UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Extending Support for Axiomatic Data Types in VerCors

Ömer Faruk Oğuzhan Şakar
Student Number: 1560158
o.f.o.sakar@student.utwente.nl

Master's Thesis
April 2020

Supervisors:
prof. dr. M. Huisman
dr. R. E. Monti
Exam Committee:
prof. dr. M. Huisman
dr. M. B. Van Riemsdijk

Formal Methods and Tools Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

VerCors is a static verifier of concurrent/parallel programs developed at the University of Twente. The software that is verified with VerCors (and similar tools) use common data types such as lists or sets. The behavior of these data types is modeled in VerCors using axiomatic data types (ADTs). VerCors currently supports axiomatic data types such as sequences/lists, sets, and bags. To extend the support for ADTs, a list of features to add to VerCors was compiled by using input from end-users. An implementation-level view of VerCors is given with general approaches to implementing a feature in VerCors. For each feature in this list a definition is given, its encoding into Viper (the back end of VerCors) is discussed and its implementation using the general approaches is explained.

Contents

Chapter 1

Introduction

The importance of software verification is stressed in numerous papers and it has been the motivation for a wide variety of research. Different techniques have been researched and realized in different software verification tools. All these different techniques combined give the user of these tools the ability to verify properties on their software projects such as functional correctness (e.g Dafny [1]), data-race freedom (e.g. GPUVerify [2]), loop termination (e.g. Frama-C [3]) and memory management (e.g. VerCors [4]). Instead of focusing on a specific technique of a tool or a case study, let us focus more on the tools themselves and how they are used.

In general, it could be said about any tool (be it software or hardware) that the initial goals for that tool are relatively simple and robust. The tool meets the users' requirements and can solve most problems the user has. More complex features can be realized using existing features and so self-defined functionality might come into existence. However there is one large caveat, the more complex the problem at hand becomes, the more experienced hands are needed to utilize these simple tools. Every user needs some experience to work with any tool, however, the threshold of becoming an expert user of the tool can be too high.

With time, tools expand on their initial feature set in the form of major/minor releases. The new features contribute to the tools in a variety of ways. Expressiveness of code is one of these ways and can come in many forms, for instance in the form of entirely new features or simpler or more complex user interfaces of existing features. Expressiveness gives the user choice in tools to tackle their problem with and if done correctly the tools become more refined to aid them on their journey.

In the context of software verification, these tools are software projects by which software is verified. There are a plethora of these tools of which Dafny, VeriFast [5] and Boogie [6] are just some popular examples. These tools (in general) are most often used by experts referred to as verification engineers. The verification engineers have to come from somewhere, or phrased differently, an amateur user will have to start somewhere to become an expert. To both lower the threshold for new users and give expert users more refined tools, a software verification tool needs to develop with a plethora of users in mind.

For this thesis, we take a look at the verification tool VerCors, a tool for the verification of concurrent and parallel software written in programming languages such as Java, C and its own language PVL. The specification language of VerCors is based on JML with permission-based separation logic for concurrent programs [7].

VerCors uses a layered approach to its implementation of functionality by processing the input file and encoding/translating it in an intermediate language COL. COL is then encoded/translated into the back end Viper for verification. Viper [8] is a toolchain used to verify concurrent programs (see Chapter 2 for more detail). VerCors supports a variety of features in the form of techniques, data types and flow constructs. This thesis focuses on a specific set of these features namely the axiomatic data types supported by VerCors as expressed in PVL.

Axiomatic data types (*ADTs*) are data structures that are defined using axioms. ADTs express the behavior of a data structure instead of its exact implementation. From a user-perspective, these ADTs behave the same as data structures used in programming languages such as Java's lists and Python's sets.

ADTs are used by different software verifiers (e.g. Dafny, Viper, VeriFast) and are interesting in this context because they abstract from the concrete implementation of a data structure. For example, a sequence of train stops can be modeled as a Java list (i.e. a data structure). This Java list has an implementation with concrete steps on how it behaves. When proving a property of that list (e.g. the trains stops when it should) the implementation of this list is not necessarily important, it is the behavior of the list that is important. Thus the behavior of a list is expressed axiomatically and the resulting ADT is used to specify properties on the Java list.

To make ADTs more useful, basic operations are natively supported, giving the user a robust base to work with (e.g. such as the concatenation of two sequences or the difference between two sets). Although users can define their own features on ADTs using these basic operations, there are some shortcomings to the support of ADTs in VerCors:

- Only a handful of ADTs are supported, namely sequences, sets and bags. In addition to these ADTs, software projects also use more frequently-used data structures such as maps and deques. These ADTs can be supported to verify a wider range of problems.
- The set of basic operations on the supported ADTs is limited. For example, the operations on a sequence that are currently supported are the comparison of sequences, concatenation of two sequences, getting the head/tail of a sequence and retrieving the element at some index. Using these basic operations, other operations can be defined by the users, however compared to Java's or Python's operations on lists, this set of basic operations can be expanded upon.
- When defining some general-purpose functionality, that functionality is defined for a specific type. For example, if we have some function for appending a single value to a sequence of integers, we have to redefine it for a sequence of fractions. This results in code duplication (in ghost code or actual functions). This can be solved by allowing abstract/generics functions.

These shortcomings are solved in this thesis by focusing on two main questions:

RQ1 What ADTs or functionality on ADTs is desired from VerCors/PVL?

RQ2 How can the architecture of VerCors and the back end Viper support the functionality of **RQ1**?

The following approach is taken to answer these questions. Different resources are consulted and a survey is held to get feedback from users. **RQ1** is answered by extracting a list of features from these sources (see Chapter 3). With this concrete list, **RQ2** is made concrete for each feature in the list (see Chapter 4). These individual concrete questions are then answered by looking at the architecture of VerCors (see Chapter 5) and discussing how this feature can be implemented using the architecture of VerCors (see Chapter 6 and on).

Normally, the resulting features would be evaluated by performing a user study to determine whether the mentioned shortcomings were solved. Unfortunately, a user study was not performed due to a lack of participants with different levels of experience. It would be optimal to have both experts that have used VerCors and participants that have no (or little) experience with VerCors. The first group of participants would mainly consist of VerCors developers. The second group is more difficult to find. It was considered to have students participate after their own course on software verification. This idea could not be realized since the time allocated for this thesis did not overlap enough with the timeslot students with basic knowledge of software verification were available.

The main challenge addressed by this thesis is **RQ2**. After determining a list of features to implement, this question can be made concrete for each feature, however the implementation of said feature or its encoding into Viper is not always trivial. The more complex a specific feature gets, the less straightforward it gets to implement it into VerCors (e.g. the COL encoding) and encode it into Viper (e.g. sound axioms over an ADT). The complexity of implementing a feature in VerCors is tackled by discussing the architecture of VerCors at an implementation-level and formalizing general approaches to implement a feature (see Chapters 5 and 6). The complexity of the encoding into Viper is specific to each feature and is discussed in detail in Chapters 8, 9 and 10.

A chapter roadmap of this thesis is given in Figure 1.1. The relation between the individual chapters is as follows. Background information on VerCors, its architecture, the back end Viper and the ADTs supported by VerCors is given in Chapter 2. Next, Chapter 3 answers RQ1 by consulting different sources, resulting in a list of features to implement. With RQ1 answered, the research questions are rephrased in Chapter 4 to be more specific to the remaining features to be implemented. Next, a detailed implementation-level view of VerCors is explained in Chapter 5. Different approaches to implement a feature in VerCors are described in Chapter 6. The features to implement are discussed in Chapters 7, 8, 9 and 10. Chapters 8, 9 and 10 are independent and can be read in any order. Chapter 11 concludes the thesis.

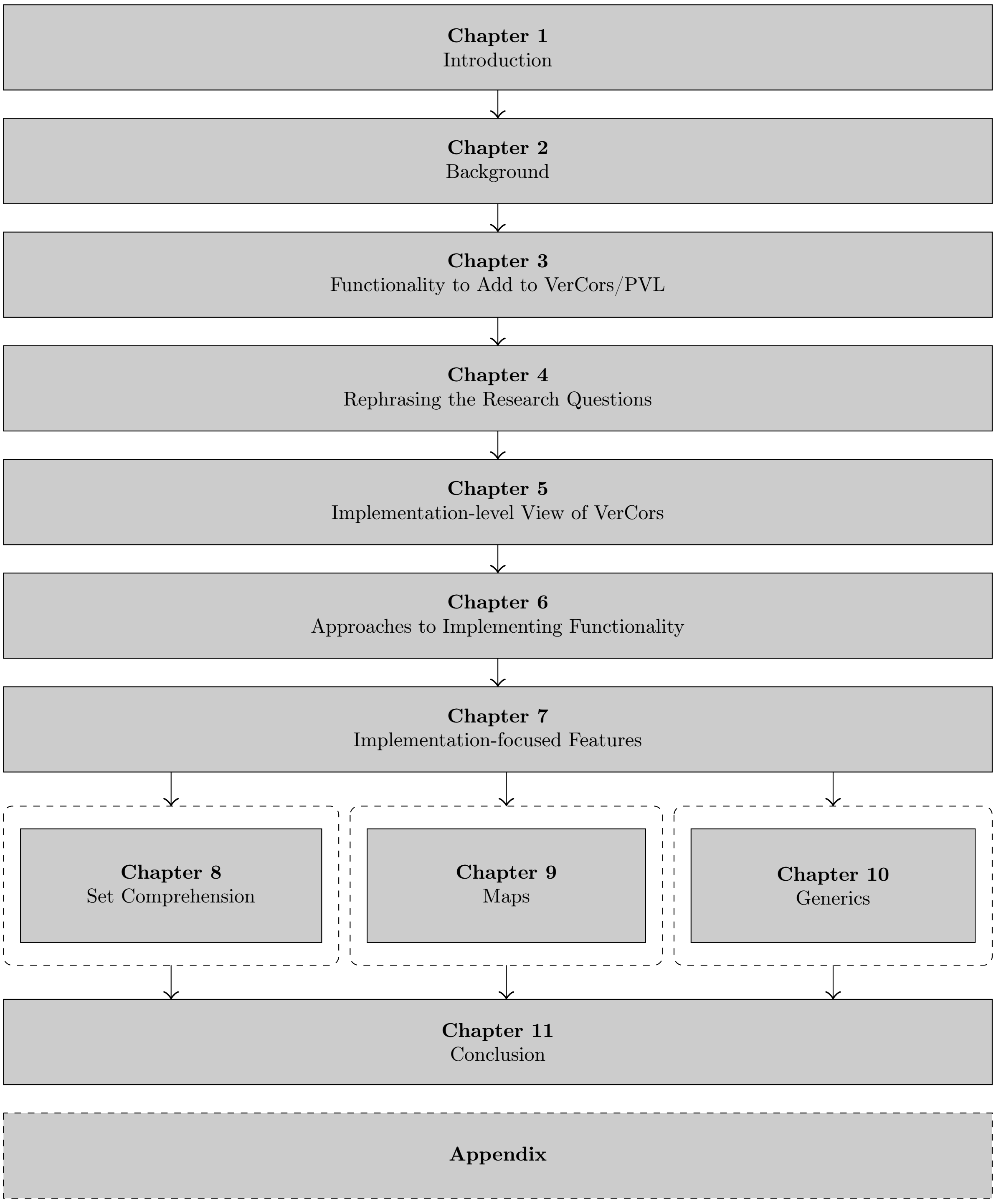

Chapter 1
Introduction
Chapter 2
Background
Chapter 3
Functionality to Add to VerCors/PVL
Chapter 4
Rephrasing the Research Questions
Chapter 5
Implementation-level View of VerCors
Chapter 6
Approaches to Implementing Functionality
Chapter 7
Implementation-focused Features
Chapter 8
Set Comprehension
Chapter 9
Maps
Chapter 10
Generics
Chapter 11
Conclusion
Appendix

Figure 1.1: Chapter roadmap for this thesis

Chapter 2

Background

In this chapter, background information required for the rest of the thesis is discussed. The software verification tool VerCors is introduced with a design-level view of its architecture. The back end Viper is introduced with its language and architecture. Axiomatic data types supported by VerCors are introduced with a discussion on their semantics. Also in this chapter, triggers (or patterns) are introduced with a look at the tool Axiom Profiler to analyze them.

2.1 VerCors

At the core of this research is VerCors. VerCors is a tool for static verification of concurrent and parallel programs developed at the University of Twente [9]. With VerCors, different properties of software can be proven such as data race freedom [10], memory management [11] and functional correctness [10].

This research requires implementation work on VerCors, therefore the inner workings of VerCors are discussed. Figure 2.1 shows a design-level view of the architecture of VerCors taken from [12][1].

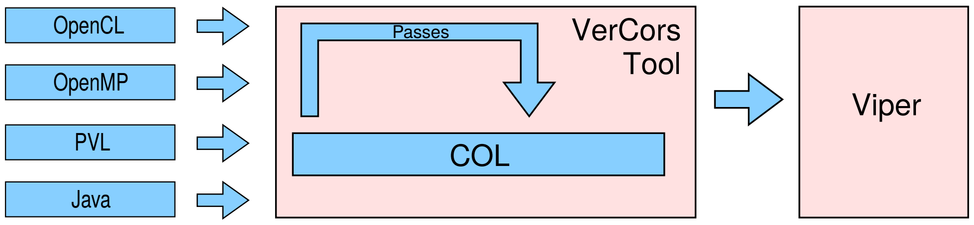


Figure 2.1: Design-level view of VerCors

Figure 2.1 shows that VerCors can be split into three parts:

1. **(Left side)** The translation of an input language to the intermediate language COL.
2. **(Middle)** Passes that transform the COL language into the COL language.
3. **(Right side)** The translation of the COL language into the Viper back end.

[1]It should be noted that some older literature on VerCors, such as [10, 13, 14], represent the VerCors architecture using a (now) outdated figure. Although the architecture of VerCors has not changed drastically, the figures have become somewhat obsolete.

Below a detailed explanation is given of each part, leaving out implementation details such as class/method names as they are discussed in Chapter 5.

2.1.1 The intermediate language: COL

COL (*Common Object Language*) is a language used internally by VerCors during all three steps. It is the intermediary language between the input language and the output language. Since COL is an intermediary language, it does not require a concrete syntax and only has an abstract syntax.

COL is expressed as an AST (*abstract syntax tree*). The nodes in the COL AST represent different structures supported by VerCors. For example, the addition of two constants 1 and 2 can be expressed as a COL AST (see Figure 2.2).

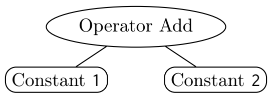


Figure 2.2: A simplified representation of a COL AST for adding two constants

2.1.2 The input languages

VerCors currently supports several input languages such as Java, OpenMP, OpenCL and VerCors' own language PVL. The architecture of VerCors has been designed to easily extend VerCors with new input languages. There are two ingredients to achieve this extendability; First, the COL language and second ANTLR, a parser generator for structured text or binary files [15]. A parser is generated for the new input language using ANTLR and the input file is encoded in COL. The rest of the work in VerCors is independent of the input language.
To support a (new) input language, the following steps are taken in VerCors:

- An ANTLR grammar file is defined for the input language. ANTLR generates a parser from the grammar file. This step is performed only once, since the parser and accompanying Java classes are generated beforehand and bundled with VerCors. During development, VerCors generates a new parser if the grammar changed.
- Using the parser, the input file is converted into a parse tree. This is functionality provided by ANTLR.
- This parse tree is transformed into a COL AST. This COL AST is the basis for the rest of the transformations performed by VerCors.

Prototypal Verification Language

PVL (*Prototypal Verification Language*) is described as a procedural toy language for prototyping verification features [16]. To be more precise, PVL is a *purely* prototypal language meaning it has no runtime environment. This gives the benefit of easily adding new features to PVL and by extension VerCors without having to implement a compiler or interpreter. PVL is a simple object-oriented language with verification flow constructs as part of the language (e.g. constructs to specify preconditions, loop invariant, etc.).

Listing 2.1 shows a simple PVL program. This program consists of a class `Incrementer` with a method `incrementAllByN`. This method takes two arguments: a sequence/list of integers `input` and an integer `n`. Inside the method, a sequence `res` is created. Using a while loop, we iterate over the input list and append the value plus `n` to the sequence `res`. Pre/post conditions and loop invariant are used to aid the tool to verify the behavior of the program.

