Here, P and Q are logical predicates, mapping the program state returning to a boolean value. If this triple is proven correct and the precondition P holds in the program state before the execution, the postcondition Q will be satisfied by the result state, if the program terminates (partial correctness). Using this notation, inference rules for each program statement can be defined to enable a formal proof.

Haslbeck and Nipkow [14] identified three main formalizations to reason about resource consumption in Hoare logic, in particular about runtime. Out of these, only the following two formalizations are relevant for us:

- Carbonneaux et al. [6] use a quantitative logic based on the potential method for amortized analysis. They replace the logical predicates in Hoare triples by potentials mapping a program state to a natural number or infinity instead of boolean values. Predicates can be lifted to a potential by replacing false by infinity and true by zero. Such Hoare triples will be defined to be correct if both potentials are not infinity and the pre-potential is greater or equal to the sum of the program cost and the resulting potential.
- Atkey [3] extended separation logic with so-called time credits similar to the banker's approach in amortized analysis. This means he uses natural numbers as a way to represent consumable resources, which can be separated by addition similar to the partitioning of heap space in normal separation logic. Therefore, the credits can also be stored in data structures for amortized analysis.

The reasoning in this thesis will stay rather informal, but we will use the following notation inspired by Atkey's approach in semi-formal contexts:

$$\begin{gathered}\{P \star (P_0 \to \$_{R_0} c_0) \star \cdots \star (P_n \to \$_{R_n} c_n)\} \\ \texttt{program} \\ \{Q \star (Q_0 \to \$_{R'_0} c'_0) \star \cdots \star (Q_k \to \$_{R'_k} c'_k)\}\end{gathered} \tag{2.2}$$

The predicate P is required to hold in the pre-state, as before. Additionally, for each conditional resource credit specification $P_i \to \$_{R_i} c_i$ where P_i is satisfied, we require at least c_i credits for resource R_i to be available. Since we are using separating conjunctions ($\star$), the credit requirements add up if multiple conditions for the same resource are true. If all preconditions hold and the Hoare triple itself is valid, the predicate Q and all conditional resource credit specifications $Q_i \to \$_{R'_i} c'_i$ will be satisfied in the final state, accordingly.

Like for normal Hoare triples, there is an implicit universal quantification in this definition. It universally quantifies over the possible pre-states, from which the ones where the preconditions hold are selected. Also note, that the credit amounts are evaluated in the respective states. Therefore, they can be seen as functions mapping program states to natural numbers (including zero). This is similar to the potentials in the Carbonneaux logic, but we do not embed logical predicates as potentials here.

As before, we require the program to terminate for the postconditions to hold (partial correctness). But if the resource R_i is running time[1], we implicitly also perform a proof of termination.

Usually, the functional specification with P and Q will be omitted in this thesis. To clearly mark this abstraction, we will use $\langle\cdot\rangle$ brackets in these cases instead of the usual $\{\cdot\}$. The inference will follow similar rules as in the above logics, but the rules are often left implicit.

2.1.3 Asymptotic Bounds

The amortized analysis approaches presented in the Section 2.1.1 and the logics mentioned Section 2.1.2 are all reasoning in terms of concrete units of resources, such as time steps in some cost model. When we are not interested in tight implementation bounds and would rather like to compare algorithmic choices, abstracting cost model and implementation details by using asymptotic bounds might be useful.

We will use the well-known Landau-notation for asymptotic bounds. Unfortunately, there is only a widely accepted definition of asymptotic dominance for functions with one argument, but the resource usage of algorithms often depends on multiple variables. Therefore, we give the following explicit definition of asymptotic bounds on multivariate functions that will be used in this thesis:

$$\begin{aligned} f \in O(g) \;:\Leftrightarrow\; & \exists c > 0, N.\ \forall x_0, x_1, \ldots, x_n \geq N. \\ & |f(x_0, x_1, \ldots, x_n)| \leq c \cdot |g(x_0, x_1, \cdots, x_n)| \end{aligned} \tag{2.3}$$

This definition is inspired by the definition in Section 2.2 of [29]. A more general and formal treatment of asymptotic bounds can be found in [11] and chapter 5 of [13].

2.1.4 Existing Tools

Statically inferring the running time of arbitrary programs is an undecidable problem, since it would solve the halting problem. Therefore, tools for

[1] and the program consumes time credits at every program point that is repeated (i.e. in every loop iteration or recursive function call) as it should

runtime analysis experience the usual trade-off between automation and expressiveness. There are completely automatic analyses, but they can only support a rather small subset of programs. On the other hand, there are tools that support a large range of programs, but require user interaction for certain more complicated parts of the proof. The latter are mainly based on interactive theorem provers. Our approach, based on program verification, is more on the automation side, but requires users to specify the (asymptotic) bound they wish to verify and invariants on loops.

Automatic Analyses

A promising branch of automatic inference tools is evolving under the term 'Automatic Amortized Resource Analysis' (AARA) [17]. They statically derive concrete resource bounds using the potentials approach described above. These potentials are represented as a linear combination of some fixed base functions with abstract coefficients. The analysis collects linear constraints on the coefficients which are then solved by a linear program (LP) solver to retrieve a concrete cost. Keeping the constraints linear is only possible for a certain restricted set of programs.

For functional programs this approach is implemented in the tool 'Resource Aware ML' (RAML) [19]. There the potentials are formalized in the type system of an ML-like language [16, 18]. The tool currently derives upper and lower polynomial bounds on the number of user-defined ticks, allocated heap cells or evaluations steps in big-step operational semantics. Internally, the polynomial bounds are represented with binomial base functions. This enables a simple inference rule for pattern matches on inductive data types (like lists), frequently occurring in functional programming. Therefore, recursion guarded by pattern matching is supported very well, but boolean guards are not. More recent publications presented several extensions including exponential and probabilistic bounds, as can be seen on the project's website[2].

Carbonneaux et al. applied this technique to imperative programs in tools called C^{4}B [8] and Pastis [7]. They exclude non-linear constraints by setting the corresponding coefficients to zero, which means the conditions cannot be satisfied for certain programs. An interesting feature is the use of so-called 'weakening hints' given by the user to trigger an automatically selected rewrite function on the potential.

There are also other automatic approaches, especially for the problem of 'Worst-Case Execution Time' (WCET) [28]. WCET aims at hard real-time systems. Their approaches are not of particular interest here, since the programs analyzed usually terminate in linear time without any recursion.

[2] https://www.raml.co/publications/

Other tools using abstract interpretation (e.g. [24]) or term rewriting (e.g. [4]) usually lack compositionality, which we would like to have in our verifier.

Interactive Tools

The semi-automatic tools we looked at are mostly based on separation logic with time credits. Our original inspiration stems from Guéneau's PhD thesis [13] which extends previous work by Charguéraud and Pottier [9]. He puts particular emphasis on formalizing and encoding asymptotic bounds, arguing that they improve modularity and simplify user annotations. Moreover, working with asymptotic bounds makes the verification more robust against the concrete choice of a cost model and therefore more independent from platform details. This enables Guéneau to use a very simple, but elegant, cost model where only entering a loop or function body costs one time credit.

The tool is implemented as a Coq library. Because he extends the separation logic implementation, it is possible to reason about functional correctness properties at the same time and use their invariants in the resource proof. The implementation is able to automatically infer a concrete cost function with abstract placeholders for asymptotic costs. This concrete cost closely follows the program's structure and therefore, the simplification to an asymptotic bound often needs human insight. Especially recursive functions need to be solved by the user, using for example the master theorem [12].

Zhan and Haslbeck follow a very similar approach in their implementation for the Isabelle theorem prover [29]. They put emphasis on proving the asymptotic bounds separately in the end after inferring a concrete cost function, but there is no large difference in comparison to Guéneau's approach. A more significant improvement is the application of the Akra-Bazzi theorem [1, 12] to automatically extract asymptotic bounds from recurrences of divide-and-conquer algorithms. Solving recurrences by pattern-matching has also been applied in Timed ML (TiML) for a functional language [27].

2.2 Languages and Tools

This section introduces the Viper verification infrastructure and the corresponding intermediate language. Viper is used as a backend in the Prusti verifier for Rust that is described afterwards. In this thesis we will extend Prusti by encoding the complexity reasoning into Viper's intermediate language.

2.2.1 Viper

Viper [22] is a verification infrastructure providing an intermediate language, Silver, and automatic tools to perform formal verification on code in that language — for simplicity, we will refer to both as Viper in this thesis. Conceptually, formal verification means proving Hoare triples as described in Section 2.1.2 using formal methods. For this proof in Viper, Silver code is ultimately translated to proof obligations that are solved by the SMT (satisfiability modulo theories) solver Z3.

Viper is designed to be used as a backend for a family of verification tools targeting different programming languages. The intermediate language enables a relatively high-level of encoding (and hence less encoding effort) by natively supporting many concepts from modern programming languages, such as custom data types. In particular, Viper supports reasoning about heap manipulating programs and certain notions of concurrency. Framing global heap locations is supported by using access permissions based on implicit dynamic frames [25].

We will describe all Viper features that are relevant for this thesis in the next paragraphs. For a more comprehensive description see the aforementioned papers or the Viper tutorial[3]. The examples in this section are partially adopted from this tutorial.

Heap Access Permissions

In Viper, heap locations are represented by field accesses on references, such as `x.val` in the example in Listing 1. We can access such a field only if the program state at that point contains the appropriate permission. Field permissions in Viper are a combination of the field access and a fractional value between zero and one, both inclusive. Since the total permission amount for each heap location cannot exceed one, holding a full permission with that amount means exclusive access and we are allowed to write to the location (since there are no aliases to be updated). Amounts below one and above zero allow arbitrarily many shared read accesses.

The permission state can be explicitly manipulated by performing inhale or exhale operations. Informally speaking, `inhale A` will *add* the permissions contained in `A` to the program state and `assume` any value constraints in `A`. The counterpart `exhale A` will `assert` that all value constraints hold, then check that all access permissions in `A` are present in the current program state and *remove* them.

The example in Listing 1 shows how to use these features in practice. First, we inhale a read permission. Notice that permissions in assertions are de-

[3] https://viper.ethz.ch/tutorial/

noted by an accessibility predicate `acc` with the heap location and the permission amount as arguments. We are then able to read `x.val`. After inhaling a full permission for `y.val`, Viper can infer that `x` and `y` cannot be aliases, since the total permission amount to their val field would be greater than one. The permission amount `1/1` also allows us to write the value `3` to `y.val` in line 5. As we exhale all permissions to `y.val` afterwards, Viper's program state will lose all information about this heap location. Consequently, even after inhaling permissions to read, the assertion on `y.val` in line 8 will fail.

```
1 inhale acc(x.val, 1/2)
2 value := x.val
3 inhale acc(y.val, 1/1)
4 assert x != y
5 y.val := 3
6 exhale acc(y.val, 1/1)  // havoc
7 inhale acc(y.val, 1/2)
8 assert y.val == 3       // will fail
```

Listing 1: Example for usage of heap access permissions

Predicates

To be able to summarize accessibility predicates and in particular to model recursive data structures, Viper allows defining named, parameterized assertions. These are called predicates and an example definition for a linked list can be found in Listing 2. Since automatic reasoning about statically unbounded recursive definitions is undecidable in general, Viper does not automatically unfold predicates. Instead, predicates need to be explicitly replaced by their body using an `unfold` statement. After the contained assertion is used, the predicate can be folded back again using a `fold`. Abstract predicates without a body can be used to model features that are not natively supported by Viper.

```
1 predicate list(this: Ref) {
2    acc(this.val) && acc(this.next)
3    && (this.next != null ==> list(this.next))
4 }
```

Listing 2: Definition of the list predicate

Similarly to fields, predicate instances can be inhaled and exhaled. In fact, folding and unfolding require the corresponding permissions to be available. It is also possible to inhale, exhale, fold and unfold fractional permis-

sion amounts of a predicate. During folding and unfolding all contained permission amounts will be multiplied by the fraction. In contrast to field permissions, fractions above one are allowed for predicates. The currently held permission amount can be checked by using the `perm(P(..))` expression.

```
1 inhale list(l)
2 unfold acc(list(l), 1/2)
3 value := l.val
4 l.val := 3       // will fail
5 fold acc(list(l), 1/2)
```

Listing 3: Example for usage of predicates

The example in Listing 3 illustrates the use of predicates. Even though we hold full permission to the `list` predicate, we only unfold half of it. According to the definition in Listing 2, the unfold results in read permissions to the `val` and `next` fields of the list element as well as read access to the following elements if there are any. Hence, we are only able to read the element's value and are not allowed to write to it, despite having full permission for the `list` predicate in the beginning.

Methods

Viper's statements as presented in Listings 1 and 3 need to be encapsulated in methods. These form separate verification units for modular verification. The interface to callers consists of the method's signature and a contract of pre- and postconditions, defined with the `requires` and `ensures` keywords. Methods with a body definition are verified by inhaling the preconditions in the beginning and exhaling the postconditions in the end. When verifying a caller, the preconditions will be exhaled before the call and the postconditions inhaled afterwards. This way, heap locations that are not completely exhaled by the preconditions can be framed across the method call, meaning all value information is preserved. On the other hand, permissions that are completely absorbed can imply a write inside the method and hence, the value is undefined afterwards.

The example in Listing 4 shows a client method that receives two references together with write permission for their `val` fields. After setting both fields to negative numbers, the method `abs` is called on `b`. Because of the precondition, the full permission to `b.val` is exhaled and only afterwards inhaled again together with the non-negativity constraint. Since the permission for `a.val` was never lost, we can ensure its value in the client's postcondition. Even though the body of the `abs` method is defined, the client cannot assert

```
1  method client(a: Ref, b: Ref)
2     requires acc(a.val) && acc(b.val)
3     ensures acc(a.val) && acc(b.val)
4     ensures a.val == -2
5     ensures b.val >= 0
6     ensures b.val == 5    // will fail
7  {
8     a.val := -2
9     b.val := -5
10    abs(b)
11 }
12
13 method abs(x: Ref)
14    requires acc(x.val)
15    ensures acc(x.val)
16    ensures x.val >= 0
17 {
18    if (x.val < 0) {
19       x.val := -x.val
20    }
21 }
```

Listing 4: Interaction of two Viper methods

that `b.val` is five, since the modular verification only takes the specifications into account.

Functions

Another top-level declaration type in Viper are functions. They abstract over side-effect free expressions and hence are functions in the mathematical sense. A major use case of functions are recursive expressions. In contrast to predicates, functions do not need to be explicitly unfolded, but will be replaced by their body once and then only when a mentioned predicate is unfolded. Similar to methods, functions can have pre- and postconditions, but no permissions can be mentioned in the postcondition, as the precondition must still hold afterwards. Functions can also be abstract or uninterpreted when the body is omitted.

Listing 5 shows an example function declaration. It computes the length of a linked list, that was defined in the previous examples. To be able to traverse the list, it requires permissions for the list predicate (line 2). This predicate is temporarily unfolded using the `unfolding` expression. The list is then traversed recursively and 1 is added for each element.

```
1  function list_length(l: Ref): Int
2     requires list(l)
3  {
4     unfolding list(l) in
5        l.next == null ? 1 : 1 + list_length(l.next)
6  }
```

Listing 5: Definition of a recursive Viper function

Quantifiers

Assertions in Viper can contain quantifiers, but SMT solvers cannot fully support them. Therefore, existentials might not always be provable (in a reasonable amount of time). Also, when universal quantifiers occur in an inhale or in the left-hand side of an implication, they are not actually interpreted universally by the SMT solver. Instead, the quantified expression is only instantiated if an explicit trigger expression, also known as pattern, is matched. If not provided, these triggers will be automatically inferred by Viper, but the heuristic-based inference does not work well in all cases. Typically, one provides explicit triggers that are function applications on the quantified variable.

```
1  function f(x: Int): Int
2     ensures result >= 0
3  function g(x: Int): Int
4
5  method quantifiers()
6  {
7     assume forall x: Int :: {f(x)} g(x) == f(x)
8     assert forall x: Int :: g(x) >= 0   // fails
9     assert forall x: Int :: f(x) == g(x)
10 }
```

Listing 6: Demonstration of triggering for universal quantifiers

Listing 6 shows an example where the trigger is chosen badly. There are two abstract functions `f` and `g` and we only know that `f` has non-negative results. If we assume that the results of both functions are the same for all inputs, we should be able to verify that `g(x)` is also non-negative for any `x`. But the verification of the assertion in line 8 fails, since the first quantifier is only instantiated for values `x` on which `f` is applied. Therefore, the assertion only mentioning `g(x)` fails, whereas the assertion in line 9 succeeds.

2.2.2 Prusti

As mentioned in the introduction, the programming language Rust [21] provides stronger safety guarantees than previous systems programming languages. Most of these guarantees are enabled by its so-called ownership type system. It ensures that at each point during an execution, there is either one exclusive usable binding which can modify a memory location or arbitrarily many shared read references. This improves memory safety, as it statically prevents data races, dangling pointers, undesirable aliasing side effects and other common memory-related failures.

This concept of exclusive write and shared read capabilities closely resembles Viper's access permissions concept, as explained in the previous section. Therefore, the Prusti project was able to automatically infer all permission logic needed in its front-end verifier for Rust [2]. This is a huge advantage, as Prusti does not require any initial specifications, but only the properties that are relevant for the user need to be specified. The user is also not required to have any knowledge of the underlying permission logic of Viper.

This inferred core proof can be understood as a re-verification of Rust's memory safety guarantees, but it enables proving other properties on top of it. For example, Prusti supports verifying the absence of panics, which are Rust's runtime failures. This includes failing `assert!` statements in the Rust code. It is also possible to statically ensure the absence of overflows. Finally, users can provide their own specifications in Rust syntax with a few extensions.

Our technique for the verification of resource bounds is not Rust-specific, but we have implemented it as an extension of Prusti. Therefore, all examples presented in this thesis will be written in Rust (or Viper) syntax.

Specification Syntax

The specifications are given by using Rust's built-in attributes and macros. Attributes are defined using this syntax above function declarations: `#[attribute]`. Similarly to Viper, pre- and postconditions can be defined using the `requires(...)` and `ensures(...)` attributes. Prusti specifications that can be used in these contracts are a superset of Rust boolean expressions, but need to be side-effect free. The extensions to the Rust syntax include implications via `==>` and quantifiers, written for example as `forall(|var: type, ...| filter ==> expression)`.

Rust functions can also have the `trusted` attribute, which can be used if the body of the function cannot be verified by Prusti to generate body-less methods in Viper. Marking functions as `pure` will generate Viper functions which can be used in Prusti's specifications. Obviously, such functions need to be deterministic and side-effect free and can therefore only take immutable

references. Finally, there are loop invariants which can be specified using Prusti specifications inside the `body_invariant!(...)` macro. In contrast to the most common loop invariant definition, these body invariants do not need to hold when the loop condition is false and hence cannot be assumed after the loop.

Viper Encoding

Prusti works as a plug-in to the Rust compiler. It intercepts the compilation to retrieve the user specifications and access Rust's intermediate representations MIR and HIR (mid-level and high-level IR). The MIR is translated into Viper code, which is verified by a Viper backend. In the end, verification failures are translated back to proper error messages on the Rust-level.

Since variables can be referenced in Rust and therefore, need to have access permissions for the transfer to references, variables are encoded as Viper references. The corresponding permissions are represented by a Viper predicate for each type. For primitive types this contains just the access permissions for the value field by default. With optional flags more information can be encoded, such as the non-negativity of unsigned integers, as in the following example:

```
predicate u32(self: Ref) {
   acc(self.val_int, write) && 0 <= self.val_int
}
```

The necessary `fold` and `unfold` operations to use these predicates are automatically inferred inside Prusti.

Functions in the Rust code are encoded as Viper methods by default. Viper functions are used when the Rust function is marked as `pure`. In contrast to what we have described in the Viper section, pre- and postconditions are encoded as explicit inhales and exhales, also for calls.

More details about Prusti can be found in the paper mentioned before [2] or online[4].

[4] https://www.pm.inf.ethz.ch/research/prusti.html

Chapter 3

Reasoning Principles

This chapter presents the basic principles of our approach to complexity verification, independently of its implementation in Prusti or Viper. The first section explains how we reason about program complexities using resource credits. In the next section we introduce the running example that will be used throughout this thesis. Then we explain how we verify the resource usage of loops and recursive functions. In Section 3.4 we define our representation of credits which is coefficient-based like in AARA [17]. How we use this representation will be explained in detail in the following three sections. Section 3.5 shows a syntax for specifying costs ensuring their non-negativity. Section 3.6 explains our reasoning about credits in terms of coefficients. And Section 3.7 presents how the representation is transformed throughout the program. Finally, we discuss a possible approach for amortized analysis.

3.1 Resource Credits

As mentioned in the introduction, our general goal is to verify bounds on the resource usage of (Rust) programs using a procedure-modular verifier. For this verification we use resource credits to represent the right to use a certain amount of a consumable resource. This means we aim to prove Hoare triples of the following form (omitting the functional specification and therefore using $\langle\cdot\rangle$ brackets):

$$\langle (P_0 \to \$_{R_0} c_0) \star \cdots \star (P_n \to \$_{R_n} c_n) \rangle \;\; \texttt{procedure} \;\; \langle \$_{R_0} 0 \star \cdots \star \$_{R_n} 0 \rangle \qquad (3.1)$$

In contrast to the definition in Section 2.1.2, we only require a certain amount of resource credits in the precondition and the remaining credit amounts in the postcondition to be ≥ 0, which gives us upper bounds on the resource consumption of each procedure. For modular verification these bounds are

then assumed to hold in callers of the procedure. Also note that we allow defining credit amounts that are only required under a certain condition P_i (which can be $\top$, a tautology, to define an unconditioned amount). Care has to be taken when such conditions overlap, because in that case the actual upper bound on the execution is the sum of all credit amounts from the overlapping conditions.

To verify such bounds, we perform a number of reasoning steps that essentially prove a Hoare triple for each operation. Starting with the credit amount in the precondition, we monotonically decrease it while conceptually stepping over the operations in program order. The amount of credits that are subtracted in each step are (at least) the credit cost of the corresponding operation. A monotonic decline together with the postcondition of having ≥ 0 credits left is required to ensure that the credits defined in the precondition are an actual upper bound on the resource usage (assuming we are reasoning about a resource that can only be consumed once and cannot be regained).

The costs of an operation are defined as in definition 3.1 either in the precondition of a procedure (for procedure calls) or in a *cost model* for each operation. Each cost defined in the precondition of such a Hoare triple needs to be non-negative to ensure a monotonically decreasing credit amount. Since we reason on the abstraction level of resource credits, our approach is independent of the specific resource that is being analyzed. The semantic meaning of a resource credit is separately defined by the cost model (and procedure preconditions).

Cost models can be defined for every quantity that might be interesting, such as running time, (peak) memory usage, energy consumption or number of additions. However, for some of these quantities it might be difficult to define the correct cost on the abstract level of a high-level programming language such as Rust or even its intermediate representation.

3.2 Binary Counter Example

Throughout this thesis, we will use the example in Listing 7 for demonstration purposes. It contains a Rust implementation of a binary counter similar to the one described in Section 2.1.1 and presented in Figure 6 of [8], except this one has a dynamic size. The current count is contained in the field in line 2. It is modeled by a vector of booleans which represents the bits with the least significant bit in the first element.

The counter has a single method (or function in Rust terms) that increments the counter by adding 1 to its binary count. To do so, it iterates through the vector flipping the bits as long as they are all one (i.e. true) (lines 8-11). It

then sets the next bit (a zero bit) to one, extending the vector with a `push` if the end is reached.

```
1  struct BinaryCounter {
2      count: Vec<bool>,
3  }
4  impl BinaryCounter {
5      #[requires(time_credits(O(self.count.len())))]
6      fn increment(&mut self) {
7          let mut i = 0usize;
8          while i < self.count.len() && self.count[i] == true {
9              self.count[i] = false;
10             i += 1;
11         }
12
13         if i < self.count.len() {
14             // ==> self.count[i] == false
15             self.count[i] = true;
16         }
17         else {
18             //#[requires(time_credits(O(self.len())))]
19             self.count.push(true);
20         }
21     }
22 }
```

Listing 7: Binary counter implementation in Rust

In line 5 we added a precondition. It represents the credit specification in the Hoare triple in definition 3.1. Here, we only use a single unconditioned credit amount $\$_R c$. The syntax for specifying credit bounds is our proposed extension to the Prusti specification syntax, which will be explained in more detail in Section 5.5. Since the precondition specifies `time_credits`, the resource R is runtime in this case. We are also specifying the number of credits c to be asymptotic in the length of the current bit vector, as that is the maximal number of loop iterations that will occur. The exact meaning of this asymptotic bound will be explained in Section 4.

There is also a bound on the `push` function of Rust's `Vec`, whose implementation and specification is not part of the listing, but the credit requirement is copied to the comment in line 18. As the `push` might need to copy the entire content of the vector to a new location when expanding it, we assume a linear worst-case complexity bound. In most real-world cases you would want to use the amortized constant bound instead, but we do not, since we

need the linear bound for a better demonstration of our approach in the next sections.

3.3 Loops and Recursion

A static verifier cannot verify a dynamic amount of loop iterations by unrolling them. Therefore, loops are typically verified by proving invariants for their body, similar to the induction step in induction proofs.

For our credit verifier we could instead also prove a bound on the credits used in the loop body and then compute the total bound on the loop as the product of the maximal number of loop iterations and the bound on the body. This would be possible for the loop in lines 8–11 of Listing 7. The loop body may consume a single time credit if we assume a cost model where only assignments to an element of `count` cost one credit. The maximal number of iterations is bounded by the current length of the `count` vector (and the length is not changed in the loop body). Therefore, we would be able to prove the bound `self.count.len()` $\cdot 1$ on the loop, as desired.

However, in general it might not be easy to find an explicit bound on the number of loop iterations. Also this method might be too coarse to compute a tight bound when the resource usage varies a lot between loop iterations. Furthermore, when the bound for the loop body is given asymptotically this approach might lead to wrong results as pointed out in section 3.3 of [13].

For these reasons, we decided to instead require a loop invariant that specifies the total (asymptotic) amount of credits needed for all remaining loop iterations. Such invariants can be specified using the built-in `body_invariant!` macro of Prusti.

In our example a credit invariant could be `self.count.len()` $-$ `i`, as shown in Listing 8. This invariant means we have `self.count.len()` credits available in the beginning (since `i` is zero). By incrementing `i` we decrease the credit amount in the invariant by 1 in each iteration, which gives us enough credits (in our cost model) to pay for the assignment in line 3. The credit amount specified in the invariant also stays non-negative, because the loop condition ensures that `i` $<$ `self.count.len()`.

```
1 while i < self.count.len() && self.count[i] == true {
2     body_invariant!(time_credits(self.count.len() - i));
3     self.count[i] = false;
4     i += 1;
5 }
```

Listing 8: Binary counter loop with invariant

This invariant approach has the additional advantage that it is very similar to how we would treat recursive functions. In fact, we can view the verification of a loop as the verification of a recursive function with the invariant in the precondition. For our example this equivalent recursive function is presented in Listing 9.

```
1 #[requires(*i < self.count.len() && self.count[*i] == true)]
2 #[requires(time_credits(self.count.len() - *i))]
3 fn loop_fn(&mut self, i: &mut usize) {
4     self.count[*i] = false;
5     *i += 1;
6     if *i < self.count.len() && self.count[*i] == true {
7         self.loop_fn(i);
8     }
9 }
```

Listing 9: Binary counter loop as recursive function

This view is useful to understand how a credit loop invariant would be verified. If we enter the loop, we will subtract the credits mentioned in the invariant as we would do for a function call. Recall that body invariants are only checked when the loop condition holds. That is why the loop condition occurs as a precondition in line 1 which needs to be checked in `increment` before calling the function.

At the beginning of the loop or function body we then assume we have the credit amount that is specified in the invariant available. Inside the loop body we can only make use of these credits, as we could otherwise use the credits from outside the loop multiple times (in each loop iteration). If the loop condition still holds after the loop, we will need to make sure that there are enough credits left over to pay for the following loop iterations. Therefore, the loop invariant (evaluated in the state after the loop body) is subtracted, as it would be done for a normal function call like in the recursive function on line 7.

3.4 Coefficient-based Representation

We represent the credit amounts internally by non-negative coefficients a_i of some base functions $b_i(\sigma)$ like in AARA [17]. This simplifies proof obligations, since it mainly requires reasoning in terms of coefficients.

$$\$_R c := \$_R \sum_{i=0}^{k} a_i \cdot b_i(\sigma), \qquad a_i \in \mathbb{Q}_0^+ \tag{3.2}$$

In contrast to the method used in AARA, the coefficients a_i are not single abstract variables, but some expressions containing concrete and abstract (i.e. quantified) *constants*. This means, that the coefficients must not depend on the program state σ. We additionally restrict the base functions $b_i(\sigma)$ to the following form:

$$b_i(\sigma) := \pm \prod_{r=0}^{l} (\llbracket expr_r \rrbracket_\sigma)^{e_r}, \qquad e_r \in \mathbb{N}^+,\ \llbracket expr_r \rrbracket_\sigma \geq 0 \tag{3.3}$$

We use $\llbracket \cdot \rrbracket_\sigma$ to denote evaluation of program expressions in state σ. If the evaluation does not depend on the program state, σ might also be omitted. The base functions can be either positive or negative. This enables modeling negative coefficients using positive coefficients for a negative base function. The product may also be empty, in which case we get $b_i(\sigma) = 1$ and $b_i(\sigma) = -1$ by convention (since 1 is the multiplicative identity).

$expr_r$ may be any side-effect free (i.e. pure) program expression that evaluates to a *non-negative number*. It should depend on the program state σ and must not contain any arithmetic operations like addition or multiplication. In particular, this includes program variables and formal arguments, but also the size of data structures like in line 5 of Listing 7. The expression also has to be well defined in the corresponding program state, which implies only (function applications on) formal arguments or global variables may be used in preconditions.

In essence, this means we reason about credits in terms of (conditional) *multivariate polynomials* and hence can only verify such bounds. However, at least in theory, we could allow other base functions to enable the verification of more complex bounds, but this might lead to proof obligations that are beyond the arithmetic capabilities of an SMT solver like Z3, especially by making the coefficients resulting from the conversions mentioned in Section 3.7 too complex.