Besides the constructs in the example, PVL supports many other constructs. For the rest of the thesis, PVL syntax is only introduced where necessary. An overview of the syntax of PVL can be found at [17].

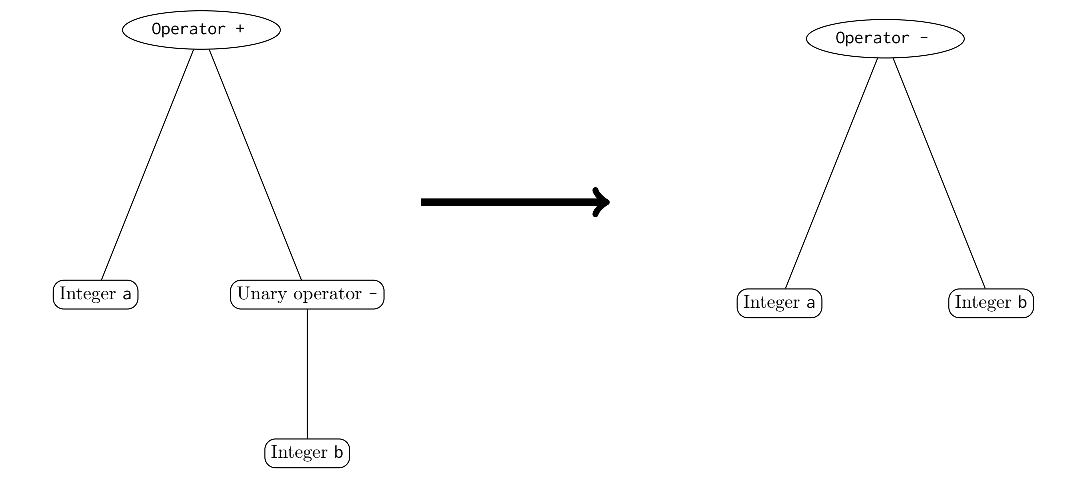
```
class Incrementer {
  requires n >= 0;
  ensures |\result| == |input|;
  ensures (\forall int k; 0 <= k && k < |\result|; \result[k] == input[k]+n);
  seq<int> incrementAllByN(seq<int> input, int n) {
    seq<int> res = seq<int> {};
    int i = 0;

    loop_invariant 0 <= i && i <= |input|;
    loop_invariant i == |res|;
    loop_invariant (\forall int k; 0 <= k && k < i; res[k] == input[k]+n);
    while (i < |input|) {
      res = res + seq<int> {input[i]+n};
      i = i + 1;
    }
    return res;
  }
}
```

Listing 2.1: A simple program in PVL

2.1.3 The transformations: Passes

After the initial transformation of the input language to a COL AST, the resulting COL AST goes through a series of transformations called *passes* in VerCors. These passes are tree visitors that aim to transform the COL AST into a new COL AST. Each pass changes parts of the COL AST with a specific purpose in mind.

For clarity, suppose we have a pass named `simplifyMathExpressions`. The goal of this pass is to simplify expressions of the form $a + (-b)$ to $a - b$ where `a` and `b` are integers. When this pass is applied to the current COL AST, the tree is traversed and the combination of the plus and minus operators is matched and rewritten. Figure 2.3 shows a visual representation of this pass.



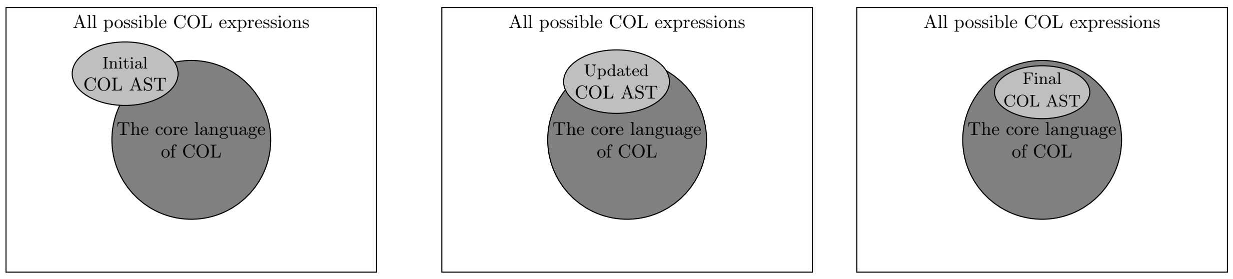
(a) Original COL AST, before the pass (b) New COL AST, after the pass

Figure 2.3: An example pass `simplifyMathExpressions`

2.1.4 The final transformation: From COL to back end language

The final COL AST, which is the result of all the passes, should contain only expressions that have a mapping to the back end language. The reason for this constraint is that only a subset of all possible expressions in COL can be mapped to a structure used by the back end. That subset we refer to as the *core language* of COL.

The initial COL AST possibly contains expressions that are not part of the core (see Figure 2.4a) and those expressions will be rewritten (by a pass) into expressions that are part of the core (see Figure 2.4b). After all passes have finished, the final COL AST consists only of core language expressions (see Figure 2.4c).



(a) Initial COL AST.
The initial COL AST is defined partially by the core and partially by non-core expression.

(b) Updated COL AST.
After a pass, some of the non-core expressions are expressed with equivalent expressions in the core language.

(c) Final COL AST.
The final COL AST consists only of expression of the core language.

Figure 2.4: A visual representation of the COL AST from the initial AST to the final AST

What is and isn't part of the core of COL is dependent on the chosen back end. The back end has its own set of functionality that the core of COL can be directly mapped to. For example, if both the chosen back end and PVL have an operator to get the size of a sequence, then the PVL operator can be mapped to the operator in the back end. In the initial step, the PVL syntax is transformed into a COL expression with the size/length operator and that operator is directly mapped to the equivalent operator in the back end.

It can be assumed that for any new, self-defined operator a mapping does not exist and thus a pass will have to be defined to transform it into something that is mapped. It can also be assumed that if an operator exists, the operator either directly maps to a back end operator or a pass rewrites the existing operator into another operator that is mapped to the back end.

2.1.5 The back end: Viper

The main back end of VerCors is Viper. Viper (*Verification Infrastructure for Permission-based Reasoning*) is a toolchain built for quick and simple development of other verification tools [8]. A visual representation of the Viper infrastructure can be seen in Figure 2.5 (based on the figure drawn by [18], p. 11 fig. 3).

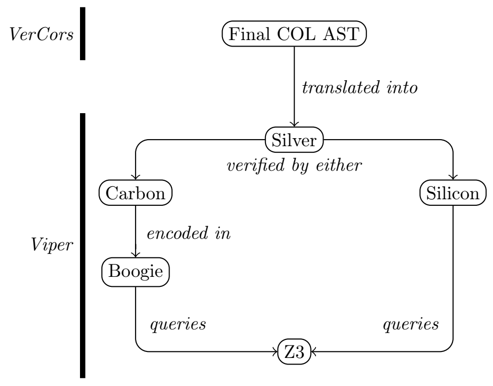


Figure 2.5: The Viper infrastructure

The input language of Viper is called Silver (see below). VerCors transforms the final COL AST into a Viper problem (in the form of a Silver file), Viper processes the problem using one of its back ends and encodes it as an SMT problem to be solved by Z3.

Viper supports two back ends named Carbon and Silicon that use different techniques to verify the problem. Silicon is an automated verifier based on symbolic execution [19] and Carbon is a verifier based on the generation of verification conditions [20]. When verifying a problem with VerCors, a flag is set to run Viper with either Carbon or Silicon.

2.1.6 Silver

Silver is an imperative language with methods/functions at top-level. Since the COL AST is transformed into a Silver AST, all relevant constructs for this thesis are introduced based on the Viper tutorial[2].

Natively supported types

The Viper tutorial lists the types supported by Silver/Viper:

[2]For a complete overview of Silver constructs, please visit http://viper.ethz.ch/tutorial/.

- `Int`: mathematical integers.
- `Bool`: Booleans.
- `Rational`: Mathematical rationals.
- `Perm`: Permission amounts. 0 or `none` for no permission, 1 or `write` for write permissions and a value in between 0 and 1 for read permission.
- `Ref`: References to objects.
- `Seq[T]`: Immutable sequences.
- `Set[T]`: Immutable sets.
- `Multiset[T]`: Immutable multisets.

Fields

Objects are modeled in Silver as references. The fields of these objects are declared at top-level. Listing 2.2 shows an example of two fields `myInt` and `myBool` of types `Int` and `Bool` respectively.

```
field myInt: Int
field myBool: Bool
```

Listing 2.2: An example of fields in Silver

Every reference (to an object) has all declared fields. A field can be accessed iff there is sufficient permission for it. Permission to a certain field can be acquired using the accessibility predicate `acc`. This predicate is often used in the pre/postcondition of a method to acquire/return permission to read or write to a field.

Listing 2.3 shows an example using the accessibility predicate. Suppose that we want to model an object `incrementer` that has an integer field `inc` and we want to define a method named `myMethod` to increment that field by 1. On the top-level we define the field (line 1) and the method (lines 3-9) with the `incrementer` object instance as an argument of type `Ref`. In the precondition, write permission for the field `inc` on the object `incrementer` is acquired using `acc`. The same permission is returned, again, using the accessibility predicate. If sufficient permission would not be acquired, for example no permission or only read permission, the verification effort would fail with cause *"There might be insufficient permission to access incrementer.inc"*.

```
1  field inc : Int
2
3  method myMethod(incrementer: Ref)
4    requires acc(incrementer.inc, write)
5    ensures acc(incrementer.inc, write)
6    ensures incrementer.inc == old(incrementer.inc) + 1
7  {
8    incrementer.inc := incrementer.inc + 1
9  }
```

Listing 2.3: Example of permissions in Silver

Pure functions

Functions in Silver are pure, meaning that a call to that function does not affect the state of the program (including permissions). This allows for functions to be used both in code and specifications.

Listing 2.4 shows an example similar to Listing 2.3. Two functions `myFunc` and `equivalentMyFunc` have been defined with the `incrementer` object as their argument. The preconditions state that there is 1/2 permission (i.e. read permission) on the field `inc`.

The difference between the two functions is in the optional body. The body consists of a single (possibly recursive) expression. This expression is implicitly used as a postcondition. Instead of defining the body, the behavior of the function can also be expressed in the postcondition. The function `myFunc` uses a body and thereby expresses its behavior and the function `equivalentMyFunc` uses the postcondition to express its behavior using the keyword `result` to refer to the result of the function.

```
field inc : Int

function myFunc(incrementer: Ref): Int
  requires acc(incrementer.inc, 1/2)
{
  incrementer.inc + 1
}

function equivalentMyFunc(incrementer: Ref): Int
  requires acc(incrementer.inc, 1/2)
  ensures result == incrementer.inc + 1
```

Listing 2.4: Examples of a function in Silver

Domains

New ADTs can be introduced in Silver using domains. Domains consist of a name, type parameters, functions and axioms.

- **Domain name**: The name of the new type. This name is globally unique and globally accessible.
- **Type parameters**: The type parameters for the new type. The scope for these types is the domain. These type parameters have to be instantiated when used outside of the domain.
- **Functions**: Function signatures defined in the domain. The name of a function is globally unique and globally accessible. The functions are bodyless and uninterpreted (if no accompanying axioms are defined over it).
- **Axioms**: Expressions that express the behavior of the functions and the relation between them. An axiom has a globally unique name and a body consisting of a Boolean expression to express the behavior of a function or the relation between functions.

Only the name of the domain is strictly mandatory. Functions and axioms are not strictly mandatory, however a domain without functions and axioms over it is only a type. No instance of that type can be obtained since there are no constructor functions.

Listing 2.5 shows a domain modeling a tuple. The domain has two type parameters `F` and `S` that are the types of the first and second element in the tuple respectively. A function `constructor` is declared along with two destructor functions `fst` and `snd`. The function `constructor` takes two arguments of type `F` and `S` and returns a `Tuple[F,S]` and the two destructors take a `Tuple[F,S]` and return something of type `F` and `S` respectively.

The behavior of these functions and the relation between them are expressed using two axioms named `FstAxiom` and `SndAxiom`. The `FstAxiom` can be read as follows; for all elements `f1` and s1 of type F and S respectively, if a tuple is constructed (using the constructor function), the function fst returns the first element f1. The axiom SndAxiom can be read similarly.

```
domain Tuple[F,S] {
   function constructor(f:F, s:S): Tuple[F,S]
   function fst(t:Tuple[F,S]): F
   function snd(t:Tuple[F,S]): S

   axiom FstAxiom {
      forall f1:F, s1:S :: fst(constructor(f1,s1)) == f1
   }

   axiom SndAxiom {
      forall f1:F, s1:S :: snd(constructor(f1,s1)) == s1
   }
}
```

Listing 2.5: A Tuple domain

Axioms are assumed to hold in every state of the program. Unsoundness can be introduced if an axiom is unsatisfiable or if two axioms are contradictory. Consider the example in Listing 2.6. A domain MyDomain is made with an axiom stating false. Since this axiom should hold in all states of the program, the assertion false in myMethod succeeds[3].

```
domain MyDomain {
  axiom falseAxiom {
    false
  }
}

method myMethod() {
  var myVariable: MyDomain
  assert false
}
```

Listing 2.6: Example of unsoundness caused by domain axioms

2.1.7 Natively supported ADTs in Viper

Viper natively supports three composite data types: sequences, sets and bags. Viper expresses these data types axiomatically (in the same way as domains), hence they are called *axiomatic* data types. The two back ends Carbon and Silicon have their own axiomatizations for these ADTs using the SMT2 format used by Z3 [21, 22].

The implementations of these ADTs are similar to the tuple example in Listing 2.5. The type Seq is introduced and functions over Seq are declared. The behavior of the functions and the relation between them is expressed using axioms.

For completeness, let us look at an axiom that expresses that when two sequences are appended, the length of the resulting sequence is equal to the sum of the length of the two sequences. Listing 2.7 shows the axiom in the SMT2 format. Some predefined functions are used in this axiom, namely the function Seq#Empty() for an empty sequence, the function Seq#Length(s0) for the length of the argument and Seq#Append(s0, s1) for appending the two arguments. The

[3]For some versions of Viper, it suffices to have the domain specified without using the domain to get unsoundness. For other versions, the unsoundness is only introduced when the domain is used at least once in the program.

axiom reads as follows: for all sequences `s0` and `s1` (line 1), if both `s0` and `s1` are not empty (line 2), then the length of `s0` and `s1` appended is equal to the length of `s0` plus the length of `s1`.

```
1 axiom (forall<T> s0: Seq T, s1: Seq T ::
2   s0 != Seq#Empty() && s1 != Seq#Empty() ==>
3     Seq#Length(Seq#Append(s0,s1)) == Seq#Length(s0) + Seq#Length(s1));
```

Listing 2.7: Example of an axiom

2.2 Natively Supported ADTs in VerCors

VerCors natively supports three ADTs: sequences, sets and bags. These ADTs are directly mapped to Viper's sequences, sets and multisets respectively. Since these three ADTs are both natively supported by VerCors and Viper, a definition is given below of each ADT with a small discussion on their definition in different contexts.

2.2.1 Sequences

When it comes to an established definition of a sequence, there is no single definition broad enough to capture everything since some definitions contradict others. Let us examine by which properties a sequence is defined in other contexts.

- The mathematical definition speaks mostly about sequences of integers. In the context of programming languages, sequences are also defined on other types such as objects, thus this definition is ignored.
- In programming languages often one of two terms is used to describe sequences: sequence or list. For example, Java uses the term list and PHP uses the term sequence. They might even be used interchangeably. Some languages have both sequences and lists which are used to describe similar yet different data structures, for example Haskell lists can be infinite and Haskell sequences are finite.
- Sequences can be strictly finite or possibly infinite. Infinite sequences are achieved using different techniques such as lazy evaluation baked into the language (in Haskell) or generators (in Python).
- Some languages allow sequences to be changed, in other words, the sequences are mutable. Other languages have immutable sequences where if it is required to change the sequence, the sequence is first copied and the changes are applied to the new sequence.