3.5 Defining Coefficient-based Costs

As mentioned in Section 3.1, we require credit costs to be non-negative for a monotonically decreasing credit amount. Therefore, we need to verify that costs specified by a user in a cost model or as method preconditions are non-negative. However, checking the non-negativity of an arbitrary polynomial as a whole might be too difficult for an SMT solver due to possibly super-linear constraints in multiple variables. Therefore, we break down the non-negativity constraint by restricting the possible user specifications.

To simplify the non-negativity check for user-provided credit polynomials, we do not allow arbitrary multivariate polynomials with negative base functions, as specified in definitions 3.2 and 3.3. Instead, negative base functions

can only be used by subtracting two $expr$ (defined like the $expr_r$ before) at the base of an exponentiation. This means credit costs can only be specified in the following form:

$$\$_R \sum_{i=0}^{k} a_i \cdot \overline{b}_i(\sigma), \qquad a_i \in \mathbb{Q}^+ \tag{3.4}$$

$$\text{where } \overline{b}_i(\sigma) := \prod_{r=0}^{l} (\llbracket \overline{expr}_r \rrbracket_\sigma)^{e_r}, \qquad e_r \in \mathbb{N}^+ \tag{3.5}$$

$$\text{and } \llbracket \overline{expr}_r \rrbracket_\sigma := \begin{cases} \llbracket expr_r \rrbracket_\sigma & \text{or} \\ \llbracket expr_r^{(0)} \rrbracket_\sigma - \llbracket expr_r^{(1)} \rrbracket_\sigma \end{cases} \tag{3.6}$$

$$\llbracket expr_r \rrbracket_\sigma \geq 0,\ \llbracket expr_r^{(0)} \rrbracket_\sigma \geq 0,\ \llbracket expr_r^{(1)} \rrbracket_\sigma \geq 0$$

Since the coefficients a_i are required to be non-negative and all $expr$ are non-negative, using this form, we only need to separately check non-negativity of the differences $\llbracket expr_r^{(0)} \rrbracket_\sigma - \llbracket expr_r^{(1)} \rrbracket_\sigma$, which can be ensured by a simple side condition: $\llbracket expr_r^{(0)} \rrbracket_\sigma \geq \llbracket expr_r^{(1)} \rrbracket_\sigma$. If the credits are only required under a certain condition P_i, the non-negativity checks are also only necessary under this condition.

We allow specifying negative base functions like this, since it is a recurring pattern needed in credit specifications, especially for loops and recursive functions, as we have seen in Listing 8. We could also allow more complex arithmetic expressions at the base of the exponentiation, but this would again complicate the non-negativity checks and also the automatic translation to our normal form. Such expressions can still be modeled, but require rewriting the program to take the result of the expression as an (ghost) argument. The formal argument can then be used in specifications.

We replace such user specifications internally with a representation in our normal form from definitions 3.2 and 3.3. This representation can be computed using binomial expansion as follows:

$$\begin{aligned}
& a_i \cdot \left(\llbracket expr_r^{(0)} \rrbracket_\sigma - \llbracket expr_r^{(1)} \rrbracket_\sigma \right)^{e_r} \\
= \quad & \left(\binom{e_r}{0} \cdot a_i \right) \cdot \left((\llbracket expr_r^{(0)} \rrbracket_\sigma)^{e_r} \quad \cdot (\llbracket expr_r^{(1)} \rrbracket_\sigma)^0 \right) \\
& + \left(\binom{e_r}{1} \cdot a_i \right) \cdot \left(-(\llbracket expr_r^{(0)} \rrbracket_\sigma)^{e_r - 1} \cdot (\llbracket expr_r^{(1)} \rrbracket_\sigma)^1 \right) \\
& + \ldots \\
& + \left(\binom{e_r}{e_r} \cdot a_i \right) \cdot \left(\pm(\llbracket expr_r^{(0)} \rrbracket_\sigma)^0 \quad \cdot (\llbracket expr_r^{(1)} \rrbracket_\sigma)^{e_r} \right)
\end{aligned}$$

This is the same base conversion as defined for $\llbracket expr^{(0)} + expr^{(1)} \rrbracket_\sigma$ in Table 3.1. The amount of base functions gets increased in the order of the product

of all $(e_r + 1)$ where $[\![\overline{expr}_r]\!]_\sigma$ is a difference. These exponents and the number of differences are usually very small. Therefore, this is not a big issue for the verifier performance on usual complexity bounds.

Applying this translation to the credit amount `self.count.len() − i` from the loop invariant in Listing 8 is very simple, since the exponent is one:

$$\begin{aligned} a_0 \cdot \bar{b}_0(\sigma) &= 1 \cdot ([\![\texttt{self.count.len()}]\!]_\sigma - [\![\texttt{i}]\!]_\sigma)^1 \\ &= 1 \cdot \binom{1}{0} \cdot [\![\texttt{self.count.len()}]\!]_\sigma + 1 \cdot \binom{1}{1} \cdot (-[\![\texttt{i}]\!]_\sigma) \\ &= a_0 \cdot b_0(\sigma) + a_1 \cdot b_1(\sigma) \end{aligned}$$

The translation of a more complex example is given in Appendix A.

3.6 Coefficient-based Reasoning

In our coefficient-based representation costs are subtracted coefficient-wise. Which means the current credit amount needs to be available in terms of large enough coefficients for the same base functions. Otherwise, we could not preserve the non-negativity of the coefficients as required in Definition 3.2.

For example, if we assume that the call to `push` in line 19 of Listing 7 costs exactly `self.len()` credits, we will require a coefficient that is at least 1 for the base function $([\![\texttt{self.len()}]\!]_\sigma)^1$ in the credit amount before the call. We can then subtract 1 from this coefficient.

More formally, if we encounter a statement with the following credit specification:

$$\langle \$_R \sum_{i=0}^{k} cost_i \cdot b_i(\sigma) \rangle \ \texttt{statement} \ \langle \$_R 0 \rangle, \qquad cost_i \geq 0$$

We can deduce:

$$\begin{aligned} &\langle P \to \$_R \sum_{i=0}^{k} a_i \cdot b_i(\sigma) \star Q \to \$_R \sum_{i=0}^{k} a'_i \cdot b_i(\sigma) \rangle \\ &\ \texttt{statement} \\ &\langle P \to \$_R \sum_{i=0}^{k} (a_i - cost_i) \cdot b_i(\sigma) \star Q \to \$_R \sum_{i=0}^{k} (a'_i - cost_i) \cdot b_i(\sigma) \rangle \\ &\text{only if } (P \vee Q) \wedge (P \to a_i - cost_i \geq 0) \wedge (Q \to a'_i - cost_i \geq 0) \end{aligned}$$

For simplicity we only show two conditional credit amounts here. Note that one of them needs to be true, such that we have any amount of credits available for the subtraction.

Furthermore, there is no condition for the cost. If the cost contains conditional conjuncts, we subtract each cost only if the corresponding condition is satisfied. This means, to deal with overlapping conditions, each combination of condition values needs to be taken into account with the cost being the sum of all satisfied conditions:

$$
\begin{aligned}
&(P \to \$_R \sum_{i=0}^{k} cost_i \cdot b_i(\sigma)) \star (Q \to \$_R \sum_{i=0}^{k} cost'_i \cdot b_i(\sigma)) \\
\Leftrightarrow\ &((P \wedge Q) \to \$_R \sum_{i=0}^{k} (cost_i + cost'_i) \cdot b_i(\sigma)) \\
&\star ((P \wedge \neg Q) \to \$_R \sum_{i=0}^{k} cost_i \cdot b_i(\sigma)) \\
&\star ((\neg P \wedge Q) \to \$_R \sum_{i=0}^{k} cost'_i \cdot b_i(\sigma))
\end{aligned}
$$

Procedure postconditions are also verified coefficient-wise. This means, coefficients for positive base functions need to be greater than or equal to the coefficient defined by the postcondition. Coefficients for negative base functions need to be smaller or equal. This is an over-approximation of the actual requirement, that the total amount of credits needs to be greater than or equal to the total amount defined in the postcondition. We do this approximation to again be able to mainly reason in terms of constant credits. As defined in Section 3.1, we require a non-negative amount of credits left for each resource in the postcondition, i.e. all coefficients are zero in the coefficient-based representation. This means, we do allow arbitrary coefficients for positive base functions, but only the coefficient 0 for negative ones.

3.7 Base Conversions

Credits in preconditions can only be defined in terms of program expressions that are accessible and well defined in the beginning of the corresponding procedure, in particular (function applications on) formal arguments. Therefore, credits may not be available in terms of the right base functions when a cost needs to be subtracted as described in the previous section. For example, for the recursive function in Listing 9 the credits from the precondition are defined in terms of `self.count.len()` and `*i`. However, the recursive function call requires the credits to be defined in terms of `self.count.len()` and the new value of `*i` that is the result of the incrementation in line 5.

Therefore, we need some conversion between coefficients of different base functions. We have already seen such a conversion in Section 3.5 for the

conversion of differences into proper base functions. For the assignment `i += 1` from the loop function, we could apply the following conversion:

$$\begin{aligned} & 1 \cdot [\![\texttt{self.count.len()}]\!]_\sigma + 1 \cdot (-[\![\texttt{i}]\!]_\sigma) + 1 \cdot (-1) \\ = \; & 1 \cdot [\![\texttt{self.count.len()}]\!]_{\sigma'} + 1 \cdot (-[\![\texttt{i}]\!]_{\sigma'}) \end{aligned}$$

This equation is valid, since we know from the assignment that $[\![\texttt{i}]\!]_{\sigma'} = [\![\texttt{i}]\!]_\sigma + 1$, where σ is the program state before the assignment and σ' is the state after it. After the conversion we can subtract the cost of the recursive call ($1 \cdot [\![\texttt{self.count.len()}]\!]_{\sigma'} + 1 \cdot (-[\![\texttt{i}]\!]_{\sigma'})$) coefficient-wise.

$[\![expr_s]\!]_{\sigma'}$		$a \cdot ([\![expr_s]\!]_{\sigma'})^e$
$=$	$\Rightarrow$	$=$
$[\![const]\!]$		$([\![const]\!])^e \cdot a$
$[\![expr]\!]_\sigma$		$a \cdot ([\![expr]\!]_\sigma)^e$
$[\![expr * const]\!]_\sigma$		$([\![const]\!])^e \cdot a \cdot ([\![expr]\!]_\sigma)^e$
$[\![expr^{(0)} * expr^{(1)}]\!]_\sigma$		$a \cdot ([\![expr^{(0)}]\!]_\sigma)^e \cdot ([\![expr^{(1)}]\!]_\sigma)^e$
$[\![expr + const]\!]_\sigma$	$\binom{e}{0} \cdot ([\![const]\!])^0 \cdot a$	$\cdot ([\![expr]\!]_\sigma)^e +$
	$\binom{e}{1} \cdot ([\![const]\!])^1 \cdot a$	$\cdot ([\![expr]\!]_\sigma)^{e-1} +$
	$\dots$	
	$\binom{e}{e} \cdot ([\![const]\!])^e \cdot a$	$\cdot ([\![expr]\!]_\sigma)^0$
$[\![expr^{(0)} + expr^{(1)}]\!]_\sigma$	$\binom{e}{0} \cdot a$	$\cdot ([\![expr^{(0)}]\!]_\sigma)^e \cdot ([\![expr^{(1)}]\!]_\sigma)^0 +$
	$\binom{e}{1} \cdot a$	$\cdot ([\![expr^{(0)}]\!]_\sigma)^{e-1} \cdot ([\![expr^{(1)}]\!]_\sigma)^1 +$
	$\dots$	
	$\binom{e}{e} \cdot a$	$\cdot ([\![expr^{(0)}]\!]_\sigma)^0 \cdot ([\![expr^{(1)}]\!]_\sigma)^e$
$\dots$		

Table 3.1: Possible base conversions

Table 3.1 lists some of the base conversions that are possible, but one could define arbitrarily many such equivalence facts.For a conversion to be valid, some equivalence for part of the base function must hold. Typically, we are conditioning on the equivalence of a single $expr_s$ that occurs in the base function $b_i(\sigma) = \pm \prod_{r=0}^{l} ([\![expr_r]\!]_\sigma)^{e_r}$ (i.e. $s \in [0, l]$). The right-hand side of the equivalence is shown in the first column of the table. Note that, since all $expr$ are defined to be non-negative, the result of the arithmetic operation on the right-hand side also needs to be non-negative.

The corresponding conversion conceptually replaces the right-hand side of

the condition equality by $[\![expr_s]\!]_\sigma$ (or vice-versa). The resulting mathematical expression then needs to be transformed back into our normal form for polynomials, as specified in Definitions 3.2 and 3.3. Therefore, we need to separate constants that become part of the coefficient and non-constant expressions. The non-constant expressions are also split into separate exponentiations. The resulting conversions are shown in the column on the right of the arrow ($\Rightarrow$) in the table. The coefficient after the conversion is shown on the left of the middle $\cdot$, the exponentiations on the right. For an assignment, like the incrementation in our example, $a \cdot ([\![expr_s]\!]_{\sigma'})^e$ may be an replacement for the expression shown in the column below it.

The first row of the table shows arguably the simplest conversion. When $[\![expr_s]\!]_{\sigma'}$ is equal to some constant program expression $[\![const]\!]$, we can replace it by the constant (or vice-versa). The constant will be exponentiated and added to the coefficient. Therefore $a \cdot ([\![expr_s]\!]_{\sigma'})^e = ([\![const]\!])^e \cdot a$.

The second conversion is based on the equivalence between $[\![expr_s]\!]_{\sigma'}$ and another expression $[\![expr]\!]_\sigma$ and just is a one-to-one replacement.

The next conversion is a combination of the previous ones. It multiplies the expression $expr$ by a constant $const$. Therefore, the exponentiation of the constant is added to the coefficient and $expr$ replaces $expr_s$. A similar rule can also be applied for divisions by a non-zero constant. The corresponding conversion is omitted in the table for simplicity, but can be retrieved by viewing the division as a multiplication by the inverse of the constant.

When multiplying two non-constant expressions as in the next row, both get added to the base function. Here we do not allow divisions, since the result would not be a polynomial with non-negative exponents.

The last two conversions shown are more complicated, since the sum $(a+b)^e$ has to be separated by binomial expansion. This means we replace one base function by $e+1$ new ones. The coefficients gets extended by binomial coefficients. When a constant is added, also the corresponding part becomes part of the coefficient as before. The same conversions also apply to subtractions by conceptually replacing the second summand by a negative one. In this case the sign of the base function will be changed for odd exponents. Keep in mind that the result of the subtraction is still required to be non-negative and therefore the overall credit amount stays non-negative.

In the table we only show the part of the base function $b_i = \pm \prod_{r=0}^{l} ([\![expr_r]\!]_\sigma)^{e_r}$ that is actually converted. Each expression on the right-hand side of the table therefore needs to be multiplied by $\pm ([\![expr_s]\!]_\sigma)^{e_s - e} \cdot \prod_{r=0,\ r \neq s}^{l} ([\![expr_r]\!]_\sigma)^{e_r}$ to retrieve the actual base functions. Note that we allow converting only a part $e \in [0, e_s]$ of the exponent e_s. Likewise, only a part $a \in [0, a_i]$ of the coefficient a_i might be converted, leaving $(a_i - a) \cdot \pm \prod_{r=0}^{l} ([\![expr_r]\!]_\sigma)^{e_r}$ behind.