In short, the most general definition of a sequence is an ordered/enumerated collection. Other assumptions on sequences/lists are language-dependent. The most-suitable definition for sequences in VerCors is *an ordered/enumerated, finite, immutable collection.*

2.2.2 Sets and bags

Similar to sequences, sets and bags also have no single definition. The exact definition is context-dependent. Let us first examine by which properties sets are defined.

- The mathematical definition of sets is an unordered collection of objects (e.g. numbers, matrices or arrays). Although mathematical objects are not the same as objects in object-oriented programming, a set in programming can also be defined with the same definition;

an unordered collection of objects. Bags are called multisets in mathematics which are also unordered collections of objects. The difference between sets and bags is that values in sets are unique and in bags values are not necessarily unique. For example, the set $\{1, 2, 3\}$ is equivalent to the bag $\{1, 2, 3\}$ and not equivalent to the bag $\{1, 1, 2, 3\}$.

- Sets/bags are either mutable or immutable.
- Sets/bags are either strictly finite or possibly infinite. In the context of programming, sets/bags are mostly finite. There are implementations for infinite sets, however these implementations have their limitations. For example, on the main page of the Haskell package `lazyset`[4] it is mentioned that it implements infinite sets using infinite lists, however the implementation fails if the infinite set is filtered. Another example is an article on DZone[5]. This implementation uses Java streams, however it has two limitations. First, the implementation is immutable and second the size of the set is limited by the maximum number of elements in a Java stream which is $2^{63} - 1$. Although sets with the size larger than $2^{63} - 1$ are possible, the size is reported as $2^{63} - 1$. The same limitations also hold for a bag implementation.

In short, the most general definition of a set is an unordered collection of objects with unique values and the most general definition of a bag is an unordered collection of objects. Other properties are language-dependent. The most suitable definition for sets in VerCors is *an unordered, finite, immutable collection with unique values* and the most suitable definition for bags is *an unordered, finite, immutable collection*.

2.3 Triggers

As stated above, both back ends of Viper use the SMT solver Z3 to solve the problem at hand. SMT solvers (*Satisfiability Modulo Theories*) are tools to decide the satisfiability of formulas in first-order theories (e.g. arithmetic or arrays). The efficiency of these tools is based on various heuristics of which one is finding candidates (i.e. constants) to instantiate universal quantifiers. These candidates are called *triggers*[6] and can be either user-provided or heuristically found with different heuristics [23, 24, 25].

Listing 2.8 shows an example of how triggers help the SMT solver. Suppose, we have a domain `MyDomain` with a function `f` with an integer argument. The axiom `fAxiom` states that for all integers `k` larger than zero, the result of the function (i.e. `f(k)`) is larger than zero. The trigger `f(k)` is defined in between curly braces. Whenever this trigger is matched, the body of the universal quantifier is instantiated with `k` bound to the constant.

Next, we have a method `myMethod` with an integer argument `l` that is required to be larger than zero. The body of the method consists of an assertion with a part of the assertion matching the trigger. Since the trigger matches, the SMT solver instantiates the body of the universal quantifier with the constant `l` and adds `(l > 0) ==> (f(l) > 0)` to its pool of knowledge. With this knowledge, the assertion is verified.

If the trigger would not be matched, the body of the universal quantifier would not be instantiated and the assertion would fail. For example, if the trigger in Listing 2.8 was `g(k)` with `g` being another function, the trigger does not match and as a result the body of the universal quantifier is not instantiated and so the assert fails.

[4]A link to the main page of the `lazyset` package: `https://hackage.haskell.org/package/lazyset`

[5]Link to the DZone article: `https://dzone.com/articles/infinite-sets-in-java-9`. Although DZone is no authority, the implementation and its limitations are interesting to mention.

[6]Triggers are also called patterns in tools such as Z3 and Simplify. In this thesis, the word trigger is used even in the context of tools that use the word pattern.

```
domain MyDomain {
  function f(i:Int): Int

  axiom fAxiom {
    forall k: Int :: {f(k)} (k > 0) ==> (f(k) > 0)
  }
}

method myMethod(l: Int)
requires l > 0
{
  assert f(l) > 0
}
```

Listing 2.8: An example of a trigger

The example above has a single trigger `f(k)` as part of a single trigger set. More triggers can be added to that set to make the set more restrictive. For example, if the trigger would state `{f(k), g(k)}` with some function `g`, the instantiation would only happen if both functions `f` and `g` have been used with the same argument. It is also possible to specify multiple trigger sets where, if all triggers in one of the sets are matched, the body of the universal quantifier is instantiated. For example, for the two trigger sets `{f(k)}{g(k)}`, the instantiation would happen if `f(k)` is matched or `g(k)` is matched.

In essence, a trigger is a (Silver) expression and there are certain restrictions placed on these expressions by Viper[7]:

- *Each quantified variable must occur at least once in a trigger set.* For example, if we quantify over `x`, `y` and `z`, all three variables must be mentioned within the trigger set.
- *Each trigger expression must include at least one quantified variable.* For example, a function without any arguments cannot be a trigger.
- *Each trigger expression must have some additional structure (typically a function application); a quantified variable alone cannot be used as a trigger expression.*
- *Arithmetic and boolean operators may not occur in trigger expressions.* In Viper, the membership operator `in` and the subset operator `subset` are allowed as triggers.
- *Accessibility predicates (the acc keyword) may not be used in trigger expressions.*

2.3.1 Categorization of trigger

Triggers can be categorized into two groups based on how they are effectively matched: *good* trigger and *bad* triggers. We define a good trigger as a trigger that helps the SMT solver to prove the given problem. Whether a trigger is good (or not) is case-dependent. A bad trigger we define as a trigger that does not help the SMT solver. Bad triggers are split into two groups: *restrictive* triggers and *looping* triggers.

Listing 2.9 shows an example of a function `f` with the behavior expressed using three different triggers (based on the Viper Tutorial). Depending on the trigger, the SMT solver verifies or fails in cases that seem straightforward. The behavior of the function is expressed as an assumption in the three methods.

[7]These restrictions are listed in the Viper Tutorial [26]. The italic parts are quoted from the tutorial.

- `restrictive_trigger`: The chosen trigger is `f(f(f(k)))`. The trigger is only matched if the function `f` is used within the same function twice. This trigger is not used in the assumption or in any other expression, meaning that it is never matched and so the assertion fails.
- `good_trigger`: The chosen trigger is `f(f(k))`. This trigger matches against the expression `f(f(3))` and the body of the universal quantifier is instantiated with $k \to 3$. This results in the verification of the assertion.
- `looping_trigger`: The chosen trigger is `f(k)`. For the first assertion, the trigger matches `f(3)` (with `k` $\leftarrow$ `3`), `f(f(3))` (with `k` $\leftarrow$ `f(3)`) and `f(9)` (with `k` $\leftarrow$ `9`). Since `f(3)` is matched, `f(f(3)) == f(3*3)+1` is instantiated and the assertion succeeds.

  For the second assertion, the verifier goes into an matching loop. The trigger matches `f(3)` (with `k` $\leftarrow$ `3`), `f(f(3))` (with `k` $\leftarrow$ `f(3)`) and `f(4)` (with `k` $\leftarrow$ `4`). For `f(3)`, `f(f(3)) == f(3*3)+1` is instantiated. Parts of this new expression again matches the trigger, namely `f(9)`. This results in `f(f(9)) == f(9*9)+1` being added to the pool of knowledge. As can be seen, parts of this new expression again match the trigger and so a matching loop is formed. The same holds for the other two matched expression.

  A matching loop in itself is not necessarily a bad trigger according to the definition given above. If the matching loop is finite in all cases (e.g. in a recursive definition), it is considered a good trigger since it helps the SMT solver verify the problem, although it is inefficient compared to a good trigger. However, if the matching loop is infinite, the SMT solver times out and fails to verify the problem. This difference can be seen in the two assertions where the first assertion is a finite matching loop (with one iteration) and the second assertion is an infinite loop.

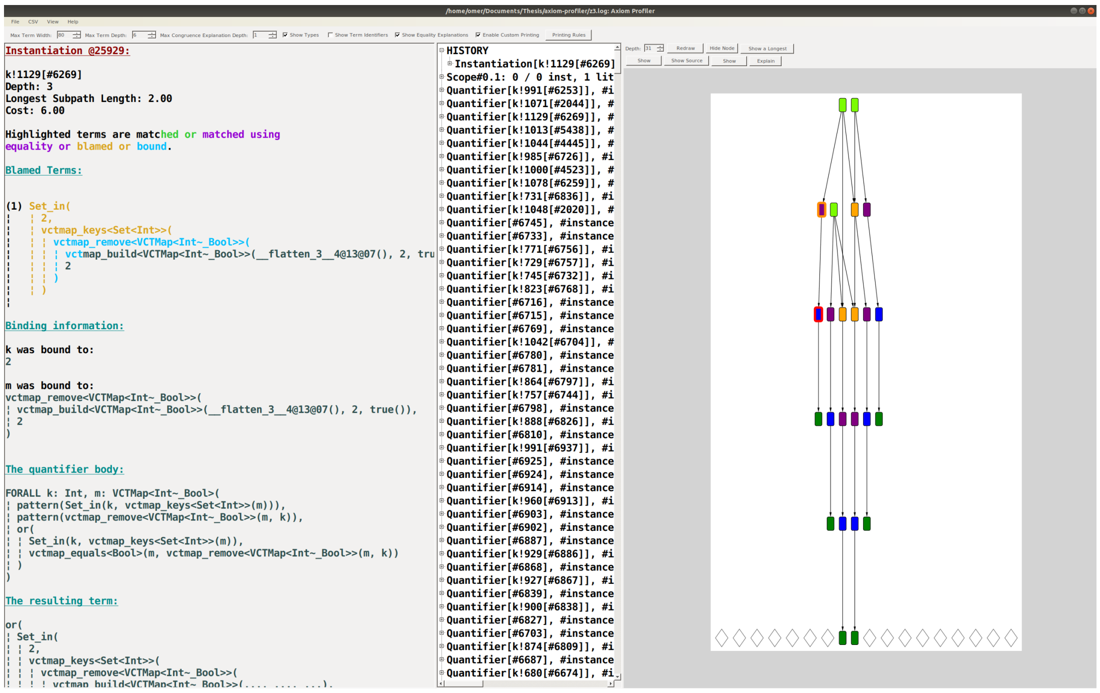
```
function f(i: Int): Int

method good_trigger() {
  assume forall k: Int :: {f(f(k))} f(f(k)) == f(k*k)+1
  assert f(f(3)) == f(9)+1 // Verifies
}

method restrictive_trigger() {
  assume forall k: Int :: {f(f(f(k)))} f(f(k)) == f(k*k)+1
  assert f(f(3)) == f(9)+1 // Fails
}

method looping_trigger() {
  assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
  assert f(f(3)) == f(9)+1 // Verifies
  assert f(f(3)) == f(4)+1 // Times out
}
```

Listing 2.9: Different categories of triggers and their behaviors

2.3.2 Axiom Profiler

The triggers in Listing 2.9 and matching expressions are simple enough to reason about and hypothesize which trigger is matched against which expression. As the program grows, reasoning about triggers and instantiated terms becomes much more difficult. The tool *Axiom Profiler* can be used to *understand and debug SMT quantifier instantiations* [27]. Provided with a Z3 log, Axiom Profiler visualizes the instantiations and produces readable text to understand what happens in Z3.

The user-interface of Axiom Profiler is divided into three columns (see Figure 2.6). The middle column is *raw data*[8]. The right column is the *instantiation graph*. The colored nodes in the graph are instantiations where different colors represent different quantifiers and if the newly instantiated expression results in another instantiation, an arrow is drawn in between them. When a node is selected, the *instantiation information* is shown in the left column which includes which trigger is matched, what the variables are bound to and what the resulting expression is.



```
Max Term Width: 80   Max Term Depth: 6   Max Congruence Explanation Depth: 1   [✓] Show Types   [ ] Show Term Identifiers   [✓] Show Equality Explanations   [✓] Enable Custom Printing
Depth: 31
```

```
Instantiation @25929:

k!1129[#6269]
Depth: 3
Longest Subpath Length: 2.00
Cost: 6.00

Highlighted terms are matched or matched using
equality or blamed or bound.

Blamed Terms:


(1) Set_in(
¦    ¦ 2,
¦    ¦ vctmap_keys<Set<Int>>(
¦    ¦ ¦ vctmap_remove<VCTMap<Int~_Bool>>(
¦    ¦ ¦ ¦ vctmap_build<VCTMap<Int~_Bool>>(__flatten_3__4@13@07(), 2, tru
¦    ¦ ¦ ¦ 2
¦    ¦ ¦ )
¦    ¦ )
¦

Binding information:

k was bound to:
2

m was bound to:
vctmap_remove<VCTMap<Int~_Bool>>(
¦ vctmap_build<VCTMap<Int~_Bool>>(__flatten_3__4@13@07(), 2, true()),
¦ 2
)


The quantifier body:

FORALL k: Int, m: VCTMap<Int~_Bool>(
¦ pattern(Set_in(k, vctmap_keys<Set<Int>>(m))),
¦ pattern(vctmap_remove<VCTMap<Int~_Bool>>(m, k)),
¦ or(
¦ ¦ Set_in(k, vctmap_keys<Set<Int>>(m)),
¦ ¦ vctmap_equals<Bool>(m, vctmap_remove<VCTMap<Int~_Bool>>(m, k))
¦ )
)

The resulting term:

or(
¦ Set_in(
¦ ¦ 2,
¦ ¦ vctmap_keys<Set<Int>>(
¦ ¦ ¦ vctmap_remove<VCTMap<Int~_Bool>>(
¦ ¦ ¦ ¦ vctmap_build<VCTMap<Int~_Bool>>(.... .... ...),
```

```
⊟ HISTORY
  ⊞ Instantiation[k!1129[#6269]
⊞ Scope#0.1: 0 / 0 inst, 1 lit
⊞ Quantifier[k!991[#6253]], #i
⊞ Quantifier[k!1071[#2044]], #
⊞ Quantifier[k!1129[#6269]], #
⊞ Quantifier[k!1013[#5438]], #
⊞ Quantifier[k!1044[#4445]], #
⊞ Quantifier[k!985[#6726]], #i
⊞ Quantifier[k!1000[#4523]], #
⊞ Quantifier[k!1078[#6259]], #
⊞ Quantifier[k!731[#6836]], #i
⊞ Quantifier[k!1048[#2020]], #
⊞ Quantifier[#6745], #instance
⊞ Quantifier[#6733], #instance
⊞ Quantifier[k!771[#6756]], #i
⊞ Quantifier[k!729[#6757]], #i
⊞ Quantifier[k!745[#6732]], #i
⊞ Quantifier[k!823[#6768]], #i
⊞ Quantifier[#6716], #instance
⊞ Quantifier[#6715], #instance
⊞ Quantifier[#6769], #instance
⊞ Quantifier[k!1042[#6704]], #
⊞ Quantifier[#6780], #instance
⊞ Quantifier[#6781], #instance
⊞ Quantifier[k!864[#6797]], #i
⊞ Quantifier[k!757[#6744]], #i
⊞ Quantifier[#6798], #instance
⊞ Quantifier[k!888[#6826]], #i
⊞ Quantifier[#6810], #instance
⊞ Quantifier[k!991[#6937]], #i
⊞ Quantifier[#6925], #instance
⊞ Quantifier[#6924], #instance
⊞ Quantifier[#6914], #instance
⊞ Quantifier[k!960[#6913]], #i
⊞ Quantifier[#6903], #instance
⊞ Quantifier[#6902], #instance
⊞ Quantifier[#6887], #instance
⊞ Quantifier[k!929[#6886]], #i
⊞ Quantifier[#6868], #instance
⊞ Quantifier[k!927[#6867]], #i
⊞ Quantifier[#6839], #instance
⊞ Quantifier[k!900[#6838]], #i
⊞ Quantifier[#6827], #instance
⊞ Quantifier[#6703], #instance
⊞ Quantifier[k!874[#6809]], #i
⊞ Quantifier[#6687], #instance
⊞ Quantifier[k!680[#6674]], #i
```

Figure 2.6: The user-interface of Axiom Profiler

As an example, let us show the instantiation graphs produced by Axiom Profiler for the methods in Listing 2.9.

- `good_trigger`, Figure 2.7a: There are seven nodes meaning that there are seven cases where a universal quantifier body is instantiated. When Viper encodes its own problem into the SMT2 format, it introduces (internally used) axioms and functions. The purple and blue nodes are related to those internal functions. The node which is interesting for our method is the green node.

  Figure 2.7b shows the instantiation information of the green node. Do note here that the functions `f%limited` (i.e. the function generated by Viper) and `f` are defined to be equivalent and the expression `$Snap.unit()` can be ignored since it is not used by our expression and has no effect on this case. The instantiation information reads as follows:

  1. Blame: f(f(3)). This means that the blamed term is matched against our trigger.

[8]For the rest of the thesis, the middle column is hidden since the data is difficult to comprehend in that form.

2. Bind: $k \leftarrow 3$. With the blamed term, the variable `k` is bound to the constant 3.
3. The shown quantifier body is the one we defined in the `good_trigger` method.
4. Resulting term: $1 == f(f(3)) + -f(3 \times 3)$. The term resembles the body of the quantifier which is rewritten by Z3 using simple arithmetic rules.

Using the resulting term, the assertion is verified.

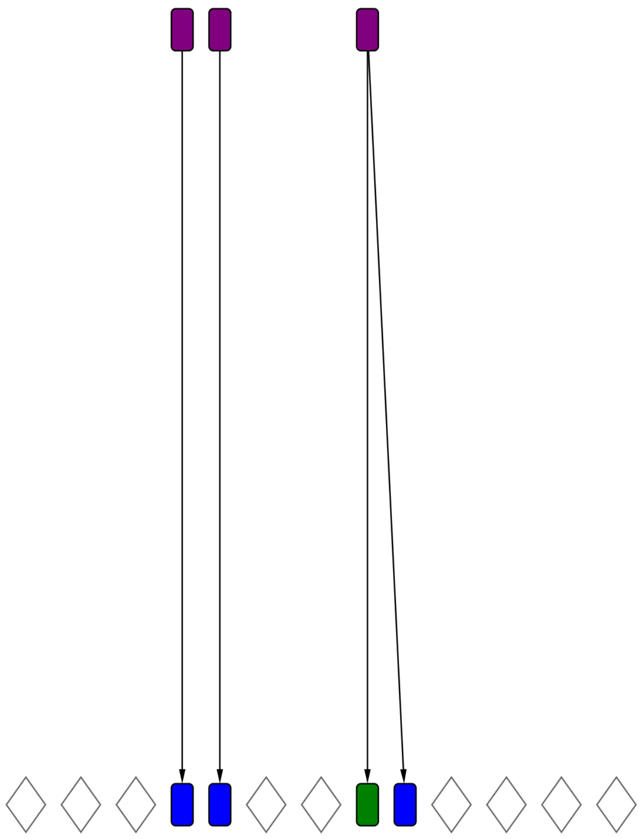
For the rest of the thesis, the instantiation information column is not shown. Instead, the information in this column is presented in text using the instantiation graph. The reason for this is to present the information in a readable format (as in the enumeration above) instead of the SMT2 format used by Z3 and to abstract from variable/function names generated by Viper which are not relevant for the analysis.

- `restrictive_trigger`, Figure 2.7c: We reasoned that the assertion in this method fails since the trigger is not matched and the instantiation graph confirms our hypothesis. The purple and blue nodes are all related to functions generated by Viper. Since there is no expression matching the trigger, the instantiation graph does not show an instantiation for that trigger and the assertion fails.
- `looping_trigger`, Figure 2.7d: We hypothesized that in this method three terms matched the trigger: `f(4)`, `f(f(3))` and `f(3)`. In the instantiation graph we see these three term in the form of three paths (from top to bottom) annotated with `(1)`, `(2)` and `(3)` respectively. The purple nodes are again related to a function generated by Viper and thus can be ignored. The blue nodes are related to our function.

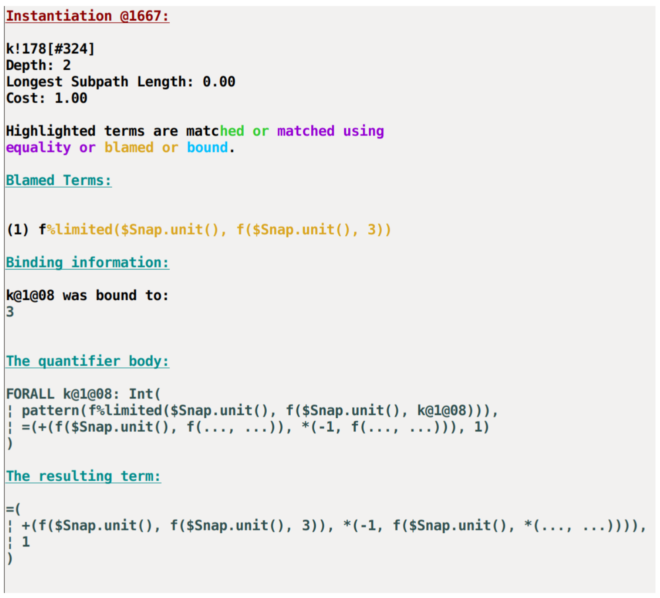
  For path `(1)`, the first blue node from the top is the trigger matching `f(4)`. The variable `k` (in the trigger) is bound to 4 and the resulting term is $f(f(4)) + -f(4 \times 4) == 1$. There are two new expressions in this new term matching our trigger, namely `f(4×4)` and `f(f(4))`. These new expressions are again matched and new terms are instantiated as represented by the two blue nodes in path `(1)` labeled `(4)`. These new terms in turn contain new expressions matching the trigger and so a matching loop is started. The same holds for the other two paths.

  A visual inspection of the instantiation graph already points to a matching loop. There is a match of a trigger in a quantifier represented by some colored node (in this case the blue node). The resulting term matches some other trigger (in this case the purple node). This continues until the same trigger is matched again (in this case the blue node) or in other words there is a loop of the same triggers being matched.

  If there is an infinite matching loop, the instantiation graph is (in theory) infinite. By default, Axiom Profiler shows only a small part of the graph. By allowing more nodes to be drawn, the entire graph can be shown. The graph that is drawn is still finite. While Z3 tries to verify the problem by matching triggers and instantiating more terms, it times out and only a finite amount of instantiations are shown. However, if left running, Z3 keeps on instantiating new terms and never stops.

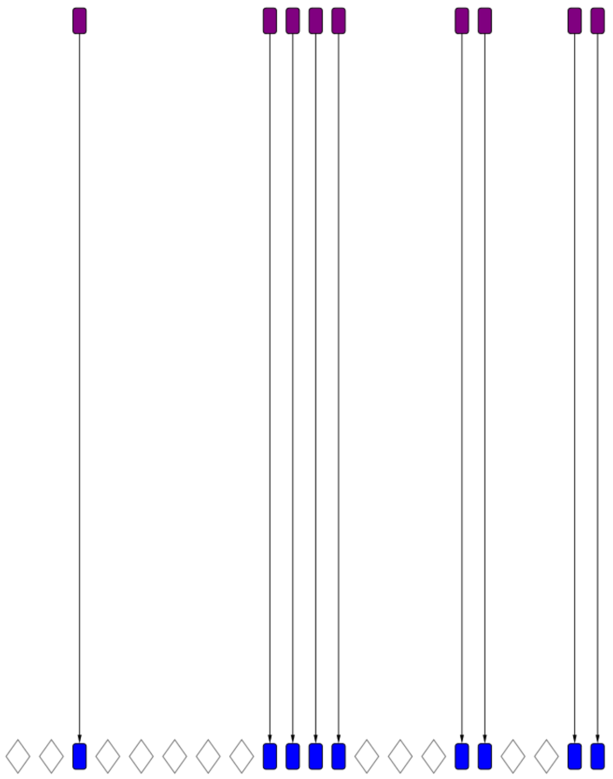


(a) The instantiation graph for the method `good_trigger`

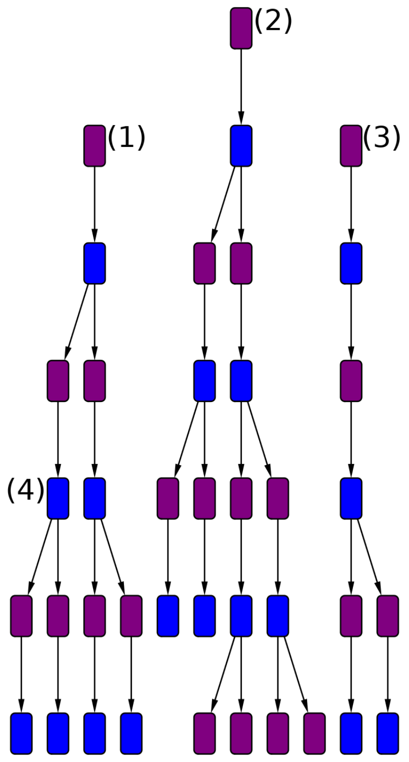


```
Instantiation @1667:

k!178[#324]
Depth: 2
Longest Subpath Length: 0.00
Cost: 1.00

Highlighted terms are matched or matched using
equality or blamed or bound.

Blamed Terms:


(1) f%limited($Snap.unit(), f($Snap.unit(), 3))

Binding information:

k@1@08 was bound to:
3


The quantifier body:

FORALL k@1@08: Int(
¦ pattern(f%limited($Snap.unit(), f($Snap.unit(), k@1@08))),
¦ =(+(f($Snap.unit(), f(..., ...)), *(-1, f(..., ...))), 1)
)

The resulting term:

=(
¦ +(f($Snap.unit(), f($Snap.unit(), 3)), *(-1, f($Snap.unit(), *(..., ...)))),
¦ 1
)
```

(b) The instantiation information for the method `good_trigger`



(c) The instantiation graph for the method `restrictive_trigger`



(d) The instantiation graph for the method `looping_trigger`

Figure 2.7: An analysis of the methods in Listing 2.9 using Axiom Profiler

Chapter 3

Functionality to Add to VerCors/PVL

In this chapter, the first research question on what functionality is desired by users is answered. Different sources have been used to answer the question including a survey. The survey questions and results are discussed in Sections 3.1 and 3.2. Section 3.3 presents the result in a list of features to implement in VerCors/PVL.

When looking through the GitHub page of VerCors, two sources can be found that give a general idea of what functionality users are missing. These two sources are the VerCors ADT documentation [28] and the VerCors examples directory [29]. The VerCors ADT documentation contains sections on possible future enhancements. These enhancements are suggestions from users who both work on the back end of VerCors and use VerCors for their own verification. The VerCors examples directory contains examples in different languages such as Java and PVL that give an idea of what functionality is written by the users to help their verification efforts.

From these two sources, we built the following list of functionality to add to VerCors:

- Maps
- Set comprehension
- List comprehension[1]
- Subset notation for sets and bags
- Simple syntax for sequence, set and bag creation. For example, instead of `seq<int> {a, b, c}` we would have `[a, b, c]`.
- Appending and prepending values to sequences.
- Taking ranges/subsequences from a sequence.
- Removing elements from a sequence.

3.1 Survey Questions

Although these two sources do give some idea on how PVL is used, a survey was conducted to get direct input from users on desired functionality. The survey below has been sent to two groups;

[1]Although list/sequence comprehension is an interesting feature, realizing this feature is not straightforward. Mainly due to the difficulty of mapping values to indices of the resulting sequence, it was decided to drop this feature since it does not make sense in the context of a language without a runtime.

the VerCors team (consisting of users and developers of VerCors) and students who have used VerCors/PVL in the past. The survey questions can be found in Figure 3.1.

The last question was preceded with a list of features that were already determined to be implemented from the previous sources.

1. Have you used PVL before?
2. Which of the currently supported ADTs have you used before?
3. Were there any auxiliary functions you wrote for those ADTs which could be used in general?
4. Was there ever a point where you were looking for a specific ADT which was not supported? Did you instead model the problem using a supported ADT? If so, could you explain what you wanted to model and how you solved it?
5. What new functionality/ADT would help you with your software verification in PVL?

Figure 3.1: Survey questions

3.2 Survey Results

The results of the survey can be found in Appendix A. In total there were 9 responses. From these results, it can be concluded that all responses are from users who have used PVL before (based on question 1). From the 9 responses, all have used sequences, 4 have used bags and 4 have used sets (based on question 2). The responses from the last three questions are combined into the list below with a small discussion on the feature itself.

- **A `get` function for sequences**: In the time since the survey, this issue has been solved by rewriting the grammar. All cases that did not work before (e.g. `seq<int> {1, 2, 3}[0]`) are now properly supported by the syntax.
- **Checking if a sequence is a permutation of another**: In the VerCors ADT wiki, a suggestion was made to have the <= operator defined on sequences as $\subseteq$ and $<$ to be defined as $\subset$. Using these operators, a permutation operator can be easily defined (either by the user or in VerCors itself).
- **Summing over sequences**: Summing over a sequence requires the sequence to have elements that can be summed. Currently, the only types that can be summed natively are integers and permissions. For a sequence of integers, the binding expression `\sum` can be used to sum elements between two bounds.

  Having such a function for a generic type `T` is interesting if the function takes a (pure) function as its argument that performs the summing. Using this function, any type could be "summed" if a function can be defined summing the two elements of type `T`.
- **Min/max functions over bags**: A min/max function over bags requires the bag to have orderable elements of some type. The previous discussion is also applicable here were a general function min/max should be defined that takes a comparison function as its argument.
- **Maps**: Maps are already part of the list of features to research and implement.

- **Pairs, Triples, Tuples**: Tuples of any size is an interesting general-purpose ADT if the elements have a generic type.
- **Custom ADTs**: Instead of implementing different ADTs, a custom ADT can be defined by the users. In another discussion, a suggestion by Marieke Huisman was to have generic classes/functions. After some thought on the possibility of having generics in VerCors, it was decided to implement it.
- **Sequences with per element permission**: If we have interpreted the suggestion correctly, it is already part of VerCors. Besides having permissions on the sequence itself, permissions can be specified for the elements (in the case of objects).
- **Simple constructors for sets, bags, sequences and maps**: This feature is already planned to be implemented.
- **MCRL2 support**: If a mapping could be made between MCRL2 functions and COL functions, this feature could be possible. However, this feature is out of scope for this thesis mainly due to time constraints.
- **Multidimensional arrays**: This feature is already part of PVL.
- **Higher-order functions**: The examples given are the `map` and `fold` function. For these functions to be possible/to make sense within PVL, it should be possible to give a function argument (that maps or folds the elements) to these functions.

3.3 List of Features to Implement

From the VerCors ADT wiki, the VerCors example directory and the survey, a list of features to implement is collected (see Figure 3.2).

- Appending and prepending values to sequences.
- Taking ranges/subsequences from a sequence.
- Removing elements from a sequence.
- Simple syntax for sequence, set and bag creation. For example, instead of `seq<int> {a, b, c}` we would have `[a, b, c]`.
- Subset notation for sets and bags.
- Set comprehension.
- Maps with basic operations.
- Generic classes and functions.

Figure 3.2: Features to implement

Chapter 4

Rephrasing the Research Questions

In Chapter 3, the first research question was answered resulting in a list of features to implement. The second research question is rephrased to be more concrete based on that list.

As a reminder, the two general research questions and the list of features to implement can be found in Figure 4.1. The implementation of these features is discussed in two parts: implementation-focused and design-focused. The first five features are relatively small compared to the other three features. For the first five features, Chapter 7 focuses on the implementation of these features. For the remaining features, Chapters 8, 9 and 10 focus more on the design of these features.

RQ1 What ADTs or functionality on ADTs is desired from VerCors/PVL?

RQ2 How can the architecture of VerCors and the back end Viper support the functionality of RQ1?

1. Appending and prepending values to sequences, sets and bags.
2. Taking ranges/subsequences from a sequence.
3. Removing elements from a sequence.
4. Simple syntax for sequence, set and bag creation. For example, instead of `seq<int> {a, b, c}` we would have `[a, b, c]`.
5. Subset notation for sets and bags.
6. Set comprehension.
7. Maps with basic operations.
8. Generic classes and functions.