Base conversions are typically applied when an assignment took place. Since assignments in Rust's intermediate representation MIR can have at most two operands on their right-hand side[1], the conversions described here cover almost all relevant assignments for our Rust verifier. However, one could also apply them when an equivalence fact is provided by a postcondition or a branching condition (which might be more difficult to do automatically). If an expression occurs only in positive or negative base functions (with a non-negative coefficient), one could even apply a conversion if an inequality holds by ensuring that the credit amount could only decrease.

3.8 Amortized Analysis

Since our method reasons in terms of resource credits like the banker's approach to amortized analysis (see Section 2.1.1), it is in principle suited to verify amortized bounds as well. Here, we present an approach to perform such an amortized analysis in our tool. It has not been implemented yet, but we manually tested it on the binary counter example.

To amortize costs, we need to be able to transfer credits from one method execution to another. Since we are working with a procedure-modular verifier, these credits cannot be passed explicitly, but we need some invariant on the data structure that specifies the currently stored credit amount (like a potential). The credits from the invariant, evaluated in the beginning of a method, can then be added to the available credits. To ensure that the invariant is preserved, the credits need to be subtracted at the end, according to the final state of the data structure. The same is done for the verification of loop bodies according to the recursive function equivalent described in Section 3.3.

The invariant often conditions on the state of single elements of the data structure. For random access data structures, like the vector in our binary counter, the invariant is usually stated globally with a universal quantifier over the index of the element access. For recursively defined data structures, like linked lists, that are traversed top-down, the invariant would normally be stated on the element level. This is very similar to what is done for access permissions in Viper.

Listing 10 contains the binary counter example with annotations for amortized analysis. The changes in comparison to Listing 7 are highlighted by the blue background color. The first three lines show the invariant on the data structure that specifies the stored credits. For each bit with the value one (true) we require at least one credit to be stored, i.e. enough to pay for

[1] see https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/enum.Rvalue.html#variant.BinaryOp (last accessed: 30.09.2021)

the flip back to zero (false). Note that the credit requirements for each index `i` sum up, as in the separating conjunctions we had before.

If we assume the same cost model as before, where only setting a bit in the vector has a cost of one, we will be able to prove that the loop does not require any credits in addition to the credits from the data structure invariant. Hence, we need no body invariant. The credits for setting the bit value inside the loop in line 14 can be taken from the invariant, since the loop condition tells us that the value was one (true) before, which matches the condition in the invariant, and we set the bit to zero. The potential specified by the invariant is hence decreased by (at least) one credit which is enough to pay for the assignment.

Here, we are also using the constant amortized runtime for the vector `push` instead of the worst case linear bound that we used before. Therefore, we are able to prove a constant amortized runtime for the `increment` function (line 8).

```
#[invariant(forall(|i: usize|
    (i < self.count.len() && self.count[i] == true)
    ==> time_credits(1))]
struct BinaryCounter {
    count: Vec<bool>,
}
impl BinaryCounter {
    #[requires(time_credits(O(1)))]
    fn increment(&mut self) {
        let mut i = 0usize;
        while i < self.count.len() && self.count[i] == true {
            // no invariant needed, since the loop cost
            // is completely amortized by the invariant
            self.count[i] = false;
            i += 1;
        }

        if i < self.count.len() {
            // ==> self.count[i] == false
            self.count[i] = true;
        }
        else {
            // #[requires(time_credits(O(1)))]
            self.count.push(true);
        }
    }
}
```

Listing 10: Binary counter with possible annotations for amortized analysis

Chapter 4

Asymptotic Bounds

Following the general goal of Prusti to simplify and minimize the user annotations needed, we aim for the verification of asymptotic bounds. They enable the user to omit possibly complicated expressions for the polynomial coefficients as well as lower-order base functions. As mentioned in Section 2.1.3, asymptotic bounds also make the bound independent from implementation and cost model details and are instead focusing on the algorithm used. This enables a more modular verification, since small implementation changes do not require a different asymptotic bound, which would trigger client re-verification.

This chapter first introduces how asymptotic bounds can be defined using the coefficient-based representation from the previous chapter. In the next section we show how asymptotic bounds on function calls should be modeled. Then Section 4.3 explains the difficulties that we face when proving asymptotic bounds and presents our solution using a static analysis. Finally, Section 4.4 explains the verification of loops and recursive functions in an asymptotic context.

4.1 Coefficient-based Definition

Formally, asymptotic bounds on a procedure in our coefficient-based representation are defined as follows:

$$\exists c_0 \in O\left(\sum_{i=0}^{k_0} \overline{b}_i\right), \ldots, c_n \in O\left(\sum_{i=k'_{n-1}}^{k_n} \overline{b}_i\right).$$
$$\langle (P_0 \to \$_{R_0} c_0) \star \cdots \star (P_n \to \$_{R_n} c_n) \rangle \ \texttt{procedure} \ \langle \$_{R_0} 0 \star \cdots \star \$_{R_n} 0 \rangle \quad (4.1)$$

Here we quantify over a Hoare triple with omitted functional specification (indicated by the brackets). Recall that the Hoare triple contains an implicit universal quantification over all pre-states that is evaluated inside the

existential. Using the existential quantification, we search for some credit amounts obeying the asymptotic bounds that provide enough credits for the `procedure` execution to succeed with a non-negative amount of credits left.

Also recall that each $\overline{b}_i$ represents a function on such a program state as defined in Section 3.5. Since asymptotic bounds are only given in user annotations, all of these only allow using negative base functions via a non-negative subtraction in the $\overline{b}_i$ as described before. This also implies that the sums in the big-O are always non-negative.

Because we are only reasoning in terms of multivariate polynomials as credits, the possible functions satisfying such an asymptotic bound can be approximated with a simpler condition than in Definition 2.3. We only need to ensure that there exist coefficients a_i such that the bounds are satisfied:

$$
\begin{aligned}
&(\exists a_0, a_1, \dots a_{k'_n} \in \mathbb{Q}_0^+ .\\
&\quad \langle (P_0 \to \$_{R_0} \sum_{i=0}^{k'_0} a_i \cdot \overline{b}_i(\sigma)) \star \dots \star (P_n \to \$_{R_n} \sum_{i=k'_{n-1}}^{k'_n} a_i \cdot \overline{b}_i(\sigma)) \rangle\\
&\quad \texttt{procedure}\\
&\quad \langle \$_{R_0} 0 \star \dots \star \$_{R_n} 0 \rangle)\\
&\wedge \quad \forall t \in [0,n].\ \forall j. \in (k_t, k'_t].\ \exists i. \in [k'_{t-1}, k_t].\quad \overline{b}_j \in O(\overline{b}_i) \qquad (k'_{-1} := 0)
\end{aligned}
\tag{4.2}
$$

Here we are assuming without loss of generality that for each $t \in [0,n]$ the first $k_t \leq k'_t$ base functions match the $\overline{b}_i$ in the asymptotic bound. For each additional base function $\overline{b}_j$ we need to show that it is asymptotically dominated by a base function $\overline{b}_i$ that was mentioned in the asymptotic bound.

This formulation implies that we obey the asymptotic bound as defined in 4.1. Note that the other direction does not hold, since we do not cover all possible functions that satisfy the asymptotic bound, but just approximate them.

The asymptotic dominance between two of our base functions in the above definition can simply be checked as follows — assuming all $expr_r$ and $expr_s$ only occur once in their respective products:

$$
\begin{aligned}
\overline{b}_j = \prod_{s=0}^{l'} (\llbracket \overline{expr}'_s \rrbracket_\sigma)^{e'_s} &\in O\left(\prod_{r=0}^{l} (\llbracket \overline{expr}_r \rrbracket_\sigma)^{e_r} \right)\\
\Leftarrow \quad \forall s \in [0,l'].\ \exists r \in [0,l].\ \overline{expr}'_s = \overline{expr}_r &\wedge e'_s \leq e_r
\end{aligned}
\tag{4.3}
$$

Given the asymptotic bound from our binary counter example in Listing 7 in the previous chapter, i.e.,

```
#[requires(time_credits(O(self.count.len())))]
```

definition 4.2 is instantiated as:

$$\exists a_0, a_1 \in \mathbb{Q}_0^+. \quad \langle \$_{time}(a_1 \cdot [\![\texttt{self.count.len()}]\!]_\sigma + a_0 \cdot 1) \rangle \ \texttt{increment} \ \langle \$_{time} 0 \rangle$$

The base functions $([\![\texttt{self.count.len()}]\!]_\sigma)^1$ and 1 are the only ones that are upper bounded by the only base function mentioned in the asymptotic bound, $([\![\texttt{self.count.len()}]\!]_\sigma)^1$. For the first the asymptotic dominance check succeeds, since it is identical to the bound and hence has the same *expr* and exponent. The 1 is represented by the empty product as mentioned before and hence the universal quantifier in condition 4.3 is trivially satisfied.

4.2 Function Calls

As mentioned in the beginning of this chapter, asymptotic bounds lead to fewer re-verifications of callers, because they subsume many different concrete credit amounts. To enable this abstract view, callers have to be verified only against this asymptotic bound specification.

The asymptotic bound tells us that there is some cost function satisfying the bound as defined in condition 4.1. This cost function needs to be a multivariate polynomial which is defined using positive coefficients for base functions $\overline{b}_i$ (since we can only verify those). But to remain modular, we cannot assume any additional properties for this cost function. Therefore, we need to prove the asymptotic bounds of callers for any multivariate polynomial costs of the called functions that satisfy their asymptotic bounds. This just replaces the existential quantifier in condition 4.2 by an universal quantifier for the verification of callers.

To make this more concrete, the condition for the bound on `increment` in Listing 7 would actually be verified as follows against the asymptotic specification for the `push` call in line 19:

$$\begin{aligned}
&\forall a_0^{(push)}, a_1^{(push)} \in \mathbb{Q}_0^+. \ \exists a_0, a_1 \in \mathbb{Q}_0^+. \\
&\quad \langle \$_{time}(a_1 \cdot [\![\texttt{self.count.len()}]\!]_\sigma + a_0 \cdot 1) \rangle \ \texttt{increment} \ \langle \$_{time} 0 \rangle \\
&\quad \text{where} \\
&\qquad \langle \$_{time}(a_1^{(push)} \cdot [\![\texttt{self.count.len()}]\!]_{\sigma'} + a_0^{(push)} \cdot 1) \rangle \\
&\qquad \texttt{self.count.push(true)} \\
&\qquad \langle \$_{time} 0 \rangle
\end{aligned}$$

By this notation we mean that it needs to be possible to prove the first Hoare triple using the specifications given in the second one when the corresponding statement is encountered. Credits specified in the precondition of called functions (as given by the second triple) are subtracted from the current credit amount as described in Section 3.6. This implies that a_0 and a_1 must depend on the abstract coefficients for the `push` call to make sure that there are enough credits to subtract the cost of the call without making the credit amount negative.

We also abstract the cost of calls to functions that have a non-asymptotic credit specification, as if they had an asymptotic bound specified. This gives us the same modularity improvement as for asymptotic bounds, since changes to the concrete credit specifications that do not change asymptotic behavior will not change the abstract bound for callers. However, when doing this abstraction on the call site, concrete (non-asymptotic) credit bounds need to refer to the abstract coefficients of called functions. This complicates defining concrete bounds even further, in contrast to asymptotic bounds.

4.3 Verifying Asymptotic Bounds

To the best of our knowledge, existentials around Hoare triples are not supported in existing procedure-modular verifiers, such as Viper. Instead, they prove Hoare triples over procedures in isolation. Program elements that are not inlined are only assumed to comply with their specifications and can be arbitrary otherwise. For example, abstract functions and called methods are implicitly universally quantified in Viper. Since the existential for the asymptotic bound occurs outside of the Hoare triple, we would additionally need to be able to encode global existentials after this universal quantification and before we quantify over all possible pre-states as required by the Hoare triple.

To avoid this problem, the existential could be translated to a guarded universal quantifier, which makes the order of quantifiers irrelevant. Consequently, we could use existing elements of the verification language such as variables or functions to represent the coefficients. This translation can be achieved by collecting constraints on the coefficients (from the structure of the procedure), such that the Hoare triple is valid for any coefficients satisfying the constraints. Additionally, one still has to prove that these constraints are satisfiable, i.e. that there exist coefficients such that all constraints are fulfilled. The complete translated verification goal looks as follows (omitting

the conditions P_i for simplicity):

$$\begin{aligned}
&\forall a_0, a_1, \ldots a_{k'} \in \mathbb{Q}_0^+. \quad \text{constraints}_{\texttt{procedure}}(a_0, a_1, \ldots a_{k'}) \\
&\qquad\qquad \rightarrow \langle \$_R \sum_{i=0}^{k'} a_i \cdot \overline{b}_i(\sigma) \rangle \ \texttt{procedure} \ \langle \$_R 0 \rangle \\
\wedge \quad &\exists a_0, a_1, \ldots a_{k'} \in \mathbb{Q}_0^+. \quad \text{constraints}_{\texttt{procedure}}(a_0, a_1, \ldots a_{k'}) \\
\wedge \quad &\forall j. \in (k, k']. \ \exists i. \in [0, k]. \ \overline{b}_j \in O(\overline{b}_i)
\end{aligned} \tag{4.4}$$

The newly introduced existential quantifier can be encoded in intermediate verification languages like Viper, since it does not contain a Hoare triple. Additionally, it does not depend on the program state and hence can be proven in a completely independent assertion. This approach is very similar to what AARA-based tools do by collecting linear constraints for an LP solver [19, 7].

Even though we can encode this approach into intermediate verification languages, this encoding does no perform well in Viper. As mentioned in Section 2.2.1, existentials like the one checking the satisfiability of the constraints are often not provable for the underlying SMT solver. Furthermore, we would also need to infer appropriate constraints outside of Viper.

Therefore, we tried to improve the encoding by encoding the existential as follows:

$$\neg \left(\forall a_0, a_1, \ldots a_k \in \mathbb{Q}_0^+. \quad \text{constraints}_{\texttt{procedure}}(a_0, a_1, \ldots a_k) \rightarrow \bot \right) \tag{4.5}$$

Here $\bot$ is defined to be never satisfiable. In Viper or other verifiers, this translates to assuming the constraints on coefficient variables and asserting `false` at the end. The existential will be satisfiable if the assertion fails to verify. This approach might work well, but it interferes with Prusti's functional verification. In Prusti, we want the assertions of functional properties to succeed, but we might not find their failures when the `assert false` fails.

4.3.1 Cost Inference

Since there is no satisfying solution to encode the existential around Hoare triples for our purposes, we decided to instead infer expressions for the coefficients and then check their validity in the verifier by using them in precondition as in the previous chapter. Additionally, we have to check that the inferred coefficients satisfy the asymptotic bounds by checking their corresponding base functions as in definition 4.2. Conceptually, this proves the existential by providing a concrete instantiation.

Our inference collects the required credits by walking backwards over the control-flow graph of the procedure. The credits are represented as condi-

tional multivariate polynomials (as defined in the previous chapter), i.e.:

$$\langle (P_0 \to \$_{R_0} c_0) \star \cdots \star (P_n \to \$_{R_n} c_n) \rangle \qquad \text{where } c_i = \sum_{j=0}^{k_i} a_j \cdot b_j(\sigma)$$

Conditions come from conditional branches (and from conditional credit requirements of calls) and hence model the program path that leads to a specific cost, similarly to a path constraint in symbolic execution. For unconditional asymptotic bounds on non-recursive functions the cost can also be approximated by summing all costs regardless of their path constraints, since the sum will not change the asymptotic behavior.

More specifically the inference works as follows. When a *call* or an *operation* that requires credits (in the cost model) is encountered, i.e.,

$$\langle (Q_0 \to \$_{R'_0} c'_0) \star \cdots \star (Q_m \to \$_{R'_m} c'_n) \rangle \ \texttt{call}\ \textit{or}\ \texttt{op}\ \langle \$_{R'_0} 0 \star \cdots \star \$_{R'_m} 0 \rangle$$

the corresponding cost is added to the currently inferred cost. This means we conjoin the cost to the previously inferred separating conjunction. To reduce the number of conjuncts, we can also add the credit amounts coefficient-wise if there is already a conjunct with the same condition (i.e. $P_i = Q_j$) and resource type (i.e. $R'_j = R_i$).

Conditional branches simply conjoin their condition to each predicate P_i from the corresponding branch. Assertions in the program code are treated the same by adding their assertion to all conditions inferred below.

More complicated is the handling of *assignments*. As in other backwards interpreters, we need to replace the left-hand side (LHS) of the assignment by the right-hand side (RHS), since the value of the LHS probably has changed due to the assignment. For the conditions P_i we perform exactly this substitution. However, we cannot simply do a syntactic substitution for the cost expressions c_i, since we want to preserve our coefficient-based representation of multivariate polynomials.

Therefore, we use the base conversions described in Section 3.7, but reversed. Due to the assignment, we know the equality $[\![LHS]\!]_{\sigma'} = [\![RHS]\!]_{\sigma}$ holds. Consequently, we can replace each $a_j \cdot \prod_{r=0}^{l} ([\![expr_r]\!]_\sigma)^{e_r}$ in c_i, where the LHS occurs as an $expr_s$, by the right-hand side of the conversion equality defined in the previous chapter. Since the LHS does not (necessarily) have the same value above the assignment, we eliminate it completely from each c_i, i.e. set $a := a_j$ and $e := e_s$ in the conversion.

Note that we do not perform such conversions for equalities inferred from specifications or conditionals, since for such equalities it is often not clear if the conversion should be performed, as will be explained in more detail in Section 5.2.

An example for the cost inference is shown in Listing 11 at the end of this chapter. It contains the increment function of our binary counter with the intermediate inference results as comments. As denoted at the top of the function body, we are able to infer the following cost:

$$
\begin{aligned}
&(\llbracket \texttt{i < self.count.len()} \rrbracket_\sigma \to \$_{time} 1 \cdot 1) \\
&\star\ (\llbracket \texttt{i >= self.count.len()} \rrbracket_\sigma \to \$_{time}(a_1^{(push)} \cdot \llbracket \texttt{self.count.len()} \rrbracket_\sigma + a_0^{(push)} \cdot 1)) \\
&\star\ \$_{time}(a_1^{(loop)} \cdot \llbracket \texttt{self.count.len()} \rrbracket_\sigma + a_0^{(loop)} \cdot 1)
\end{aligned}
$$

This cost is compliant with the asymptotic bound, since it only contains the base functions $\llbracket \texttt{self.count.len()} \rrbracket_\sigma$ and 1.

4.4 Loops and Recursion

When dealing with asymptotic bounds as loop invariants or preconditions of recursive functions, additional care needs to be taken. In contrast to normal function calls, the abstract coefficients for recursive function calls cannot be universally quantified, since they need to be the same as the coefficients from the precondition. Therefore, the condition to verify for our example in Listing 9 would become:

$$
\begin{aligned}
&\exists a_0, a_1 \in \mathbb{Q}_0^+. \\
&\quad \langle \$_{time}(a_1 \cdot \llbracket \texttt{self.count.len()} \rrbracket_\sigma + a_1 \cdot (-\llbracket \texttt{i} \rrbracket_\sigma) + a_0 \cdot 1) \rangle\ \texttt{loop\_fn}\ \langle \$_{time} 0 \rangle \\
&\quad \text{where} \\
&\qquad \langle \$_{time}(a_1 \cdot \llbracket \texttt{self.count.len()} \rrbracket_{\sigma'} + a_1 \cdot (-\llbracket \texttt{i} \rrbracket_{\sigma'}) + a_0 \cdot 1) \rangle \\
&\qquad \texttt{self.loop\_fn(i);} \\
&\qquad \langle \$_{time} 0 \rangle
\end{aligned}
$$

Finding coefficients to instantiate the existential then means solving a recurrence equation (or inequality). For the `loop_fn` function this recurrence would look like this:

$$\exists a_0, a_1 \in \mathbb{Q}_0^+.\ a_1 - a_1 \geq 0\ \wedge\ a_0 - a_0 + a_1 - 1 \geq 0$$

Here the -1 comes from the cost of the assignment in line 4 of Listing 9. The $+a_1$ arises from the fact that `self.count.len() - i` gets decreased by 1 due to the assignment in line 5. More details regarding the treatment of assignments can be found in the next section and also more concretely in the next chapter.

In this case the solution is simply some a_0 and a_1 such that $a_1 \geq 1$. Unfortunately, these recurrences can get arbitrarily complex. Therefore, they are hard to solve automatically in general. However, if we restrict the space of

recursive functions, there will be methods to solve the recurrences, e.g. the master theorem or extensions of it for divide-and-conquer algorithms, as described in [29], or using an LP solver for linear constraints, as in the AARA implementations mentioned in Section 2.1.4. Our implementation does not use one of these methods yet and Viper cannot really solve the recurrences either. This means, we unfortunately require the user to solve the recurrence to make the verification of asymptotic bounds on recursive functions sound.

```
#[requires(time_credits(O(self.count.len())))]
fn increment(&mut self) {
    /* ([[i < self.count.len()]]_σ → $_time 1 · 1)
      ⋆ ([[i >= self.count.len()]]_σ
        → $_time(a_1^(push) · [[self.count.len()]]_σ + a_0^(push) · 1))
      ⋆ $_time(a_1^(loop) · [[self.count.len()]]_σ + a_1^(loop) [[0]] · (−1) + a_0^(loop) · 1)
    */
    let mut i = 0usize;
    /* ([[i < self.count.len()]]_σ → $_time 1 · 1)
      ⋆ ([[i >= self.count.len()]]_σ
        → $_time(a_1^(push) · [[self.count.len()]]_σ + a_0^(push) · 1))
      ⋆ $_time(a_1^(loop) · [[self.count.len()]]_σ + a_1^(loop) · (−[[i]]_σ)
        + a_0^(loop) · 1) */
    while i < self.count.len() && self.count[i] == true {
        body_invariant!(
            time_credits(O(self.count.len() - i)));
        self.count[i] = false;
        i += 1;
    }
    /* ([[i < self.count.len()]]_σ → $_time 1 · 1)
      ⋆ ([[i >= self.count.len()]]_σ
        → $_time(a_1^(push) · [[self.count.len()]]_σ + a_0^(push) · 1)) */
    if i < self.count.len() {
        // $_time 1 · 1
        self.count[i] = true;
    }
    else {
        // $_time(a_1^(push) · [[self.count.len()]]_σ + a_0^(push) · 1)
        self.count.push(true);
    }
}
```

Listing 11: Cost inference on the binary counter increment

Chapter 5

Implementation and Viper Encoding

In this chapter we describe details of our prototype implementation. In particular, this includes the Viper encoding of the concepts introduced in the previous chapters, so that we can automatically prove resource bounds. The first section introduces our encoding of credits using Viper predicates. In Section 5.2 we explain our representation of the base conversions. The next section describes how we verify asymptotic bounds in Viper. Section 5.4 presents how we deal with loops and recursive functions. In Section 5.5 we specify the exact syntax currently supported for specifications. Finally, we illustrate limitations of the current implementation state and how one can verify programs like our binary counter example by rewriting them.

The current implementation can be found on GitHub[1] and will eventually be merged into the master branch of the Prusti project[2].

5.1 Credit Predicates

As pioneered by Atkey [3], resource credits can be understood as a new permission type in addition to heap access permissions in separation logic or logics based on implicit dynamic frames [25]. While heap access permissions grant the right to read or modify the contents of a memory location, a resource credit can be used to consume some unit of the corresponding resource. The consumption of credits can be viewed as a transfer of permissions to the corresponding operation, similarly to the (temporary) transfer of access permissions to called methods. In contrast to access permissions credits cannot be recovered. The remaining credits are 'framed' across the operation, i.e. they are still available afterwards.

[1] https://github.com/ELowis/prusti-dev/tree/credit_spec_encoding
[2] https://github.com/viperproject/prusti-dev

Due to these similarities, we decided to model credits using Viper's built-in access permissions instead of using some variables keeping the count. To be able to model an unbounded (possibly rational) number of credits, we use the permission amount to represent their count. As explained in Section 2.2.1, we cannot use field permissions for this, since they only allow permission amounts between zero and one. Instead, we define predicates to represent credits and use permissions on them for the amount.

Since we want to separate the coefficients of polynomial credit amounts, as described in Section 3.4, we define one predicate for each combination of exponents that occurs in the program (and each resource type to be analyzed). That is, we define a predicate for each base function. For example,

```
predicate time_credits_1_2(_0: Int, _1: Int)
```

represents the base function for the resource runtime with exponents one and two. The predicate takes the base expressions as arguments and the permission amount is used to represent the coefficient of the base function. This means, for instance, that `acc(time_credits_1_2(m, n), 3)` stands for $\$_{time}(3 \cdot (m \cdot n^2))$ in Viper.

As defined before, negative coefficients are represented using a negative base function and a positive coefficient, which also works with Viper's non-negative permission amounts. Therefore, we can use predicates modeling these negative base functions, for example:

```
predicate time_credits_1_2_neg(_0: Int, _1: Int)
```

Note that the order of arguments for these predicates imposes a fixed order on the elements of a base function product. This means that two base functions that are mathematically the same (by commutativity) might not be equivalent in Viper, e.g.

```
time_credits_1_1(m, n) != time_credits_1_1(n, m)
```

To avoid explicitly specifying equivalence facts that model commutativity, we require the arguments of credit predicate instances to be ordered by their name, i.e. only `time_credits_1_1(m, n)` would be allowed for this example.

Using this translation to Viper access permission, (conditional) credits can be inhaled and exhaled like other specifications, that means credits in preconditions are inhaled when verifying a method body and exhaled before method calls. Cost models can be encoded by simply inserting exhales at the corresponding program points. Viper automatically checks that coefficients do not become negative by ensuring non-negative permission amounts.

However, since we allow users to introduce coefficients for negative base functions by using differences, as described in Section 3.5, we need to explicitly assert that all permission amounts for negative base functions (i.e.

negative predicates) used in the precondition are zero at the end of the method:

```
assert forall expr1: Int :: {time_credits_1_neg(expr1)}
   perm(time_credits_1_neg(expr1)) == 0/1
```

Otherwise, coefficients of positive base functions could be used without their negative counterpart from the difference, leading to a possibly negative remaining credit amount. This would violate the non-negativity postcondition needed for proving upper bounds, as defined in chapter 3.

Additionally, we need to explicitly assert the non-negativity side condition for all expressions used at the base of an exponentiation in user specifications. This includes the aforementioned differences. The check is performed at the beginning of the Viper method and for each conversion, as we will explain in the next section.

5.2 Conversion Methods

The conversions between coefficients of different base functions, as described in Section 3.7, can be modeled using Viper methods. Listing 12 presents an example of such a conversion method. The method requires $[\![\texttt{expr\_0}]\!]_\sigma = [\![\texttt{expr + const}]\!]_\sigma$ to be true in line 9. It also checks the non-negativity of the summation, to ensure that `expr_0` is non-negative as required by our representation. We can then convert from a credits in terms of the base function $(-[\![\texttt{expr}]\!]_\sigma)$ (line 10) and $[\![\texttt{const}]\!]_\sigma \cdot a$ credits in terms of the base function (-1) (line 12) to $a \cdot (-[\![\texttt{expr\_0}]\!]_\sigma)$ credits (line 14).

By only looking at the pre- and postconditions, we can see that this method represents the following instance of a conversion from Table 3.1 in the previous chapter:

$$
\begin{aligned}
&[\![\texttt{expr\_0}]\!]_\sigma = [\![\texttt{expr + const}]\!]_\sigma \\
&\Rightarrow a \cdot \left(-([\![\texttt{expr\_0}]\!]_\sigma)^1\right) = \binom{1}{0} \cdot a \cdot \left(-([\![\texttt{expr}]\!]_\sigma)^1\right) \\
&\qquad\qquad + \binom{1}{1} \cdot [\![\texttt{const}]\!]_\sigma \cdot a \cdot (-1)
\end{aligned}
$$

More generally, we encode these conversions using Viper methods as follows. All conversion methods result in a single access permission, denoted in their postcondition, which represents the following coefficient and base function:

$$a \cdot \prod_{r=0}^{l} ([\![expr_r]\!]_\sigma)^{e_r}$$

```
1  predicate time_credits_0_neg()
2
3  predicate time_credits_1_neg(expr: Int) {
4     expr >= 0 && acc(time_credits_0_neg(), expr/1)
5  }
6
7  method convert_from_sum(a: Perm, expr_0: Int, expr: Int, const:
   ↪  Int)
8  requires expr + const >= 0
9  requires expr_0 == expr + const
10 requires a >= 0/1 && acc(time_credits_1_neg(expr), a)
11 requires const * a >= 0/1
12 && acc(time_credits_0_neg(), const * a)
13
14 ensures acc(time_credits_1_neg(expr_0), a)
15 {
16    unfold acc(time_credits_1_neg(expr), a)
17    // perm(time_credits_0_neg()) = const * a + expr * a
18    // = (const + expr) * a = expr_0 * a
19    fold acc(time_credits_1_neg(expr_0), a)
20 }
```

Listing 12: Verification of a conversion method

The preconditions of a conversion method contain the equality constraint $[\![expr_s]\!]_{\sigma'} = [\![RHS]\!]_{\sigma}$ (compare with line 9), where RHS is one of the expressions described in Section 3.7. Furthermore, the preconditions require the non-negativity constraint for $[\![RHS]\!]_{\sigma}$ (or $[\![expr_s]\!]_{\sigma'}$, equivalently) as well as the access permission representation for the original coefficients of the credit amount (lines 10 to 12). For the built-in non-negativity check of permission amounts to succeed in Viper, the method also requires that all mentioned permission amounts are non-negative.

To make the methods as reusable as possible, they take the resulting coefficient (i.e. permission amount) a, all resulting $expr_r$ and all operands from the right-hand side of the conversion equality, i.e. $expr$ if $expr \neq expr_r$, $const$, etc., as arguments. Nevertheless, different operands and predicates in the pre- and postconditions still require separate methods, e.g. for each type of RHS, each index s, each list of exponents e_r and each sign of the resulting base function.

Since we require the arguments of predicate instances to be ordered lexicographically (see Section 5.1), simple equalities between two $expr$, as shown in the second row of Table 3.1, might also require a conversion. For ex-

ample, for `k = n` we might need to convert `time_credits_1_1(m, n)` to `time_credits_1_1(k, m)`. These reordering conversions will only be necessary if the order of arguments needs to change due to an assignment. Otherwise, Viper can automatically infer the equivalence of predicates on equivalent arguments.