Figure 4.1: The general research questions and the list of features to implement

The research questions are rephrased as follows:

RQ3 For features number 1 to 5:

- **RQ3.1** What is the definition of the functionality?
- **RQ3.2** Which of the five approaches in Chapter 6 can be applied to implement the functionality?

RQ4 For set comprehension:

- **RQ4.1** What is the definition of set comprehension?
- **RQ4.2** How is set comprehension encoded into Viper?

RQ5 For maps:

- **RQ5.1** What is the definition of a map?
- **RQ5.2** What operations are defined on maps?
- **RQ5.3** How can Viper Domains be used to implement a map?

RQ6 For generic classes/functions:

- **RQ6.1** What is the definition of generic classes/functions?
- **RQ6.2** How should type checking work for generics classes/functions?
- **RQ6.3** How does the verification of generic classes/functions work?

Chapter 5

Implementation-level View of VerCors

This chapter goes over the three general parts of the architecture of VerCors introduced in Chapter 2. An implementation-level view is given of each part, explaining the role of different classes and giving concrete examples of the code and COL ASTs where useful.

5.1 The Initial Transformation: From Input Language to a COL AST

The input language parsers/tree visitors are named using a convention where the input language is appended with the string `toCOL`. For example, the parser for Java 8 is named `Java8JMLtoCOL.java` and for PVL it is called `PVLtoCOL.java`. Some additional parsing might be needed for some languages, however in general a single parser is sufficient. Since this paper is focused on PVL, we refer solely to `PVLtoCOL.java` (or simply `PVLtoCOL`) and `ANTLRtoCOL.java` which is a parser that covers language constructs common among the support input languages.

In `PVLtoCOL`, the ANTLR parsed AST for PVL is walked and each node is transformed into a COL *AST node* using an AST node factory. For example, if the function `head` is used in the input file (which returns the head of a sequence)[1], it is matched in the tree visitor and transformed to a COL expression with the corresponding operand and its arguments.

There are several approaches used to match the syntax:

- The `match` method
- The `PVLSyntax` class
- Iterating over all defined syntax

The `match` method is defined in `ANTLRtoCOL`. It takes at least one argument `ctx` of type `ParseRuleContext`. The ANTLR class `ParseRuleContext` contains all information on the current grammar rule that is matched in the input file. For example, if `a + b` is parsed, `ctx` will be a corresponding `ParseRuleContext` with references to `a` and `b`. The other arguments are Java strings that need to be matched against. These Java strings are either strings to be matched against exactly or `null` to match anything.

[1]The syntax for the head function is `head(xs)` where `xs` is a sequence.

Listing 5.1 shows an example of how the match method is used in PVLtoCOL to match the logical negation operator !. The arity of this operator (i.e. the number of arguments it takes) is one. To match the operator itself, "!" is given as an argument to the match method and null to match the argument of the operator. When the ! operator is matched, the AST node factory create is used to make a COL expression with operator Not and its argument. The corresponding operator is defined in a Java enum StandardOperator.java and these operators have an attribute arity which is the number of arguments the operator takes. In the case of the Not operator the arity is 1. The argument of the operator is accessed using the convert method.

```
  //From the context of the expression we are visiting,
  // try to match the operator "!"
  if (match(ctx,"!",null)){
    return create.expression(StandardOperator.Not,convert(ctx,1));
  }
```

Listing 5.1: Example using the match method

Another example is the construction of a sequence. Creating a new sequence in PVL initialized with some values is done using the following syntax: seq<TYPE> { VALUES }. The variables in this syntax are the type of the sequence and the (comma-separated) values. The if statement (in PVLtoCOL) that matches this syntax and transforms it into COL can be seen in Listing 5.2.

```
//Match the syntax with the type and arbitrary values
if (match(ctx,"seq","<",null,">",null)){
  //From the child 2 (the type of the sequence), get an equivalent type to be used internally
  Type t=checkType(convert(ctx,2));
  //Visit the arguments and store the result in the args array
  ASTNode args[]=convert_list((ParserRuleContext)ctx.getChild(4),"{",",","}");
  return create.struct_value(create.primitive_type(PrimitiveSort.Sequence,t),null,args);
}
```

Listing 5.2: Another example using the match method

In this case, the tokens seq, < and > are matched and the variable parts of the syntax are left null. The first null value is interpreted as the type and the second null value is interpreted as the comma-separated values of the sequence. Using the AST node factory create, a struct value (i.e. a representation of a structured value in COL) is constructed, defining it as a sequence with the given type t and values args.

The second approach to match syntax is using methods in the Java class PVLSyntax. This class specifies a mapping between StandardOperators and the concrete syntax. For example, the Not operator (used in the example above) has a mapping to the concrete syntax "!". PVLSyntax has two methods to parse a given string named parseFunction and parseOperator. For a given string, these methods return the corresponding StandardOperator if there is a mapping, else they return null. For functions, PVLtoCOL tries to match a function invokation using the parseFunction method. This means that if a new function is added to VerCors, a mapping needs to be defined in PVLSyntax and the parsing is handled in PVLtoCOL.

The last approach is to iterate over all defined StandardOperators. This approach is implemented in the Java class ANTLRtoCOL, a general-purpose parser which is also used by parsers for different languages such as C and Java. This class is the superclass of PVLtoCOL and tries its approach if the previous approaches do not succeed. ANTLRtoCOL iterates over all (defined) StandardOperators, retrieves the syntax from PVLtoCOL and tries to match it. If the syntax is matched, a COL expression is made with the matched operator and the provided arguments.

5.2 COL

COL is a language used internally by VerCors and is encoded as an abstract syntax tree. Using this AST, the input file is internally represented.

All nodes of the COL AST are subclasses of the abstract Java class `ASTNode`. This Java class contains all common properties of AST nodes such as the type of the AST node and the parent of the node. There is a variety of subclasses of `ASTNode`. For example, a COL method is encoded using the subclass `Method`, a COL contract is encoded using the subclass `Contract` and a COL expression with an operator is encoded using the subclass `OperatorExpression`. All of these classes extend (directly or indirectly) the abstract class `ASTNode`. Since there are a lot of different AST nodes, relevant AST nodes are introduced where necessary.

Since ASTs are trees, a simple visual representation of the COL ASTs can be drawn. For example, the visual representation of the simple PVL program in Listing 5.3 can be seen in Figure 5.1. It can be seen that a simple program can lead to a (visually) large AST. To improve readability, COL ASTs in the rest of the thesis are simplified by drawing only the relevant parts of the AST.

```
class Example {
  requires a >= 0;
  ensures \result == a + 1;
  int incr(int a) {
    return a + 1;
  }
}
```

Listing 5.3: A simple example PVL program

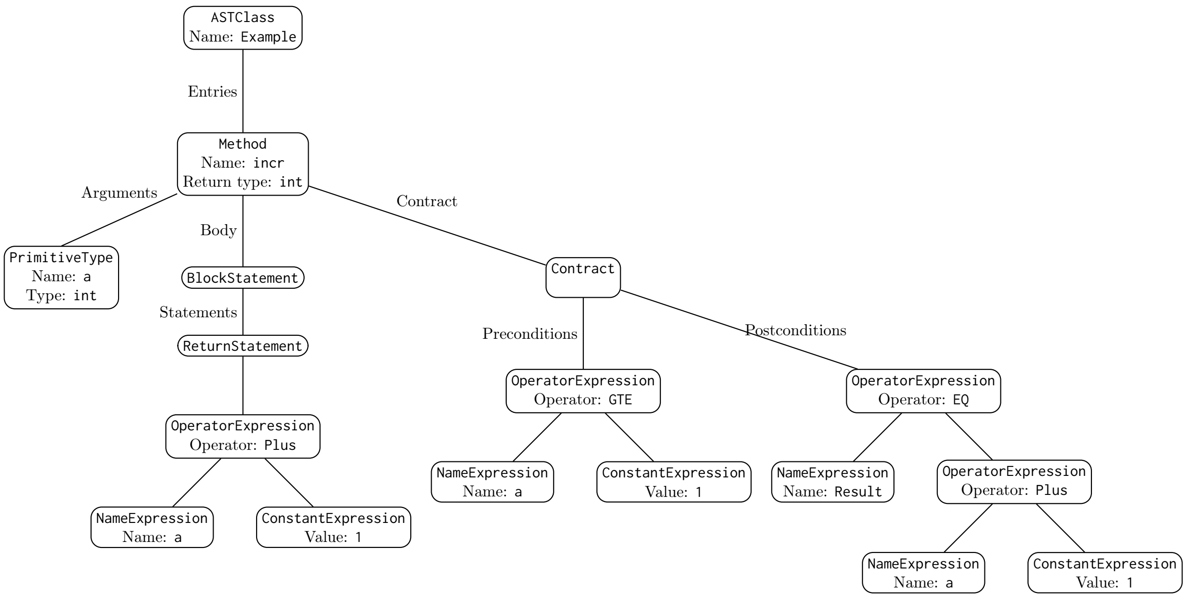


Figure 5.1: A visual representation of the PVL program in Listing 5.3

5.3 Passes

The result of the initial step is a COL AST. This COL AST is processed and transformed numerous times into (again) a COL AST by different passes. In essence, passes are tree visitors implementing the interface `ASTVisitor`.

When looking at the general behavior of a pass, two types of passes can be distinguished: *rewrite* passes and *visitor* passes.

Rewrite passes

Rewrite passes are tree visitors that rewrite the entire COL AST. These passes (which are Java classes) extend the `AbstractRewriter` class. The implementation of `AbstractRewriter` consists of a collection of methods named `visit` with different arguments. These arguments correspond to different COL AST nodes such as `OperatorExpressions` and `Methods`. When walking the COL AST, the `visit` method with the argument matching our current node is called.

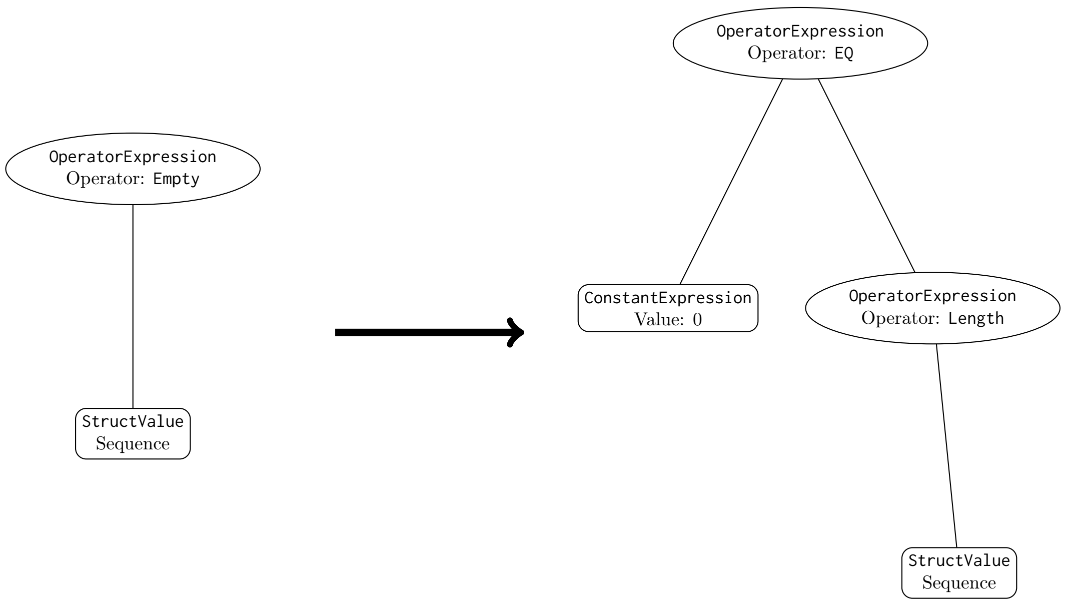
The implementation of `AbstractRewriter` copies the COL AST into a new COL AST without changing any nodes in the tree. Rewrite passes extend `AbstractRewriter` and therefore copy all AST nodes by default. If an AST node needs to be transformed, the corresponding `visit` method is overridden in the rewrite pass to implement the desired transformation.

For clarification, let us look at the following example. Suppose there is a new operator `Empty` with arity 1 that returns true if the length of the argument is zero. This new operator is not part of the core COL language or in other words it cannot be mapped to the back end. A rewrite pass is used to transform this operator into an equivalent expression using COL AST nodes that do have a mapping.

In the initial step, the syntax is transformed into a COL expression, that is `create.expression(StandardOperator.Empty, sequence)`. These types of COL expressions are encoded as `ASTNodes` of the subtype `OperatorExpression`, thus our pass overwrites the `visit` method with argument of type `OperatorExpression`. Since there are more than one operands defined in `StandardOperator`, the expression with operand `Empty` is matched and transformed into a COL expression of the form $0 == |sequence|$ (see Listing 5.4). A visual representation of the transformation can be found in Figure 5.2.

```
public void visit(OperatorExpression e){
    if (e.operator().equals(StandardOperator.Empty)) {
        result = create.expresson(StandardOperator.EQ, constant(0), length(seq));
    }
}
```

Listing 5.4: Matching and transforming the `Empty` operator



(a) Before rewrite (b) After rewrite

Figure 5.2: Rewriting the `Empty` operator to an equivalent COL expression

Visitor passes

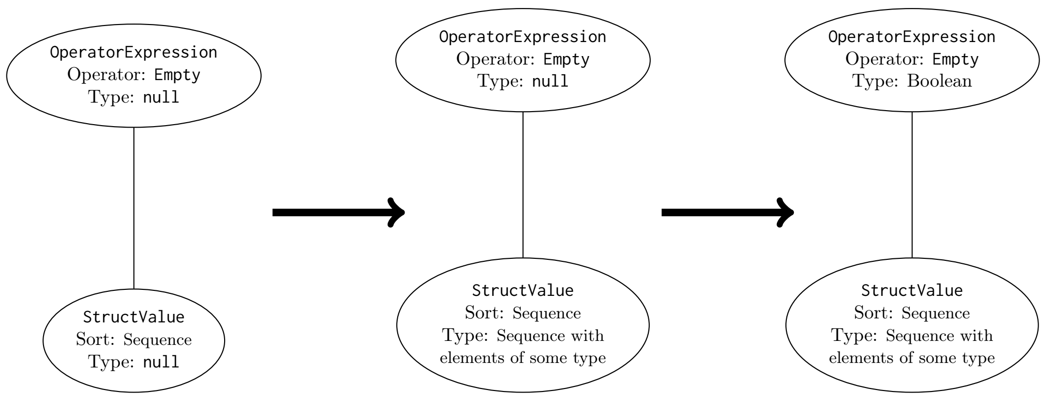
As the name suggests, visitor passes are tree visitors that visit AST nodes in contrast to rewrite passes that rewrite/copy AST nodes. Similar to rewrite passes, visitor passes also have `visit` methods for all different kinds of AST nodes. These passes extend the `RecursiveVisitor` class that by default visits all nodes.

For instance, one of the uses for visitor passes is type checking. The class responsible for type checking the COL AST is called `AbstractTypeCheck`. This pass has two functions: type checking and setting the type. Let us take the `Empty` operator again as our example. A COL expression is encoded as an `ASTNode` of type `OperatorExpression`, thus the `visit` method with the argument of type `OperatorExpression` is overwritten. A simplified version of the type check for the operand `Empty` can be seen in Listing 5.5. The argument of `Empty` has to be a sequence and the result is of type Boolean. A visual representation can be seen in Figure 5.3.

```
public void visit(OperatorExpression e){
  switch(e.operator()) {
    case Empty: {
      if (!argumentOfEmpty.isPrimitive(Sequence)) {
        Fail("argument␣of␣empty␣not␣a␣sequence"); // Internal method to signify a fail.
      }
      e.setType(Boolean);
      break;
    }
  }
}
```

Listing 5.5: Type checking the `Empty` operator

Note here that the type of `e`, the AST node corresponding to the COL expression with operator `Empty`, is set to the appropriate value. The fact that the type checking pass is a visitor pass does not imply that COL AST cannot change. Visitor passes can set attributes of the AST node, however they cannot transform or replace it entirely.