The body of a conversion method can be omitted, but it is needed to formally verify the correctness of the conversion in Viper. Such bodies contain a series of Viper's `unfold` and `fold` operations and possibly calls to other conversion methods for reuse. To be able to unfold credit predicates, the predicates also need to have a body defined, as shown in the listing. Depending of the order defined by the predicate bodies, a conversion might need to unfold credits completely to the basic predicates for one positive or negative credit.

In the example only a single unfold is needed, since it already reaches the lowest level. Viper can then perform the necessary arithmetic, shown in the comments in line 17 and 18, to verify that the fold operation succeeds.

This conversion method would for example be called with the following arguments after the assignment `i += 1` in Listing 8:

`convert_from_sum(loop_i1(), ⟦i⟧σ′, ⟦i⟧σ, 1)`

Here, `loop_i1()` represents the abstract coefficient for `i` in the loop invariant, as described in Section 5.3.2. σ' represents the program state after the assignment and σ the state before. In Rust's MIR and hence in Prusti $[\![\texttt{i}]\!]_{\sigma'}$ and $[\![\texttt{i}]\!]_{\sigma}$ would be translated to different variables.

Especially, when the body is omitted and hence modular verification would not be used, the method can also be inlined by first exhaling the preconditions and then inhaling the postcondition.

When to trigger a conversion and with which arguments, in particular which permission amount is required, will be determined by a static analysis iterating backwards over the MIR. To determine the postconditions of conversions, the analysis collects the credits required below. Calls to conversion methods will then be inserted after every assignment whose left-hand side occurs in an inferred predicate instance. Since we do not know the value of the left-hand side above the assignment, it will be completely eliminated in the inferred credit amount by applying the conversions with the corresponding arguments in reverse. This is very similar to the credit inference for asymptotic bounds (see Section 4.3.1). Therefore, we combined them in our implementation.

If the credits required after the assignment are only needed under a certain condition P_i, the conversion call will only be performed if the condition is true:

```
if (P_i) {
   conversion_method(...)
}
```

As mentioned before, the verifier could be more powerful if we also performed conversions for equalities learned from conditions and functional specifications. However, for these it is more difficult to infer if the conversion should be performed or not. For example, for an equality $[\![k == m + n]\!]_\sigma$ we do not know if $[\![k]\!]_\sigma$ or $[\![m]\!]_\sigma$ and $[\![n]\!]_\sigma$ should be used to represent the credit amount, such that we reach a representation that matches the precondition. Therefore, a more sophisticated analysis (or additional user annotations) would be needed to perform such conversions.

5.3 Asymptotic Bounds

To verify asymptotic bounds, we implemented the cost inference as described in section 4.3.1 in Prusti as an backwards interpreter on Rust's MIR. The inferred cost is added as a precondition to the Viper encoding. We verify that the inferred credit amount is large enough to execute the function by exhaling the costs as described before. Additionally, we need to check if the inference result satisfies the asymptotic bounds that the user specified. This means the inferred cost must not contain any coefficient for a base function that does not comply with the asymptotic bounds. In Viper, we do this by asserting that the permission amount for all other base functions is zero after inhaling the inferred credits.

For example, the asymptotic bound $O([\![n]\!]_\sigma \cdot [\![m]\!]_\sigma + [\![k]\!]_\sigma)$ allows the following base functions $\overline{b}_i(\sigma)$ to occur in the inferred cost function according to the definition asymptotic dominance, given in Section 4.1:

$$\begin{aligned}
\overline{b}_1(\sigma) &= [\![n]\!]_\sigma \cdot [\![m]\!]_\sigma && \in O([\![n]\!]_\sigma \cdot [\![m]\!]_\sigma) \\
\overline{b}_2(\sigma) &= [\![n]\!]_\sigma && \in O([\![n]\!]_\sigma \cdot [\![m]\!]_\sigma) \\
\overline{b}_3(\sigma) &= [\![m]\!]_\sigma && \in O([\![n]\!]_\sigma \cdot [\![m]\!]_\sigma) \\
\overline{b}_4(\sigma) &= [\![k]\!]_\sigma && \in O([\![k]\!]_\sigma) \\
\overline{b}_5(\sigma) &= [\![1]\!]_\sigma && \in O([\![n]\!]_\sigma \cdot [\![m]\!]_\sigma) \cap O([\![k]\!]_\sigma)
\end{aligned}$$

To check that the inferred cost only contains (non-zero) coefficients for these base functions, we add the assertions shown in Listing 13 to the Viper encoding after inhaling the inferred cost.We add an assertion for every predicate that occurs in the inferred cost. Therefore, there might be more assertions like the one on `time_credits_1_2(expr1, expr2)` needed depending on the inference result.

We impose these assertions only on positive predicates, since negative predicates only make the required credit amount smaller. Therefore, the exis-

```
assert forall expr1: Int, expr2: Int ::
   {time_credits_1_2(expr1, expr2)}
   perm(time_credits_1_2(expr1, expr2)) == 0/1
assert forall expr1: Int, expr2: Int ::
   {time_credits_1_1(expr1, expr2)}
   !(expr1 == n && expr2 == m)                    // b̄1
   ==> perm(time_credits_1_1(expr1, expr2)) == 0/1
assert forall expr: Int :: {time_credits_1(expr)}
   !(expr == n) && !(expr == m) && !(expr == k)    // b̄2, b̄3, b̄4
   ==> perm(time_credits_1(expr)) == 0/1
```

Listing 13: Asymptotic bound check in Viper

tential would still succeed only with coefficients for positive base functions. Also note that inferred negative predicates cannot make the overall credit amount negative, since they are only introduced for non-negative differences in costs.

However, for differences in asymptotic bounds an additional check is required, such that $\$_R \sum_{i=0}^{k'} a_i \cdot \overline{b}_i(\sigma)$ is actually enough for the execution. The inferred coefficients must comply with the relationship that would be the result of translating $\overline{b}_i$ to our internal representation b_j, as described in Section 3.5. For each pair of coefficients for the resulting b_j, where there is no $\overline{b}_i = b_j$ in the asymptotic bound for at least one of the two, we need to perform a check that they contain the same a_i for the difference. For instance, one such assertion could look as follows for $\overline{b}_i = (\llbracket expr1 \rrbracket_\sigma - \llbracket expr2 \rrbracket_\sigma)^3$:

```
assert perm(time_credits_3(expr1)) * (3 2) ==
↪  perm(time_credits_1_2(expr1, expr2)) * (3 0)
```

Coefficients for resulting negative base functions b_i can also be smaller than required, as described before, therefore a $<=$ is sufficient instead of the $=$.

5.3.1 Conditional Asymptotic Bounds

Allowing conditional asymptotic bounds in the specifications complicates determining the allowed coefficients. Since we only deal with the user annotations syntactically in the encoder, we do not know which conditions might overlap. Overlapping conditions could lead to a larger actual asymptotic bound, since the user-defined asymptotic bounds sum up. Therefore, we need to compute the resulting asymptotic bound for each combination of the conditional asymptotic bounds. The asymptotic bound will then be checked under the combined condition.

To minimize the number of combinations, we summarize all combined con-

ditions that lead to the same asymptotic bound, by disjoining (i.e. joining with a $\vee$) them and possibly simplifying the resulting disjunction. If we order the resulting asymptotic bounds by dominance, we can additionally omit negated conditions by checking the combined conditions with an if-then-else starting with the largest asymptotic bound.

An example for this encoding is shown in Listing 14. The assertions for the asymptotic bound checks will look the same as presented before and are hence omitted.

```
#[requires(P0 ==> time_credits(O(1)))]
#[requires(P1 ==> time_credits(O(n)))]
#[requires(P2 ==> time_credits(O(m)))]
fn foo(n: u32, m: u32) { ... }
```

```
if (P1 && P2) {
    // check O(n + m)
}
elseif (P1) {       // && !P2
    // check O(n)
}
elseif (P2) {       // && !P1
    // check O(m)
}
elseif (P0) {       // && !P1 && !P2
    // check O(1)
}
```

Listing 14: Example encoding of conditional asymptotic bounds

5.3.2 Coefficient Functions

For (non-recursive) function calls, the existential over coefficients in the asymptotic bound becomes a universal quantifier, as described in Section 4.2. Therefore, we can simply use abstract Viper functions (without a body) or variables to model the possible coefficients. We decided to use functions, since they only need to be declared once and not in every method. For each called function we declare a Viper function for each possible coefficient with the only postcondition that the result is non-negative. Note that in this model all calls to the same function inside one Viper method have the cost function as an upper bound.

For the `push` function in line 19 of Listing 7, we would for example declare the following two coefficient functions in the encoding:

```
function push_0(): Perm
ensures result >= 0/1
function push_len1(): Perm
ensures result >= 0/1
```

These represent the two possible coefficients $a_0^{(push)}$ and $a_1^{(push)}$ which occur in the example in Section 4.2. The cost for the function call will then in Viper be represented by:

```
acc(time_credits_1(len), push_len1())
&& acc(time_credits(), push_0())
```

5.4 Loops and Recursion

Loops and Recursive functions are verified as described in Sections 3.3 and 4.4.

This means loops are treated like their recursive function equivalent. However, they are not translated into a separate Viper method. Therefore, additional care has to be taken. If the loop condition is true before the loop, we exhale the credits specified in the invariant. Additionally we exhale all other credits and save their permission amounts in temporary variables to inhale them again after the loop. This ensures that we only use the credits from the invariant for verifying the loop body, as we would do for a recursive function.

To verify that the loop body preserves the invariant, we verify the body starting in an abstract state where only the body invariants[3] are assumed to be true, as Prusti does already for the functional verification. This means we also inhale the credits from the body invariants. If the loop condition still holds after one abstract body execution, we exhale the body invariants to ensure their preservation.

As mentioned in Section 4.4, verifying asymptotic bounds for loops and recursive functions adds an additional complication.

We can construct an existentially quantified Viper assertion for this proof. The recurrence relation is constructed by replacing the calls to coefficient functions of the recursive function in the inference result with quantified variables.

This check is (optionally) translated to an existentially quantified assertion in our Viper encoding. However, Viper is often not able to solve the existential, even for very simple examples, as stated before. Therefore, it is currently up to the user to check the recurrence relation.

[3] loop conditions are currently not automatically assumed to hold in Prusti and therefore need to be stated as explicit body invariants

5.5 Specification Syntax

As we have seen in the examples presented before, we add a new expression to Prusti's specification syntax to represent credit amounts. In this section we explain more precisely how credit amounts can be defined and where they can occur in our current implementation.

Expressions specifying credit amounts can be used in `requires` attributes of non-pure functions (or in body invariants). The specifications may only contain a single credit expression or an implication with a credit expression on the right-hand side. Multiple credit expressions need to be defined in separate preconditions or invariants.

Credit expressions start with an identifier ending in `_credits` where the prefix defines the credit or resource type, like `time` in our examples. Currently, these types do not have any cost model attached. Therefore, they present only a syntactical separation of different credit currencies that can be analyzed at the same time. After the identifier a credit polynomial can be provided in parenthesis. This polynomial can either be defined asymptotically or concretely. Asymptotic amounts need to be specified in parenthesis after an uppercase O.

The polynomial follows the form described in Sections 3.5 and 4. Asymptotic polynomials contain a sum of one or more base functions separated by '`+`' operators. The base functions may either be the literal '`1`' or a sequence of exponentiations, separated by '`*`' operators. The exponentiations can contain a pure expression evaluating to a non-negative numerical value and an exponent after a '`^`' sign, that is a (small) positive integer literal. Currently, we only support formal arguments (and local variables) of type `u32` at the base of these exponentiations. This also means differences as described in Section 3.5 are not supported yet.

Concrete credit amounts need to additionally define a non-negative coefficient expression before every base function, multiplied with a '`*`' operator. Since the coefficient expression needs to be independent of the program state, it may only contain arithmetic operations on unsigned integer literals (fractional coefficients are not yet supported), and possible special function calls representing abstract coefficients of called functions, as described in Section 5.3.2.

5.6 Limitations and Modeling Unsupported Features

Our implementation still has a few limitations. Most of these limitations can be overcome by fully implementing what we have described in the previous chapters. Nevertheless, our prototypical implementation is already capable

of verifying the binary counter example from Section 3.2 after some rewriting (except for solving the recurrence for the asymptotic bound on the loop). The rewritten implementation of the example is provided in the two listings at the end of this section. We refer to them to demonstrate how most of the limitations can be mitigated.

Since we were focusing on asymptotic bounds, *concrete* credit annotations (like in Listing 9) cannot be verified in the current state of the implementation. Instead, concrete credit specifications will be abstracted as asymptotic bounds. However, this is a limitation that is relatively easy to lift, since concrete credit amounts were supported in an earlier stage of development for testing purposes. Also note that concrete credit amounts would currently be abstracted for calls as described in Section 4.2, which could be changed to support verifying concrete costs without the need for abstract coefficients in their specification.

We have not implemented *cost models* for different resources yet. Therefore, costs have to be always explicitly modeled by using calls to functions with credit annotations. Especially, one needs to be careful to have such a function call in every loop body and recursive function when termination should be proven. To verify our binary counter example, we need to model setters and getters for the count vector as explicit functions anyway, since Prusti does not support reasoning about `Vec` at the moment. As shown in Listing 15 at the end of this section, we use `trusted` (i.e. unverified) function stubs for setting, reading and pushing a bit. Setting a bit is assumed to require `O(1)` time credits (line 13), extending the vector using `push_bit` costs linear time in the old length (as in the original version). The getter is marked as `pure` to be able to use it in specifications. Note that pure functions cannot have credit annotations.

The current implementation does not support the encoding of *loops*. However, loops can be modeled by using a recursive function, as described in Section 3.3 and shown in Listing 16. This view is also very similar to how loops are encoded in Prusti. Recall that Viper can effectively never solve the *recurrence* relation required for the verification of asymptotic bounds on recursive functions. Therefore, even the translation to a recursive function could only be verified soundly by using concrete credit amounts (which is currently not working as mentioned before). When the existential with the recurrence for asymptotic bounds is omitted, the verification will not be sound without requiring the user to solve the recurrence manually.

As mentioned in the previous section, credit specifications may only contain a single formal argument at the *base of exponentiations*. Nevertheless, as briefly described in Section 3.5 for more complicated expressions, we can still model specifications in terms of more complicated expressions by introducing an additional argument. This ghost argument represents the result

of the expression and is updated if the value changes.

For example, the recursive function representing the loop from our binary counter has an additional argument `self_len_minus_i` in line 27 of Listing 16 to represent the difference of `counter.len` and `*i`, as specified in the highlighted precondition. This additional argument can then be used in the credit specification in line 23. The loop function will be called in line 9 with the result of the difference. The value of the variable additionally needs to be updated before the recursive call. When `i` gets incremented, `self_len_minus_i` needs to be decremented (line 30). Similarly to the loop function, the `increment` and `push_bit` functions also require an additional argument which represents the length of the current count vector, since field accesses are also not supported yet.

Also note that the current credit inference requires function calls to only take local variables (or their references) as arguments for simplicity. Therefore, we need to use temporary variables to represent the difference of `counter.len` and `i` (line 6) as well as the boolean literals `true` and `false` (lines 4 and 28).

Formal verification of base *conversions*, as mentioned in Section 12, can be activated with a configuration flag. However, we have only implemented a body for the conversion replacing a constant by an *expr* yet. Therefore, all other conversions will fail to verify. One has to trust their mathematical correctness and that we use the corresponding pre- and postconditions in our implementation. Also recall that we do not perform conversions due to equalities learned from postconditions or conditionals yet. For example the postcondition in line 27 of Listing 15 would not be used even if it were defined in terms of the `old_self_len` argument. Instead, another function call with linear cost in the length of the count after `push_bit` in Listing 16 would require updating `self_len` with a manual assignment.

```rust
struct Counter {
    b: bool,    // needed, since need state to change even if
    ↪ len stays the same (different get result)
    len: u32,
}

#[pure]
#[trusted]
fn get_bit(counter: &Counter, i: u32) -> bool {
    return counter.b;       // dummy implementation
}

#[trusted]
#[requires(time_credits(O(1)))]
#[requires(i < counter.len)]
#[ensures(counter.len == old(counter.len))]
#[ensures(get_bit(counter, i) == val)]
// everything else unchanged:
#[ensures(forall(|j: u32| (j < counter.len && j != i)
    ==> get_bit(counter, j) == old(get_bit(counter, j))))]
fn set_bit(counter: &mut Counter, i: u32, val: bool) {
    ()      // dummy implementation
}

#[trusted]
#[requires(old_self_len == counter.len)]
#[requires(time_credits(O(old_self_len^1)))]
#[ensures(counter.len == old(counter.len) + 1)]
#[ensures(get_bit(counter, counter.len - 1) == val)]
// everything else unchanged:
#[ensures(forall(|j: u32| j < old(counter.len)
    ==> get_bit(counter, j) == old(get_bit(counter, j))))]
fn push_bit(counter: &mut Counter, val: bool, old_self_len:
↪ u32) {
    counter.len = counter.len + 1; // dummy implementation
}
```

Listing 15: Abstracting the Vec field of the binary counter

```rust
#[requires(self_len == counter.len)]
#[requires(time_credits(O(self_len^1)))]
fn increment(counter: &mut Counter, self_len: u32) {
    let temp_true = true;
    let mut i = 0u32;
    let len_minus_i = self_len - i;
    if i < counter.len {
        if get_bit(counter, i) == true {
            loop_fn(counter, &mut i, len_minus_i);
        }
    }

    if i < counter.len {
        set_bit(counter, i, temp_true);        // O(1)
    }
    else {
        push_bit(counter, temp_true, self_len);// O(self_len)
    }
}

// loop translation
#[requires(self_len_minus_i == counter.len - *i)]
#[requires(time_credits(O(self_len_minus_i^1)))]
#[requires(*i < counter.len && get_bit(counter, *i) == true)]
#[ensures(counter.len == old(counter.len))]
#[ensures(*i >= counter.len || get_bit(counter, *i) != true)]
fn loop_fn(counter: &mut Counter, i: &mut u32, mut
↪   self_len_minus_i: u32) {
    let temp_false = false;
    set_bit(counter, *i, temp_false);          // O(1)
    *i += 1; self_len_minus_i -= 1;

    if *i < counter.len {
        if get_bit(counter, *i) == true {
            loop_fn(counter, i, self_len_minus_i);
        }
    }
}
```

Listing 16: Rewritten binary counter increment implementation

Chapter 6

Evaluation

Since our implementation is still incomplete and not optimized, we are only doing a qualitative evaluation here. This means we present a few examples that illustrate the features of our proposed extension to Prusti. If not stated otherwise, all examples verify using the current implementation except for solving the recurrence relation. Section 6.1 presents different implementations for the computation of the n-th Fibonacci number. Some more abstract examples that include common algorithmic patterns or a certain degree of complexity are shown in Section 6.2.

Throughout this chapter we will use the functions presented in Listing 17 to model the use of a certain asymptotic amount of credits. The credit type `r_credits` is just a placeholder for the types that will be used in the examples. Calls to these functions can be seen as a way to specify a cost model. As mentioned in Section 5.6, our implementation does not have built-in cost models yet.

```
1  #[trusted]
2  #[requires(r_credits(O(1)))]
3  fn constant() {}
4
5  #[trusted]
6  #[requires(r_credits(O(n^1)))]
7  fn linear(n: u32) {}
8
9  #[trusted]
10 #[requires(time_credits(O(n^1*m^1)))]
11 fn multi(n: u32, m: u32) {}
```

Listing 17: Functions to model costs

6.1 Fibonacci Numbers

In this section we present the asymptotic runtime verification of different algorithms to compute the n-th Fibonacci number. The Fibonacci numbers form a mathematical sequence that is defined by the following recurrence relation:

$$F_1 := 1, \quad F_2 := 1 \tag{6.1}$$

$$F_n := F_{n-1} + F_{n-2}, \qquad \text{for } n > 2 \tag{6.2}$$

This means every number in this sequence is the sum of its two predecessors, or one if there are less than two predecessors. There is also an alternative definition starting at zero which leads to equivalent algorithms.

There are two principle approaches to compute the n-th Fibonacci number. One computes the number bottom-up starting at F_1 and F_2, always remembering the last two numbers to sum them. The other approach follows the recurrence relation for F_n top-down until it reaches F_1 and F_2.

6.1.1 Bottom-up Algorithm

We will start with an algorithm using the bottom-up approach. Usually, this algorithm is formulated as a loop, but as mentioned in the Section 5.6 our implementation does not support loops yet. Therefore, we use the recursive formulation shown in Listing 18 on the next page. Given the two previous Fibonacci numbers, `fib_bottom_up` computes the next one and calls itself recursively until the index `n` is reached.

We assume that each invocation of `fib_bottom_up` consumes a constant amount of time credits (line 13) by itself before performing the recursive call. As there are no built-in cost models, omitting the function call with constant cost would allow us to prove that no credits are needed for the whole recursion, which is counter-intuitive when thinking about runtime. With the constant cost per recursive call, we need to instead specify linear bounds on the entry function and the recursive function.

Recall that bounds on recursive functions cannot be automatically verified at the moment. Instead, our cost inference generates the following recurrence relation to be solved for a sound proof:

$$\forall c_0 \in \mathbb{Q}_0^+.\ \exists a_0, a_1 \in \mathbb{Q}_0^+.\ a_1 \geq a_1 \wedge a_0 \geq a_0 + c_0 - a_1$$

Here, c_0 represents the abstract coefficient for the cost of the `constant` function, i.e. it costs $c_0 \cdot 1$, a_0 and a_1 represent the coefficients for the cost function of `fib_bottom_up`. The recurrence is easily solvable by any a_1 satisfying $a_1 \geq c_0$ (and $a_0 \geq c_0$ for the terminating branch). Therefore, the verification of the asymptotic bound is sound.

```rust
#[requires(n > 0)]  // 1-based index
#[requires(time_credits(O(n^1)))]
fn fib(n: u32) -> u32 {
    if n > 2 {
        let fib_1 = 1;
        let fib_2 = 1;
        fib_bottom_up(n-2, fib_2, fib_1)
    } else { 1 }
}
#[requires(n_minus_i > 0)]
#[requires(time_credits(O(n_minus_i^1)))]
fn fib_bottom_up(mut n_minus_i: u32, fib_i: u32, fib_i_minus_1:
↪ u32) -> u32 {
    constant();
    // compute (i+1)-th Fibonacci number
    let fib_i_plus_1 = fib_i + fib_i_minus_1;
    n_minus_i -= 1;
    if n_minus_i > 0 {
        fib_bottom_up(n_minus_i, fib_i_plus_1, fib_i)
    } else {  // i == n
        fib_i_plus_1
    }
}
```

Listing 18: Algorithm to compute n-th Fibonacci number *bottom-up*

6.1.2 Top-down Algorithm

The top-down algorithm is better suited for a recursive implementation, since it contains two recursive calls to compute the two previous Fibonacci numbers. A Rust implementation is shown in Listing 19 on the next page. Performing the recursive calls naively will lead to the re-computation of many numbers, since the computation of the $(n-1)$-th Fibonacci number includes computing the $(n-2)$-th. The resulting call structure forms a binary tree of depth n. Therefore, the runtime grows exponentially, i.e. $O(2^n)$.

We cannot specify and verify such exponential bounds with our current approach. However, we can disprove that the recursive function can run in linear time by using the inferred recurrence relation:

$$\forall c_0 \in \mathbb{Q}_0^+.\ \exists a_0, a_1 \in \mathbb{Q}_0^+.\ a_1 \geq a_1 + a_1 \wedge a_0 \geq a_0 + a_0 + c_0 - a_1 - 2 \cdot a_1$$

Since we have two recursive calls, a_0 and a_1 occur twice on the right-hand side of their respective inequalities. The first inequality is only solvable for $a_1 = 0$, which means we cannot compensate for the parts added to a_0 on the

```
#[requires(n > 0)]  // 1-based index
#[requires(time_credits(O(n^1)))]  // incorrect
fn fib_top_down(n: u32) -> u32 {
    constant();
    if n > 2 {
        fib_top_down(n-1) + fib_top_down(n-2)
    } else { 1 }
}
```

Listing 19: Algorithm to compute n-th Fibonacci number *top-down*

right-hand side of the second inequality. Therefore, the recurrence relation as a whole is not solvable for a linear bound, as expected.

Keep in mind that the recurrence relation cannot be automatically proved or disproved in the current implementation. Therefore, without encoding the failing existential, our verifier would actually be able to prove the incorrect linear bound, since it assumes that the recursive calls are just calls to any function with linear runtime. Even worse, it would also be able to verify a linear bound if the function never terminated. For example, the argument to the first recursive call could just be set to `n` by mistake, leading to an infinite recursion.

6.1.3 Top-down Algorithm with Memoization

The redundant computation of the top-down algorithm can be avoided by using memoization. Memoization is a common pattern to solve this problem. It caches already computed results and returns them if the function is called again with the same argument. For our example, this means each Fibonacci number F_n is computed at most once. This leads to a linear runtime bound in general and even to a constant runtime if the value is already stored.

As shown in Listing 20, this algorithm represents a use case for our conditional bounds. The user can specify different bounds depending on whether the value is stored in the cache or not, as we do in lines 3 and 4. The interface of the cache is presented in the separate Listing 21. For simplicity, we also assume that the base values F_1 and F_2 are already stored in the cache (line 2).

The postcondition of `fib_top_down_m` declares that all values with a smaller or equal index are stored in the cache, since the algorithm reaches all of them in recursive calls. Therefore, we could deduce from the postcondition of the first recursive call that the value for the second recursive call with index `n-2` is already stored. Hence, the second recursive call has constant

```
#[requires(n > 0)]  // 1-based index
#[requires(is_stored(1) && is_stored(2))]
#[requires(is_stored(n) ==> time_credits(O(1)))]
#[requires(!is_stored(n) ==> time_credits(O(n^1)))]
#[ensures(forall(|i: u32| i > 0 && i <= n ==> is_stored(i)))]
fn fib_top_down_m(n: u32) -> u32 {
    constant();
    if is_stored(n) { load(n) }
    else {
        let res = fib_top_down_m(n-1) + fib_top_down_m(n-2);
        store(n, res);
        res
    }
}
```

Listing 20: Algorithm to compute n-th Fibonacci number *top-down* with *memoization*

runtime cost and we can prove the linear bound by solving the following recurrence relation:

$$\forall c_0 \in \mathbb{Q}_0^+.\ \exists a_0, a_1, a_0^{(stored)} \in \mathbb{Q}_0^+.\ a_1 \geq a_1 \wedge a_0 \geq a_0 + a_0^{(stored)} + c_0 - a_1$$

In this recurrence $a_0^{(stored)}$ represents the abstract coefficient of the constant bound for cached values, which might be different from the coefficient a_0 for the (disjoint) uncached case. This recurrence is solvable for $a_1 \geq a_0^{(stored)} + c_0$.

Unfortunately, our current inference for asymptotic bounds is not powerful enough to be able to infer that `is_stored(n)` is always true for the second call. Instead it will propagate the conditional bounds to the beginning of

```
#[pure]
#[trusted]
fn is_stored(n: u32) -> bool { true } // dummy implementation
#[trusted]
#[ensures(is_stored(n))]
fn store(n: u32, val: u32) {}
#[trusted]
#[requires(is_stored(n))]
#[requires(time_credits(O(1)))]
#[ensures(forall(|i: u32| i > 0 && i <= n ==> is_stored(i)))]
fn load(n: u32) -> u32 { return n; } // dummy implementation
```

Listing 21: Modeling the cache for memoization

the Viper method, where Viper cannot infer that either. Therefore, we could only prove a linear bound with concrete credits which do not require the use of the cost inference. For asymptotic bounds, we can only workaround this by calling `load(n-2)` instead of `fib_top_down_m(n-2)`. This will be a sound replacement, since the precondition of `load` checks that the value is actually stored, which Viper can then infer from the first call. However, this is not satisfying, since it requires the user to be aware that the value for the second call will be cached. Future work might refine the cost inference to be able to deal with these cases.

6.2 Abstract Examples

This section presents the verification of two common loop patterns as well as stress tests of the base conversions and conditional bound support. All examples are kept abstract by using `r_credits` which can represent any resource of interest.

6.2.1 Multivariate Nested Loop

Listing 22 presents a common example for multivariate bounds. It contains the recursive implementation of a nested loop. The outer loop (function `nested_loop`) performs m iterations, in which the body of the inner loop is executed $\lfloor n/2 \rfloor$ times. Both loops pay a constant amount of credits for each loop iteration.

For the inner loop we have to check a recurrence relation that is very similar to the one for the bottom-up algorithm for the n-th Fibonacci number:

$$\forall c_0 \in \mathbb{Q}_0^+. \exists a_0, a_1 \in \mathbb{Q}_0^+. a_1 \geq a_1 \wedge a_0 \geq a_0 + c_0 - 2 \cdot a_1$$

```
1  #[requires(r_credits(O(m^1*n^1)))]
2  fn nested_loop(m: u32, n: u32) -> u32 {
3      constant();
4      inner_loop(n);
5      if m > 0 { nested_loop(m - 1, n) }
6      m
7  }
8  #[requires(r_credits(O(n^1)))]
9  fn inner_loop(n: u32) {
10     constant();
11     if n > 1 { inner_loop(n - 2) }
12 }
```

Listing 22: Nested loop with linear bound client

This is the typical recurrence relation for a linear for-loop, except that we only perform an iteration for every second n here. Therefore, $2 \cdot a_1$ needs to be greater than or equal to c_0.

The recurrence relation for the outer loop is similar. It only contains more coefficients for the multivariate bound:

$$\begin{aligned}
&\forall c_0, a_0^{(inner)}, a_1^{(inner)} \in \mathbb{Q}_0^+ .\ \exists a_0, a_{\mathtt{n1}}, a_{\mathtt{m1}}, a_{\mathtt{m1n1}} \in \mathbb{Q}_0^+ .\\
&\quad a_{\mathtt{m1n1}} \geq a_{\mathtt{m1n1}} \wedge a_{\mathtt{m1}} \geq a_{\mathtt{m1}}\\
&\wedge\ a_{\mathtt{n1}} \geq a_{\mathtt{n1}} + a_1^{(inner)} - a_{\mathtt{m1n1}}\\
&\wedge\ a_0 \geq a_0 + a_0^{(inner)} + c_0 - a_{\mathtt{m1}}
\end{aligned}$$

Here, $a_0^{(inner)}$ and $a_1^{(inner)}$ denote the constant and linear coefficient of the inner loop cost, respectively. The coefficients for the outer loop have a representation of their corresponding base function as a subscript. This recurrence condition is satisfiable for any coefficients, such that $a_{\mathtt{m1n1}} \geq a_1^{(inner)}$ and $a_{\mathtt{m1}} \geq a_0^{(inner)} + c_0$.