(a) Initial COL AST (b) After the type check for the `StructValue` (c) After the type check for the `Empty` operator expression

Figure 5.3: An example of a visitor pass

Besides the examples of passes given above, there are numerous visitor and rewrite passes defined in VerCors for different purposes. Examples of visitor passes are type checkers, scanners that check if a certain feature is used and visitors that collect all predicates. Examples of rewrite passes are a pass that rewrites methods marked as pure into functions and a pass that propagates invariants on a method to loops inside the method. From the examples, we can see that passes have a specific purpose or are related to a specific feature. These passes are related to features that are not relevant for this thesis since they are not related to ADTs, so they are not discussed individually.

5.4 COL AST to Viper

The final transformation of the COL AST to a Viper problem is implemented using a pass. This type of pass is called a *validation* pass and it is different from the rewrite and visitor passes in that it does not change the COL AST.

Validation passes are implementations of the abstract Java class `ValidationPass`. `ValidationPass` has two methods named `apply` and `apply_pass` that are used to apply that pass to the current COL AST. The `apply` method starts the verification back end returning a `TestReport`, a Java class containing information on whether the verification has passed or not and on exceptions that were possibly thrown. The `apply_pass` method applies the pass by calling the `apply` method and returning a `PassReport` based on the return value of the `apply` method. `PassReport` is a Java class containing information on the pass such as the arguments given to the pass, the output of the pass and whether there was a fatal error.

The validation pass for the Viper back end requires some setup compared to the default implementation of `apply_pass` (that does no setup), thus the `apply_pass` method is overwritten in `SilverBackend`. The `SilverBackend` performs the following three steps:

1. An instance of the Viper verifier is retrieved. VerCors has its own interface to setup and

retrieve an instance of the Viper verifier. During this setup, any flags or options passed onto the Viper back end are set such as the chosen back end and arguments for Z3.

2. The COL AST is converted into a Silver program.
3. The Silver program is passed onto the verifier. The Silver program is modeled in the Viper API by the Scala class `Program`.

The conversion of the COL AST to a Silver program happens in the Java class `VerCorsProgramFactory`. This class, as the name suggests, is a factory for `Program`s in VerCors. For the Viper back end, the `Program` is a class modeling a Silver program. The conversion is based on the mapping defined in the following classes:

- `SilverTypeMap`: Mapping of COL types to Silver types. Implementation of the `TypeMapping` class.
- `SilverExpressionMap`: Mapping COL expression to Silver expressions. Implementation of the `ASTMapping` class.
- `SilverStatementMap`: Mapping COL statements to Silver statements. Implementation of the `ASTMapping` class.

`TypeMapping` and `ASTMapping` match parts of the final COL AST and use the class `SilverExpressionFactory` to get equivalent Silver expressions. These Silver expressions are defined in the Silver project.

Chapter 6

Approaches to Implementing Functionality

This chapter goes over five general approaches to implement a feature in VerCors. These approaches are used as a basis for the implementation work of later chapters.

There are multiple approaches one can take to implement a feature in VerCors. To categorize these implementations, we discuss general approaches in detail. Five general approaches have been identified to implement a new feature:

- Pure syntactic sugar
- Transformed syntactic sugar
- Mapping Directly to Back End
- Function Generation
- Domains

The list is ordered such that the first approach is the simplest and least involved. As the list goes on, it becomes more involved.

6.1 Syntactic Sugar

There does not seem to be a widely used, formal definition of syntactic sugar. The simplest and most fitting description in the context of VerCors was found in [30]:

> *Many languages use syntactic sugar to define parts of their surface language in terms of a smaller core.*

This description is applicable in the context of VerCors with the surface language being PVL and the smaller core being the core language of COL that is mapped to the back end.

On an implementation level, a distinction can be made between two types of syntactic sugar: *pure* syntactic sugar and *transformed* syntactic sugar.

6.1.1 Pure syntactic sugar

We define pure syntactic sugar in the context of VerCors as follows:

> *Pure syntactic sugar is syntax that is transformed into an existing COL expression during the initial step from PVL to the COL AST*[1].

The definition entails the following; Firstly, the functionality expressed by the syntactic sugar is already part of VerCors. Secondly, the only transformation required is the initial transformation from PVL to the COL AST. All other transformations on the COL AST regarding this functionality have to already be part of VerCors.

The second condition for the transformation comes from the limited information about the type of structures during this initial step. Let us look at the example in Listing 6.1. In this example, we define a new sequence `a` and initialize it with some values. The right side of the assignment is a sequence with elements of type `MyClass` and at a later stage it will be checked if all values are of this type. In this case, a COL sequence can be constructed with elements of type `MyClass` since the type is provided.

```
seq<MyClass> a = seq<MyClass> {AInstanceOfMyClass, BInstanceOfMyClass, CInstanceOfMyClass};
```

Listing 6.1: Sequence initialization in PVL

To conclude, pure syntactic sugar can only be used if the feature already exists and all information needed for the transformation is available during the first step.

To implement a feature using pure syntactic sugar, the following steps are taken:

1. The new syntax has to be defined in the ANTLR grammar file.
2. The PVL syntax is transformed into an equivalent COL expression in `PVLtoCOL`.

6.1.2 Transformed syntactic sugar

Transformed syntactic sugar is used in cases where not all information is available during the initial step. Before defining transformed syntactic sugar, let us look at a case in Listing 6.2 where pure syntactic sugar is not applicable.

```
seq<int> a = seq<int> {someIntA, someIntB, someIntC};
seq<int> b = someIntD :: a;
```

Listing 6.2: Prepending a value to a sequence

We have a sequence of integers `a` initialized to a sequence with some values. Since the type is given, we can construct a sequence with elements of type `int`. In the next line, a sequence `b` is initialized to the value `someIntD::a` which should be read as prepend the variable `someIntD` to the sequence `a`[2]. This should eventually be transformed into an expression stating that a sequence with the value `someIntD` should be concatenated with sequence `a`, however there is one missing piece of information, the types of the variable `someIntD` and `a` are not known during the initial transformation. In the COL AST, these variables will be the AST nodes of type `NameExpression` without a type. Since the type of the resulting sequence is not known, the sequence cannot be constructed.

[1]As a reminder, the back-end of VerCors could be split up into three steps: 1. PVL to COL AST, 2. transformations on the COL AST and 3. COL AST to Viper.

[2]Currently this syntax does not exist (yet) in PVL.

Cases such as this cannot be implemented using pure syntactic sugar since some information is missing during the initial step. The missing information might be available during the next step which is the passes. In cases where the information is available at a later step, transformed syntactic sugar is applicable.

We define transformed syntactic sugar in the context of VerCors as follows:

> *Transformed syntactic sugar is syntax that is transformed into an existing COL expression during the transformation step from PVL to the COL AST and at least one rewrite pass.*

Again, this definition entails the following; Firstly, the functionality expressed by the syntactic sugar is already part of VerCors or better said, the functionality is expressible in the COL language. Secondly, in addition to the initial transformation (which is always required), at least one rewrite pass is defined that performs a transformation on parts of the COL AST related to that feature.

Pure and transformed syntactic sugar are the same approach from a design point-of-view since both express a new feature using existing features. The difference between them is that from an implementation point-of-view different steps are taken to apply them, hence they are discussed separately.

To implement a feature using transformed syntactic sugar, the following steps are taken:

1. The new syntax has to be added to the ANTLR grammar file.
2. A new operator is defined in the enum class `StandardOperator`. The new operator has an arity that is the number of arguments of the operator.
3. The PVL syntax related to the new feature is transformed into a COL expression with the new operator and the arguments it takes.
4. A rewrite pass is defined (or an existing rewrite pass is extended) to perform the transformation.

For clarification, let us look at the example of prepending a value to a sequence. As a reminder, pure syntactic sugar is not applicable due to insufficient information during the initial step, thus transformed syntactic sugar is applied. Firstly, a new operator `PrependSingle` is defined in the enum class `StandardOperator` (see Listing 6.3). Since this operator is new, it does not have a mapping to Viper and so it is mapped to an expression of the core language of COL. The syntax for the functionality is matched in `PVLtoCOL` and transformed into a COL expression with operator `PrependSingle` and its arguments (the value to prepend and the sequence to prepend to, see Listing 6.4).

In this specific case, an existing rewrite pass named `Standardize` is extended instead of defining a new rewrite pass (see Listing 6.5). Since the COL expression is an expression with an operator, it is encoded in the COL AST as an `OperatorExpression`. Thus we override the visit method with an `OperatorExpression` as its argument. When we visit an `OperatorExpression` with operator `PrependSingle`, the arguments themselves are visited using the same rewrite pass. Using an AST node factory, a new COL expression is returned with operator `Append` which concatenates the two provided arguments. The first sequence here is a newly constructed sequence with the value to prepend as its only value and the second sequence is the sequence that is prepended to. A visual representation of this rewrite/transformation can be seen in Figure 6.1.

```java
public enum StandardOperator {
    ..., PrependSingle(2), ...
}
```

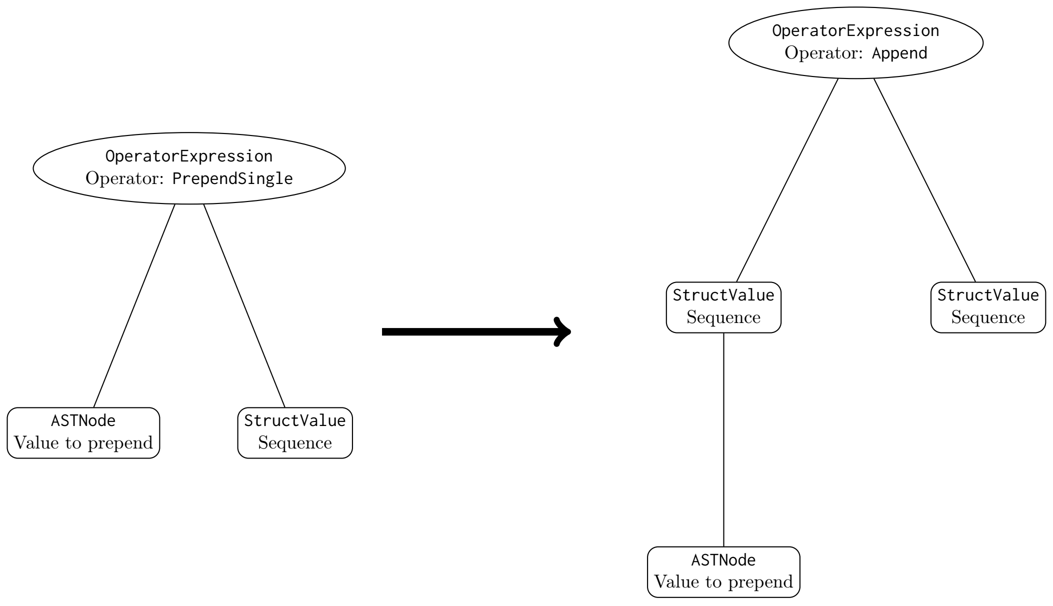
Listing 6.3: The enum class `StandardOperator` with a newly defined operator `PrependSingle`

```java
create.expression(StandardOperator.PrependSingle, arguments);
```

Listing 6.4: The construction of a COL expression with operator `StandardOperator.PrependSingle` with its arguments. The variable `create` is the AST node factory

```java
public class Standardize extends AbstractRewriter {
  @Override
  public void visit(OperatorExpression e){
    if (e.operator().equals(StandardOperator.PrependSingle)) {
      Type seqElementType = e.arg(0).getType();
      ASTNode var = e.arg(0).apply(this); // Visit the first argument, which is the value to prepend
      ASTNode seq = e.arg(1).apply(this); // Visit the second argument, which is the sequence to prepend
           ↪ to

      StructValue newSeq = create.struct_value(create.primitive_type(PrimitiveSort.Sequence,
           ↪ seqElementType), null, var);
      result = create.expression(StandardOperator.Append, newSeq, seq);
    }
  }
}
```

Listing 6.5: A part of the rewrite pass `Standardize`



(a) Before rewrite (b) After rewrite

Figure 6.1: A visual representation of the rewrite/transformation

It should be noted that there is one more step that is possibly required. As said before, when choosing between pure and transformed syntactic sugar, the difference was in the information available during the initial step.

In the example above, it is assumed that the type is known, now let us briefly look at where the type comes from. The type of the COL AST nodes is set during the type checking pass which is a visitor pass. Later passes can use this type information if needed. However, there is a caveat that lies in the fact that rewrite passes copy the entire COL AST by default without copying any additional attributes. Thus the type information and most other attributes set by a visitor or rewrite pass are omitted. This is intended behavior since it could be that as a result of a transformation the attributes set do not hold anymore, for example after a transformation it could be that the type of a part of the AST does not match anymore. Thus a prerequisite to a rewrite pass which requires the type of some AST node is the type checking pass without any other rewrite passes in between.

6.2 Mapping Directly to Back End

Mapping functionality directly to the back end is an option if the functionality is already present in the back end. This approach is very similar to the previous two approaches. The main difference is in the last step which is mapping the operator to the back end. In this section, the approach is explained with Viper as the specific back end for two reasons: explaining the approach using a concrete example helps in understanding the approach instead of an abstract explanation and Viper is chosen since it is the main back end of VerCors. This approach is the same for any other back end with the only difference in the classes where the mappings are defined.

The steps taken to map directly to the back end are as follows:

1. The new syntax has to be added to the ANTLR grammar file.
2. A new operator is defined in the enum class `StandardOperator`. The new operator has an arity which is the number of arguments of the operator.
3. The PVL syntax related to the new feature is transformed into a COL expression with the new operator and the arguments it takes.
4. A mapping is defined from the new `StandardOperator` into Silver.

The first three steps are the same steps from the previous approaches: The syntax is defined in the ANTLR grammar for PVL, a new operator is added to `StandardOperator` and the PVL syntax is matched and transformed into a COL expression with the new operator and the arguments it takes.

As explained in Section 5.4, the mapping of COL AST nodes to Silver is performed by a special pass (of type `ValidationPass`). Inside of an anonymous `ValidationPass`[3], the class `SilverBackend` is used to transform the COL AST into Silver and call the Viper toolset to verify the problem. `SilverBackend` already defines the logic to go through the COL AST and transform it to Silver based on the three mapping classes `SilverTypeMap`, `SilverExpressionMap` and `SilverStatementMap` (also mentioned in Section 5.4). In order to define a new mapping, the newly defined `StandardOperator` is matched in the appropriate mapping class and transformed into a Silver AST node (provided by the Silver project).

For clarification, let us look at the mapping of the size operator in PVL. In COL, the size operator is an expression with a single argument with the type of the argument either a sequence,

[3]An anonymous instance of a class is a local class without a name.

set or a bag encoded as the COL AST node `OperatorExpression`. In Silver, size is one of two expressions for different types of arguments, for sequences it is named `SeqLength`,and for sets and bags it is named `AnySetCardinality`. `SeqLength` and `AnySetCardinality` are part of the Silver project.

Listings 6.6 and 6.7 show the code implementing this transformation. In Listing 6.6, the `OperatorExpression` with operator `Size` is matched (line 5) and using the `SilverExpressionFactory` named `create` a Silver expression with operator size is returned (line 6). The Silver expression that is returned is dependent on the type of the argument of the `size` operator. In Listing 6.7, the type of the argument is matched and the correct Silver expression is returned (lines 4-6).

```
 1  public class SilverExpressionMap implements ASTMapping {
 2    public Exp map(OperatorExpression e) {
 3      switch(e.operator()){
 4        ...
 5        case Size:                    \\ Match the COL operator Size
 6          return create.size(o,e1); \\ return a size expression in Silver
 7        ...
 8      }
 9    }
10  }
```

Listing 6.6: A part of the Java class `SilverExpressionMap`

```
 1  class SilverExpressionFactory {
 2    override def size(o:O,e1:Exp) :Exp = {
 3      e1.typ match {
 4        case SeqType => SeqLength(e1)
 5        case MultisetType => AnySetCardinality(e1)
 6        case SetType => AnySetCardinality(e1)
 7        case _ => throw new Error("cannot␣convert␣size␣for␣type␣"+e1.typ);
 8      }
 9    }
10  }
```

Listing 6.7: A part of the Scala class `SilverExpressionFactory`

6.3 Function Generation

The previous approaches are used for relatively simple functionality. When functionality becomes more complex, function generation becomes a viable option. Function generation is *generating a function to perform the needed functionality.* The approach to generating a function is similar to transformed syntactic sugar in the steps taken to implement it. The steps taken for function generation are as follows:

1. The syntax related to the functionality needs to be defined.
2. A new operator is defined in enum class `StandardOperator`.
3. The syntax is matched in `PVLtoCOL` and transformed into a COL expression with the newly defined operator and its arguments.
4. A rewrite pass is defined (or an existing one is used) that generates the related function.

The difference with transformed syntactic sugar is in the pass that is defined. By default, a rewrite pass only walks and rewrites the COL AST. For this approach, a rewrite pass performs an extra step which is the generation of the functions. Similar to the other approaches, the `visit` method with the argument of type `OperatorExpression` is overridden. All COL expressions with the newly defined operator are rewritten to a function invokation with the provided arguments using a COL AST node factory (see Listing 6.8). A reference to the function name is kept in a static map to keep track of functions to generate.

```
colASTNodeFactory.invokation(null, null, functionName, arguments);
```

Listing 6.8: A COL representation of a function invokation with name `functionName`

Functions themselves are also encoded in the COL language (using `ASTNodes` such as `Method` and `Contract`). Functions can be broken down into five parts: The function name, the return type, the arguments, the contract and the body. From the functionality to be implemented, the name, return type, arguments follow naturally. The contract of the function is encoded using COL expressions (obtained from a COL AST node factory). The example in Listing 6.9 is a COL expression expressing that a variable `i` is within the boundaries of a certain sequence (i.e. $0 <= i \ \&\& \ i < |sequence|$).

```
create.expression(And,
  create.expression(LTE, constant(0), "i"),
  create.expression(LT, "i", size(sequence))
)
```

Listing 6.9: A precondition encoded in the COL language describing the validity of an index `i` for a given sequence

The body of the function does not have to be defined. Bodyless functions have two benefits. The first one is the most obvious, which is that only the behavior of the functionality has to be specified which saves work in implementing the functionality and secondly the Viper back end does not prove that the function is correct every time an attempt is made to verify something. The preconditions of the function are checked where the function is invoked and the postconditions are assumed on the result of the function.

When generating functions for ADTs in Viper, a new function has to be generated for each ADT if the type of the ADT differs. This is a result of the lack of generic types for functions in the Viper back end. For example, if we generate a function `foo` with a sequence argument and the function `foo` is invoked using an integer sequence and a boolean sequence, then two functions are generated; one for the integer sequence and one for the boolean sequence.

Details of the implementation depend on the generated function. A more detailed look at the implementation work is discussed for a feature using function generation (for a concrete example, see Section 7.3).

6.4 Domains

If the new functionality is a new ADT, Silver domains are the best option. Given an ADT expressed as a domain in Silver, the steps to support the ADT in VerCors are as follows:

1. The syntax related to the functionality is defined.
2. The new ADT is added as a `PrimitiveType`.

3. Operators are defined in the enum class `StandardOperator` for all domain functions that need to be supported.
4. The syntax for all functions is matched in `PVLtoCOL` and transformed into COL expressions with the newly defined operator and its arguments.
5. The Silver domain is added to the file `prelude.sil`.
6. The domain is added to the COL AST.
7. The new operators are transformed into invokations of the domain functions.

As an example, the domain `Tuple` (from Section 2.1.6) is implemented along with its three functions. The three functions `constructor`, `fst` and `snd` will be supported as functions named `tuple`, `fst` and `snd` respectively.

The syntax that we need to define is twofold: the syntax for the type and syntax for the functions. The syntax for the type is defined in the ANTLR grammar (see Listing 6.10). The type is encoded as a `PrimitiveType`. `PrimitiveType` models a type such as an integer, an array or a set where the actual type is specified by a `PrimitiveSort`. `PrimitiveSort` is a Java enum with values such as `Integer`, `Array` and `Set` to specify what the sort of a `PrimitiveType` is. For the new tuple type, a value `Tuple` is added to the Java enum `PrimitiveSort`. Listing 6.11 shows the syntax for the type being matched in `PVLtoCOL` and transformed into a `PrimitiveType` with sort `Tuple`.

Three `StandardOperators` named `TupleConstructor`, `TupleFst` and `TupleSnd` are defined for the three functions. The syntax for the three functions is added to `PVLSyntax` (see Listing 6.12). With these functions defined in `PVLSyntax`, the syntax for the functions are matched and transformed into `OperatorExpression` with the matching `StandardOperator`.

```
 non_array_type :
 'tuple' '<' type ',' type '>'
 ;
```

Listing 6.10: The ANTLR grammar rule for the constructor of `Tuple`

```
if (match(ctx,"tuple","<",null,",", null, ">")){
  Type t1=checkType(convert(ctx,2));
  Type t2=checkType(convert(ctx,4));
  return create.primitive_type(PrimitiveSort.Tuple,t1, t2);
}
```

Listing 6.11: Matching the `Tuple` type

The pass responsible for parsing the Silver domain and adding it to the COL AST is rewrite pass `SilverClassReduction`. This pass adds the `Tuple` ADT in two steps. First, the `StandardOperators` are matched and rewritten to function invokations. Listing 6.13 shows the transformation for the constructor function `tuple`. The `PrimitiveType` with sort `Tuple` is also rewritten. Listing 6.14 shows `Tuple` being rewritten as a class type `Tuple` corresponding to the name of the domain. The two arguments are labeled with the labels of the type parameters in the domain. These labels are used by Viper internally. A boolean flag named `tuple` is set to true to signify that the `Tuple` domain has to be loaded.

Next, the Silver domain is copied over to the file `prelude.sil`. This Silver file contains all domains and functions that are loaded in and added to the COL AST. After parsing the Silver file, we iterate over all the domains and match the domain `Tuple` by its name and add it to the COL AST.

```
public class PVLSyntax {

  private static Syntax syntax;

  public static Syntax get(){
    ...
      syntax.addFunction(TupleConstructor, "tuple");
      syntax.addFunction(TupleFst, "fst");
      syntax.addFunction(tupleSnd, "snd");
    ...
  }
}
```

Listing 6.12: The three functions for `Tuple` defined in PVLSyntax

```
public class SilverClassReduction extends AbstractRewriter {
  @Override
  public void visit(OperatorExpression e){
    switch(e.operator()){
      case TupleConstructor:{
        // Rewrite the arguments
        List<ASTNode> args = rewrite(e.argsJava());
        // Invoke the function constructor with its arguments and return type
        result = create.invokation(rewrite(e.first().getType()), null, "constructor", args);
        break;
      }
    }
  }
}
```

Listing 6.13: Transformation of the `Tuple` constructor into a function invokation

```
public class SilverClassReduction extends AbstractRewriter {
  @Override
  public void visit(PrimitiveType t){
    switch(t.sort){
      case Tuple:
        tuple = true;
        List<ASTNode> args = rewrite(((PrimitiveType)t).argsJava());
        args.get(0).addLabel(create.label("F"));
        args.get(1).addLabel(create.label("S"));
        result=create.class_type("Tuple",args);
        break;
    }
  }
}
```

Listing 6.14: Transformation of the `Tuple` type into a class type

Chapter 7

Implementation-focused Features

In this chapter, we discuss the implementation of the first five features mentioned in Chapter 3. These features are:

Section 7.2: Appending and prepending values to sequences.

Section 7.3: Taking ranges/subsequences from a sequence.

Section 7.4: Removing elements from a sequence.

Section 7.5: Simple syntax for sequence, set and bag creation.

Section 7.6: Subset notation for sets and bags.

For each of these features, we answer the two subquestions of RQ3 (from Chapter 4). The first section for each feature answers RQ3.1, showing a description of the behavior and the defined syntax. The next question answers RQ3.2 by discussing the chosen approach (from Chapter 6) and implementation details with design choices.

Before continuing with the implemented features, we (again) briefly discuss the immutability of sequences (and other ADTs) in VerCors. As stated in Section 2.2, VerCors has native support for sequences, sets and bags which are all immutable. To keep in line with existing functionality all implemented functionality on sequences is therefore pure (i.e. no side effects). The original sequence is not changed or overwritten, instead a new sequence is constructed which is the result of the functionality. In case it is required to change a sequence, the variable can be overwritten by the result of the functionality (e.g `a = <immutable function on a>`, where `a` is a sequence).
The implementation of the features described below can be found on GitHub. The repository is a fork of the VerCors repository and can be found at https://github.com/OmerSakar/vercors. It must be noted that the functionality below was not already part of VerCors. They are all implemented as a part of this research.

1. For the first 4 features, the implementation described in this thesis can be found in commit `a3e3518` which can be browsed at https://github.com/OmerSakar/vercors/tree/a3e3518a3e720a6b05359bd956312d1b96329d57.
2. For feature 5, the implementation can be found in commit `7875084` which can be browsed at (https://github.com/OmerSakar/vercors/tree/78750843b96ab4897b08e154342676a09a8c931a)

7.1 Checking if a Sequence is Empty

Description and syntax

The `isEmpty` function is defined as follows[1]:

Empty

Syntax:
`isEmpty(a)`

Arguments:
a - A sequence, set or bag.

Description:
Check if the argument `a` is empty.

Returns:
`true` if the size of the argument equals zero, else `false`.

Implementation

The `isEmpty` function is implemented using pure syntactic sugar. Since all information to express the operation in the COL language is available during the initial pass, we can avoid the more complex approaches.

The syntax for this operation is a function syntax with name `isEmpty` and a single argument. This syntax has already been defined in the ANTLR grammar file, thus no changes have to be made to the grammar. In `PVLtoCOL` the syntax of this operator is matched and transformed into the COL expression which describes the functionality of the `isEmpty` function, that COL expression being a comparison between the size of the given argument and the constant zero (i.e. `0 == size(argument)`, see Listing 7.1).

The reason for the argument `a` being either a sequence, set or bag is a consequence of using existing functionality, namely the `size` operation. The size operation is defined for sequences, sets and bags and by extension the `isEmpty` function is also defined on those types.

```
...
if (match(ctx, "isEmpty", tuple)) {
  ASTNode args[]=getTuple((ParserRuleContext)ctx.getChild(1));
  return create.expression(
    StandardOperator.EQ,
      create.constant(0),
      create.expression(StandardOperator.Size, args)
  );
}
...
```

Listing 7.1: The if block matching and rewriting the `isEmpty` function in `PVLtoCOL`

[1]This feature is not in the list of features to implement. It has been implemented as part of this research to understand the architecture of VerCors. To document the feature, it is mentioned in this thesis.

Small example

See Listing 7.2. The example constructs an empty sequence `a` and a non-empty sequence `b`. Using the `isEmpty` function, it is asserted that `a` is empty and `b` is not empty.

```
class EmptyExample {
  void main() {
    seq<int> a = seq<int> {};
    seq<int> b = seq<int> {1, 2, 4};
    assert isEmpty(a) && !isEmpty(b);
  }
}
```

Listing 7.2: Example of using the `isEmpty` function

7.2 Adding Single Values to Sequences

In the following section, two operations are introduced, namely appending and prepending single values to sequences. These two operations are introduced together since they are counterparts of each other.

Description and syntax

The `append` and `prepend` operators are defined as follows:

Append

Syntax:

```
a ++ b
```

Arguments:

a - A sequence with elements of type `T`.
b - A value of type `T`.

Description:

Construct a new sequence by adding a single value `b` to the end of a sequence `a`.

Returns:

A new sequence with all values from sequence `a` (in order) and the last value `b`.

Prepend

Syntax:

```
a :: b
```

Arguments:

a - A value of type `T`.
b - A sequence with elements of type `T`.

Description:

Construct a new sequence by adding a single value `a` to the front of a sequence `b`.

Returns:

A new sequence with all values from sequence `b` (in order) and the first value `a`.

Implementation

The `append` and `prepend` operations are implemented using transformed syntactic sugar due to two reasons. Firstly, the functionality can be easily expressed in COL thus function generation is not needed and secondly, the type of the value to add is not necessarily known during the initial transformation, meaning pure syntactic sugar is not an option.

The syntax for these operations is defined in the ANTLR grammar file for PVL named `PVFull.g4`. By default, ANTLR grammar rules are left-associative thus the `prepend` operator is explicitly defined as right-associative in order for $a :: b :: c$ to be interpreted as $a :: (b :: c)$ instead of $(a :: b) :: c$. Listing 7.3 shows the ANTLR grammar rule for `atomExpression` with the two cases for our operations. `atomExpressions` are the building blocks for expressions in PVL and thus it was deemed appropriate to define our operations as a part of this rule.

```
atomExpression
  : <assoc=right> atomExpression '::' atomExpression
  | atomExpression '++' atomExpression
  ...
;
```

Listing 7.3: The ANTLR grammar rules of the operators append and prepend

Two StandardOperators are defined for these operations namely AppendSingle and PrependSingle (see Listing 7.4). The operator could have been named Append and Prepend however the operator Append is already defined for the concatenation of two sequences and thus an alternative, more descriptive name was chosen. In PVLtoCOL, the syntax for the operations is matched in the visit method for the rule atomExpression named visitAtomExpression. Instead of using the match operator, the matching of the syntax is left to ANTLRtoCOL (the super class of PVLtoCOL). As a reminder, ANTLRtoCOL iterates over all syntax defined in PVLSyntax and tries to match it. So the syntax for append and prepend have been added to PVLSyntax where AppendSingle maps to ++ and PrependSingle maps to ::.

```
public enum StandardOperator {
  ..., AppendSingle(2), PrependSingle(2), ...
}
```

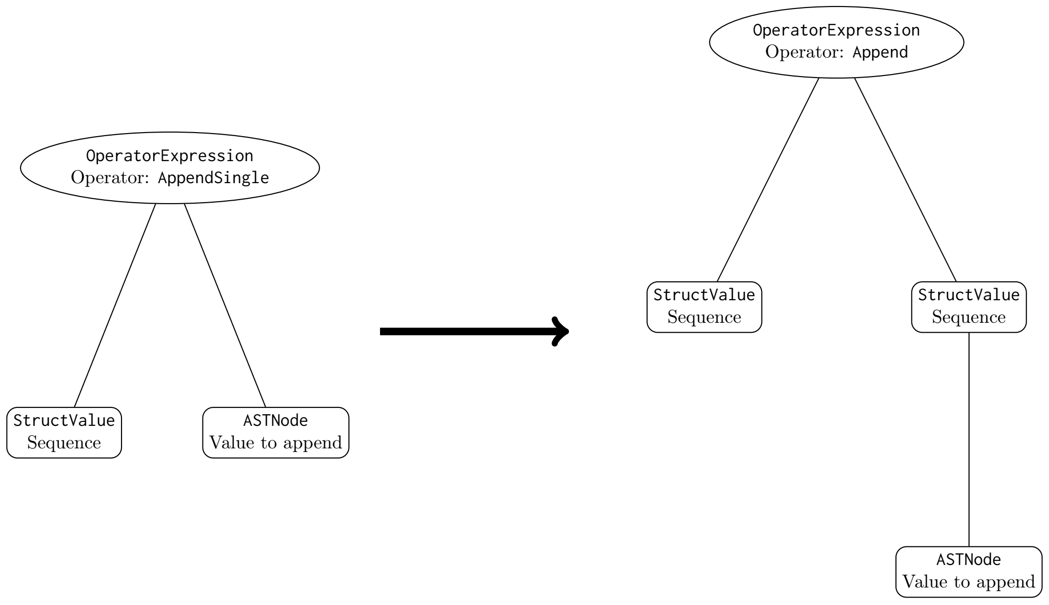
Listing 7.4: The enum class StandardOperator with operators AppendSingle and PrependSingle

Since these new operators do not have a mapping to the Viper back end, they are rewritten. The pass performing this rewrite is the rewrite pass Standardize.java. Listing 7.5 shows part of the Standardize pass which is responsible for the rewrite of the operator AppendSingle[2]. A visual representation of this rewrite can be seen in Figure 7.1. Both arguments are visited since they can be expressions themselves that need to be transformed. A new sequence is constructed with the second argument which is the value to append. This new sequence is then concatenated to the other sequence using a COL expression with operator Append. The Append operator has a mapping to the Viper back end, thus no further rewrite is necessary. It must be noted that the type check pass is a prerequisite of the Standardize pass since it requires the type of the arguments of the OperatorExpression.

```
public class Standardize extends AbstractRewriter {
  @Override
  public void visit(OperatorExpression e) {
    if (e.operator().equals(StandardOperator.AppendSingle)) {
      Type seqElementType = e.arg(1).getType(); // Get the type of the sequence
      ASTNode var=e.arg(1).apply(this); // Visit the second argument, which is the value to append
      ASTNode seq=e.arg(0).apply(this); // Visit the first argument, which is the sequence to append to

      StructValue newSeq = create.struct_value(create.primitive_type(PrimitiveSort.Sequence,
           ↪ seqElementType),null,var); //Create a new sequence with the value to be appended
      result = create.expression(StandardOperator.Append, seq, newSeq);
    }
  }
}
```

Listing 7.5: Rewriting the AppendSingle operator

[2]The rewrite for PrependSingle has been left out since it is similar to the rewrite of AppendSingle.