Listing 23 presents a client method that calls the nested loop with n being zero. As presented in Section 3.2 of [13], a flawed proof of an asymptotic bound on this example could just substitute zero for n in the asymptotic bound of the nested loop and deduce that the call costs $O(0)$ credits. Since we represent the credits needed for calls as expanded polynomials with abstract coefficients, this cannot happen in our approach. In this case, the cost of a call to `nested_loop` is represented by

$$a_{\mathtt{m1n1}} \cdot nm + a_{\mathtt{m1}} \cdot m + a_{\mathtt{n1}} \cdot n + a_0 \cdot 1$$

If we substitute zero for n in a conversion, we will still have the following cost remaining:

$$a_{\mathtt{m1}} \cdot m + a_0 \cdot 1$$

Therefore, a constant bound specification would fail in the asymptotic bound check, defined in Section 5.3, since the inferred cost is linear in m.

```
1  #[requires(r_credits(O(1)))]     // fails, O(m^1) succeeds
2  fn client(m: u32) {
3      constant();
4      let zero = 0;
5      nested_loop(m, zero);
6  }
```

Listing 23: Nested loop with linear bound client

6.2.2 Decreasing Recursive Cost

Another typical algorithmic pattern is a loop whose iteration cost depends on the loop variable and hence decreases with it. A recursive implementation of this pattern is shown in Listing 24. Every recursive function invocation costs a linear amount of credits in terms of the argument n which is decreased before every recursive call. The upper bound on the function is quadratic, since $\sum_{i=1}^{n} i = (n^2 + n)/2$.

The inferred recurrence relation for this function will look as follows:

$$\begin{aligned} &\forall l_0, l_1 \in \mathbb{Q}_0^+ . \exists a_0, a_1, a_2 \in \mathbb{Q}_0^+ . \\ &\quad a_2 \geq a_2 \\ &\wedge\ a_1 \geq a_1 + l_1 - 2 \cdot a_2 \\ &\wedge\ a_0 \geq a_0 + a_2 + l_0 - a_1 \end{aligned}$$

Here, l_1 and l_0 represent the cost function coefficients for the call to `linear`. The $-2 \cdot a_2$ and $+a_2$ are coming from the binomial expansion of $a_2 \cdot (n-1)^2$. This recurrence is solvable for $2 \cdot a_2 \geq l_1$ and $a_1 \geq a_2 + l_0$.

```
1 #[requires(n > 0)]
2 #[requires(r_credits(O(n^2)))]
3 fn decreasing(n: u32) {
4     linear(n); // decreases with every iteration
5     if n > 0 { decreasing(n-1) }
6 }
```

Listing 24: Quadratic function with decreasing linear cost

6.2.3 Stress Tests

Listing 25 presents a chain of arithmetic operations. The cost inference is able to determine the right credit amounts by applying base conversions, such that the verification succeeds. The variable t is only built-up from constant parts and hence $r \in O(n)$. The second argument of the call to `bar` only contains the product of n and m. Therefore, we can prove the overall asymptotic bound $O(n^3 m^1) = O(n^2 (m \cdot n)^1)$.

In Listing 26 we included a Rust program with multiple conditionals. We defined conditional credit amounts in the precondition of this function matching the structure of the program. Notice that the condition $k < (n + m)$ overlaps with the following three conditions and hence their asymptotic bounds get summed. For example the program path ending in line 13, has a total credit bound of $O(n + m)$.

Both examples shown here verify without any manual help by solving a recurrence, since they do not include recursive functions.

```
#[requires(cf_time_credits(O(n^3*m^1)))]
fn conversions(n: u32, m: u32) {
    let l = 3;
    let j = (l + 5) * 6;
    let t = j / 3;
    let r = t - 3 + n;
    let s = m * n;
    bar(r, s);
}
#[trusted]
#[requires(cf_time_credits(O(n^2*m^1)))]
fn bar(n: u32, m: u32) { }
```

Listing 25: Test of various base conversions

```
#[requires(k < (n + m) ==> time_credits(O(n^1)))]
#[requires(k < (n + m) && k < (n-3) && k == 3 ==>
↪  time_credits(O(n^1)))]
#[requires(k < (n + m) && k < (n-3) && k == 6 ==>
↪  time_credits(O(n^1*m^1)))]
#[requires(k < (n + m) && k < (n-3) && k != 3 && k != 6 ==>
↪  time_credits(O(m^1)))]
#[requires(!(k < (n + m)) ==> time_credits(O(1)))]
fn conditional(n: u32, m: u32, k: u32) {
    if k < (n + m) {
        linear(n);
        if k < (n - 3) {
            match k {
                3 => linear(n),
                6 => multi(n, m),
                _ => linear(m),
            }
        }
    } else { constant() }
}
```

Listing 26: Test with multiple conditions

Chapter 7

Conclusion

In this thesis we have shown how abstract credits can be used to automatically reason about the resource usage of programs in the Viper verification infrastructure. Representing credit amounts by polynomial coefficients, enables us to prove resource bounds in terms of multivariate polynomials while avoiding complex super-linear constraints. Additionally, this representation simplifies the check of asymptotic bounds. On the other hand, reasoning in terms of coefficients requires the use of explicit conversions between them. Furthermore, our representation causes over-approximations when checking the non-negativity of costs and comparing different credit amounts.

These over-approximations are unproblematic for the proof of asymptotic bounds. However, reasoning about asymptotic behavior introduces new complexities. It requires the verifier to prove that there exists a concrete credit amount that provides enough credits for the execution and obeys the asymptotic bound. For programs without loops and recursion, we can infer this concrete credit amount. Loops and recursion additionally require checking a recurrence relation on the inferred cost, which our current implementation cannot do automatically.

We model credit amounts using Viper's built-in access permissions. Since these permissions represent a similar notion of rights, we can exploit the integrated reasoning on them to reduce the amount of additional encoding needed for resource credits. This also enables us to easily add or subtract credits conditionally, leading to more precise inference results and allowing users to specify conditional resource bounds.

7.1 Future Work

In addition to lifting the limitations of the implementation mentioned in Section 5.6, the work presented in this thesis can be improved or extended in the following directions.

Asymptotic Bounds

The cost inference which is needed to prove asymptotic bounds introduces some redundancy, since it is similar to the verification Viper performs afterwards. Therefore, it might be worth considering if the reasoning about asymptotic bounds could be built into the Viper verifier itself, instead of building on top of it. This also includes the proof of recurrences.

Alternatively, one could use concrete credit amounts together with sufficiently abstract cost models, like the one provided in [13], to prove asymptotic bounds. To further simplify the specification of concrete coefficients needed in this case, our coefficient-based reasoning could be made more precise. For example, we could allow transfers between coefficients of negative base functions and their positive counter-part. This would effectively model possibly negative coefficients by the difference of a positive and a negative coefficient.

Verifying Recurrence Relations

Reasonably simple recurrence relations for recursive functions should be possible to verify automatically. As mentioned before, one could apply a similar method as in [29] based on the master theorem. Using an LP solver as in the AARA-based tools (e.g. in [7]) could also be feasible. Another alternative could be a clever heuristics-based pattern matching, since the typical recurrence relations are very similar, as we have seen in the evaluation.

Regaining Credits

For some resources it might be useful to have operations with negative costs or similarly, operations that return credits in their postcondition. This is for example the case for a verification of the currently allocated memory. Credits could be used to pay for an allocation and could be returned when the memory is deallocated.

New Base Functions

It might be worth reconsidering the base functions of our coefficient-based representation. First and foremost, this means adding more base functions to enable the verification of more complex bounds. Additionally, there might be better base functions for polynomials that simplify the conversions

needed, like the binomial base functions used in RAML [19] for inductive data types. Especially, reducing the number of additional terms introduced by conversions for additions would be useful.

Amortized Analysis

As mentioned in Section 3.8, we have not implemented our proposed method for amortized analysis yet. This includes defining a new specification syntax for type invariants.

Stronger Inference

Our cost and base conversion inference cannot work on arbitrary programs, since inferring resource bounds that would prove termination is undecidable in general. However, the inference can still be improved. We have seen an example in Section 6.1.3 where the inference fails, since it does not take postconditions of previously called functions into account. Functional specifications (and conditionals) could also be used for the conversion inference in addition to assignments.

Cost Models

We have not defined any cost models yet and instead rely on the user to define their own cost model by calling annotated functions. It would be useful for users if the verifier provided at least some pre-defined cost models for different resources. These could include runtime models at different abstraction levels and cost models for memory usage.

Appendix A

Example Base Conversion

$$
\begin{aligned}
a_0 \cdot \overline{b}_0(\sigma) &= a_0 \cdot \left([\![expr_0^{(0)}]\!]_\sigma - [\![expr_0^{(1)}]\!]_\sigma \right)^2 \cdot \left([\![expr_1^{(0)}]\!]_\sigma - [\![expr_1^{(1)}]\!]_\sigma \right)^2 \\
&= \quad \left(\binom{2}{0} \cdot \binom{2}{0} \cdot a_i \right) \cdot ([\![expr_0^{(0)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^2 \\
&\quad + \left(\binom{2}{1} \cdot \binom{2}{0} \cdot a_i \right) \cdot (-([\![expr_0^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_0^{(1)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^2) \\
&\quad + \left(\binom{2}{2} \cdot \binom{2}{0} \cdot a_i \right) \cdot ([\![expr_0^{(1)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^2 \\
&\quad + \left(\binom{2}{0} \cdot \binom{2}{1} \cdot a_i \right) \cdot (-([\![expr_0^{(0)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^1) \\
&\quad + \left(\binom{2}{1} \cdot \binom{2}{1} \cdot a_i \right) \cdot ([\![expr_0^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_0^{(1)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^1 \\
&\quad + \left(\binom{2}{2} \cdot \binom{2}{1} \cdot a_i \right) \cdot (-([\![expr_0^{(1)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^1) \\
&\quad + \left(\binom{2}{0} \cdot \binom{2}{2} \cdot a_i \right) \cdot ([\![expr_0^{(0)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^2 \\
&\quad + \left(\binom{2}{1} \cdot \binom{2}{2} \cdot a_i \right) \cdot (-([\![expr_0^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_0^{(1)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^2) \\
&\quad + \left(\binom{2}{2} \cdot \binom{2}{2} \cdot a_i \right) \cdot ([\![expr_0^{(1)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^2 \\
&= \quad a_i \qquad\qquad \cdot ([\![expr_0^{(0)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^2 \\
&\quad + (2 \cdot a_i) \qquad \cdot (-([\![expr_0^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_0^{(1)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^2) \\
&\quad + a_i \qquad\qquad \cdot ([\![expr_0^{(1)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^2 \\
&\quad + (2 \cdot a_i) \qquad \cdot (-([\![expr_0^{(0)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^1) \\
&\quad + (4 \cdot a_i) \qquad \cdot ([\![expr_0^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_0^{(1)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^1 \\
&\quad + (2 \cdot a_i) \qquad \cdot (-([\![expr_0^{(1)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^1) \\
&\quad + a_i \qquad\qquad \cdot ([\![expr_0^{(0)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^2 \\
&\quad + (2 \cdot a_i) \qquad \cdot (-([\![expr_0^{(0)}]\!]_\sigma)^1 \cdot ([\![expr_0^{(1)}]\!]_\sigma)^1 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^2) \\
&\quad + a_i \qquad\qquad \cdot ([\![expr_0^{(1)}]\!]_\sigma)^2 \cdot ([\![expr_1^{(1)}]\!]_\sigma)^2
\end{aligned}
$$

Bibliography

[1] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, May 1998. doi:10.1023/A:1018373005182.

[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. doi:10.1145/3360573.

[3] Robert Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 7(2), June 2011. doi:10.2168/lmcs-7(2:17)2011.

[4] Martin Avanzini and Georg Moser. A Combination Framework for Complexity. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 55–70, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.RTA.2013.55.

[5] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 12671279, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2660267.2660283.

[6] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for c programs. *SIGPLAN Not.*, 49(6):270281, June 2014. doi:10.1145/2666356.2594301.

[7] Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. Automated resource analysis with Coq proof objects. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 64–85, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-63390-9_4.

[8] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. *SIGPLAN Not.*, 50(6):467478, June 2015. doi:10.1145/2813885.2737955.

[9] A. Charguéraud and F. Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, 62(3):331–365, 2019.

[10] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. *SIGPLAN Not.*, 47(10):831850, October 2012. doi:10.1145/2398857.2384676.

[11] Manuel Eberl. Landau symbols. *Archive of Formal Proofs*, July 2015. https://isa-afp.org/entries/Landau_Symbols.html, Formal proof development.

[12] Manuel Eberl. Proving divide and conquer complexities in isabelle/hol. *Journal of Automated Reasoning*, 58(4):483–508, Apr 2017. doi:10.1007/s10817-016-9378-0.

[13] Armaël Guéneau. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs*. PhD thesis, Université de Paris, 2019.

[14] M.P.L. Haslbeck and T. Nipkow. Hoare logics for time bounds: A study in meta theory. *Lecture Notes in Computer Science*, 10805 LNCS:155–171, 2018. doi:10.1007/978-3-319-89960-2_9.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576580, October 1969. doi:10.1145/363235.363259.

[16] J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. In *ESOP*, pages 287–306, 2010. doi:10.1007/978-3-642-11957-6_16.

[17] Jan Hoffmann. *Types with potential: polynomial resource bounds via automatic amortized analysis*. PhD thesis, Ludwig-Maximilians-Universität München, Oktober 2011. doi:10.5282/edoc.13955.

[18] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3), November 2012. doi:10.1145/2362389.2362393.

[19] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 781–786, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-31424-7_64.

[20] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant aes-gcm. CHES '09, page 117, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-04138-9_1.

[21] N. D. Matsakis and F. S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103104, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2663171.2663188.

[22] 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. doi:10.1007/978-3-662-49122-5_2.

[23] John Regehr, Alastair David Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embed. Comput. Syst.*, 4:751–778, 2005. doi:10.1145/1113830.1113833.

[24] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 745–761, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-08867-9_50.

[25] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, pages 148–172, 2009. doi:10.1007/978-3-642-03013-0_8.

[26] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985. doi:10.1137/0606031.

[27] Peng Wang, Di Wang, and Adam Chlipala. Timl: A functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133903.

[28] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter

Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problemoverview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008. doi:10.1145/1347375.1347389.

[29] Bohua Zhan and Maximilian P. L. Haslbeck. Verifying asymptotic time complexity of imperative programs in Isabelle. *CoRR*, abs/1802.01336, 2018. URL: http://arxiv.org/abs/1802.01336, arXiv:1802.01336.

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):

Reasoning about Complexities 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):**
Engel	Lowis Anton

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)**
Frankfurt am Main, 09.10.2021	

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