(a) Before rewrite (b) After rewrite

Figure 7.1: Rewrite of the `AppendSingle` operator to an expression using the existing operator `Append`

Small example

See Listing 7.6. In the example, a sequence `a` is instantiated with some values and an integer `b` is instantiated to the value 17. The first assert appends the value 20 to sequence `a` and asserts that the value has been appended. The second assert prepends the variable `b` to sequence `a` again asserting that the value of `b` has been prepended.

```
class AppendPrepend {
  void main() {
    seq<int> a = seq<int> {1, 8, 7, 5, 9};
    int b = 17;
    assert (a ++ 20) == seq<int> {1, 8, 7, 5, 9, 20};
    assert (b::a) == seq<int> {17, 1, 8, 7, 5, 9};
  }
}
```

Listing 7.6: Example of using the `append` and `prepend` operators

7.3 Taking Ranges from Sequences

In this section, three operations are introduced related to taking ranges from sequences, namely `range`, `take` and `drop`. These three operations are introduced together since they are similar.

Description and syntax

Range

Syntax:

`a[b..c]` or `range(a, b, c)`

Arguments:

a - A sequence.
b - An integer index within the range of sequence `a`.
c - An integer index within the range of sequence `a`.

Description:

Construct a new sequence using the elements of sequence `a` starting from index `b` (inclusive) to index `c` (exclusive).

Returns:

A new sequence with all values from sequence `a` between indices `b` (inclusive) and `c` (exclusive).

Take

Syntax:

`a[b..]` or `take(a, b)`

Arguments:

a - A sequence.
b - An integer index within the range of sequence `a`.

Description:

Construct a new sequence using the elements of sequence `a` starting from index `b` (inclusive) to the end of the sequence.

Returns:

A new sequence with all values from sequence `a` starting from index `b`.

Drop

Syntax:
`a[..c]` or `drop(a, c)`

Arguments:
a - A sequence.
c - An integer index within the range of sequence `a`.

Description:
Construct a new sequence using the elements of sequence `a` starting from the start of the sequence (inclusive) up to index `c` (exclusive).

Returns:
A new sequence with all values from sequence `a` up to index `c`.

Taking ranges in Viper

The Viper back end has native support for taking ranges from sequences. The Viper syntax is defined as `a[b..c]` with the same semantics as for our operators where either `b` or `c` could be omitted to either perform a `take` or `drop` operation. However, there is one major difference between our defined operators and the Viper operators which is constricting conditions for `b` and `c`. Viper does not require `b` and `c` to be valid indices. To be more precise in what Viper does, the Viper tutorial states the following:

> sub-sequence operators: `s[e1..e2]`, where `s` is a sequence and `e1` and `e2` are integers, returns a new sequence that contains the elements of `s` starting from index `e1` until (but not including) index `e2`. The values of `e1` and `e2` need not be valid indices for the sequence; for negative `e1` the operator behaves as if `e1` were equal to 0, for `e1` larger than `|s|` the empty sequence will result (and vice versa for `e2`). Optionally, either the first or the second index can be left out (leading to expressions of the form `s[..e]` and `s[e..]`), in which case only elements at the end or at the beginning of `s` are dropped, respectively.

In short, the indices given to these operators do not have to be valid and with the documentation above it is left to the users to discover what is and is not possible. Instead of having these operations loosely defined, a stricter alternative is implemented in VerCors where it is required to have valid indices .

Implementation

This feature is implemented using function generation. Both variants of syntactic sugar are not an option since it is not simple to express in one (or a few) COL expression. Defining a new domain for this functionality is more involved than using function generation.

In the definitions of the operations above it can be seen that two syntax have been defined for each operation. The reason for defining two syntax instead of one is to give the user options. One of the syntaxes is a more intuitive syntax (e.g. `a[b..c]`) and the other syntax is a function with a descriptive name (e.g. `range(a, b, c)`). The user can decide for themselves what is easier to use or what is more readable.

For the intuitive syntax, cases are added to the grammar rule `atomExpression`. Again, `atomExpressions` are the building blocks for expressions in PVL and thus it was deemed appropriate to define our operators as a part of this rule. The function syntax is already part of the ANTLR grammar. Listing 7.7 shows the ANTLR grammar rule for `atomExpression` with the three cases for our operations.

```
atomExpression
  : atomExpression '[' (atomExpression '..' | '..' atomExpression | atomExpression '..' atomExpression )']'
  ...
;
```

Listing 7.7: The ANTLR grammar rules of the operators `range`, `take` and `drop`

A new `StandardOperator` is defined for the operation `range` namely `RangeFromSeq`. Operators for `take` and `drop` already exist which currently map to the Viper syntax (see Listing 7.8). In `PVLtoCOL`, the syntax is matched using the `match` method (see Listing 7.9). The first three if statements match the function syntax and the next three if statements match the different intuitive syntaxes.

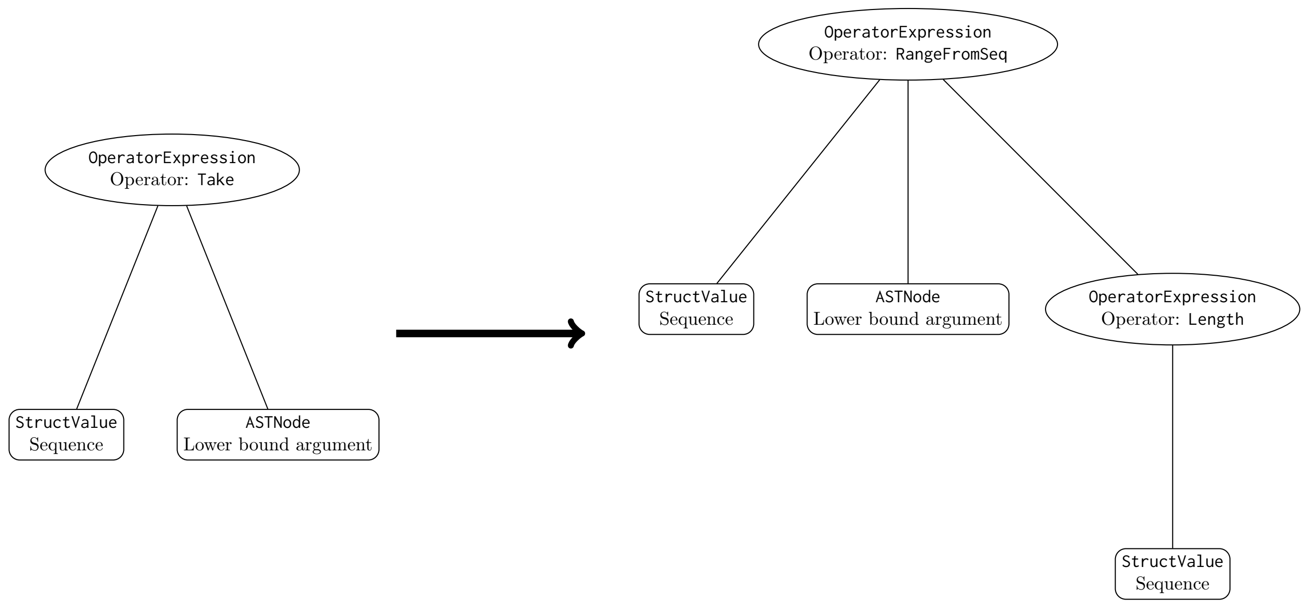
```
public enum StandardOperator {
  ..., Take(2), Drop(2), RangeFromSeq(3), ...
}
```

Listing 7.8: The enum class `StandardOperator` with operators `Take`, `Drop`, `RangeFromSeq`

```
...
if (match(ctx,"range",tuple)){
  ASTNode args[]=getTuple((ParserRuleContext)ctx.getChild(1));
  return create.expression(StandardOperator.RangeFromSeq,args);
}
if (match(ctx,"take",tuple)){
  ASTNode args[]=getTuple((ParserRuleContext)ctx.getChild(1));
  return create.expression(StandardOperator.Take,args);
}
if (match(ctx,"drop",tuple)){
  ASTNode args[]=getTuple((ParserRuleContext)ctx.getChild(1));
  return create.expression(StandardOperator.Drop,args);
}
if (match(ctx,null, "[", "..", null, "]")) {
  return create.expression(StandardOperator.Take,convert(ctx, 0), convert(ctx, 3));
} else if (match(ctx,null, "[", null, "..", "]")) {
  return create.expression(StandardOperator.Drop,convert(ctx, 0), convert(ctx, 2));
} else if (match(ctx,null, "[", null, "..", null, "]")) {
  return create.expression(StandardOperator.RangeFromSeq,convert(ctx, 0), convert(ctx, 2), convert(ctx, 4));
}
...
```

Listing 7.9: Matching the syntax of the operations `take`, `drop` and `range`

The following step is to rewrite the `take` and `drop` operators into `range` operation. It can be seen that the `take` and `drop` operators are special cases of `range` where the missing index is implicit. By providing the missing index, these operations are transformed into `range` operations. For the `take` operation, the missing index is the first index where a statement `a[..b]` is rewritten to `a[0..b]` and for the `drop` operation, the upperbound of the sequence is missing where a statement `a[b..]` is rewritten to `a[b..|a|]`. This rewrite is performed by the rewrite pass `Standardize` which has also been used in the previous section. Figure 7.2 shows a visual representation of the transformation/rewrite of the COL AST.



(a) Before rewrite (b) After rewrite

Figure 7.2: Rewrite of the `Take` operator to the operator `RangeFromSeq`

Now that the three operations have been reduced to a single operation namely `RangeFromSeq`, the function(s) can be generated. Generating the functions is done in a new rewrite pass `RewriteSequenceFunctions`[3]. This pass has two purposes, firstly identify which functions need to be generated and secondly generate the identified functions.

Similar to other rewrite passes, the COL AST is rewritten and all occurrences that need to be rewritten are matched in the `visit` method with the corresponding argument type. Since the `RangeFromSeq` operator needs to be rewritten, the `visit` method with argument of type `OperatorExpression` is overridden (see Listing 7.10). If the COL expression with operator `RangeFromSeq` is matched, the COL expression is rewritten to a COL function invokation. To invoke a function, the name and arguments of the function are required. The name of the function to invoke consists of two parts, a predefined part and a part specific to the function. In the case of the operation `range`, the predefined part is chosen to be *take_range_* and the specific part is the type of the sequence.

When the COL expression is rewritten into a COL function invokation, the type of the argument sequence is stored in a static map in the `RewriteSequenceFunctions` pass. The key of the map is the type of the argument sequence since a function only needs to be defined once for the same type of sequence. The value of the map is the name of the function to be generated based on the predefined and specific parts of the function name. The name is stored for both convenience and to avoid generating two semantically different functions under the same name.

Now onto generating the actual functions. Using the map which contains the type of the sequence on which the function operates and the name of the function to generate, the self-defined Scala method `takeRangeFromSequence` is called. Before continuing with the implementation in detail, let us first look at the function to generate.

[3]It should be noted that the `RewriteSequenceFunctions` pass is a Scala class instead of a Java class. There is no specific reason to have this pass written in Scala other than a similar pass being written in Scala. Since Scala is a JVM language, the behavior of our pass is similar to other passes.

```
override def visit(operator: OperatorExpression): Unit = {
  operator.operator match {
    case StandardOperator.RangeFromSeq =>
    val sequenceType = operator.arg(0).getType
    result = create.invokation(null, null, RewriteSequenceFunctions.getRangeFunction(sequenceType),
        ↪ rewrite(operator.args.toArray):_*)
  }
}
```

Listing 7.10: Rewriting a COL expression with operator `RangeFromSeq` to a COL function invokation

Listing 7.11 shows the PVL equivalent of the generated `take_range_sequence` function. As can be seen there are three preconditions and three postconditions. The preconditions state that the given indices should be valid and that the lower bound should be smaller or equal to the upper bound. The postconditions state that the result should be the elements from index `lowerbound` to the index `upperbound`.

```
requires 0 <= lowerbound && lowerbound < |xs|;
requires 0 <= upperbound && upperbound <= |xs|;
requires lowerbound <= upperbound;
ensures |\result| == upperbound - lowerbound;
ensures (\forall int j; lowerbound <= j && j < upperbound; \result[j - lowerbound] == xs[j]);
ensures (\forall int j; 0 <= j && j < |\result|; \result[j] == xs[j + lowerbound]);
seq<T> take_range_Sequence<T>(seq<T> xs, int lowerbound, int upperbound);
```

Listing 7.11: PVL equivalent of the generated `take_range_sequence` function

Now that we know what needs to be generated, let us look at how it is generated. As mentioned above, generating this function is done by the self-defined Scala method `takeRangeFromSequence`. The function without the pre/postconditions definitions can be seen in Listing 7.12. A `ContractBuilder` is defined by which a COL contract can be easily constructed. After the contract has been defined, a COL function declaration is created with the proper return type, contract, function name and arguments.

```
def takeRangeFromSequence(sequenceType: Type, functionName: String): ASTNode = {
  val contract = new ContractBuilder
  val result = create.reserved_name(ASTReserved.Result, sequenceType)

  val sequenceArg = new DeclarationStatement(sequenceArgName, sequenceType)
  val lowerboundArg = new DeclarationStatement(lowerboundArgName, create.primitive_type(PrimitiveSort.
      ↪ Integer))
  val upperboundArg = new DeclarationStatement(upperboundArgName, create.primitive_type(PrimitiveSort.
      ↪ Integer))

  .
  .
  .

  val declaration = create.function_decl(sequenceType, contract.getContract, functionName,
      ↪ functionArguments.toArray, null)
  declaration.setStatic(true)
  declaration // return the function declaration
}
```

Listing 7.12: Simplified version of the Scala method `takeRangeFromSequence`

The pre/postconditions are encoded as COL expressions. As mentioned in Section 6.3, the body of a function does not have to be defined, it is sufficient to express the behavior of the function in the contract. Simplified versions of the definition of one precondition and one postcondition can be seen in Listing 7.13 corresponding to the third and fifth condition in Listing 7.11. The result of this Scala method is a bodyless, static COL function expressing the `range` operation. This function is then added to the COL AST.

```
def takeRangeFromSequence(sequenceType: Type, functionName: String): ASTNode = {
  val contract = new ContractBuilder
  ...

  \\Pre: lowerbound <= upperbound;
  contract.requires(lte(lowerboundArg, upperboundArg))

  \\Post: (forall int j; lowerbound <= j && j < upperbound; result[j - lowerbound] == seq[j]);
  contract.ensures(
    create.forall(

      \\2. Condition: where lowerbound <= j and j < upperbound
      and(lte(lowerArg, someIndexJ), less(someIndexJ, upperArg))

      \\3. Statement: result[j - lowerbound] equals sequenceArg[j]
      eq(
      get(result, minus(someIndexJ, lowerArg)),
      get(sequenceArg, someIndexJ)
      ),

      \\1. Variable definition: For all integers j
      someIndexJ
    )
  )
  ...
}
```

Listing 7.13: Simplified definition of a pre and postcondition of the `range` operation.

Small example

See Listing 7.14. A sequence of integers `a` is instantiated with some values and five sequences `b`, `c`, `d`, `e` and `g` are defined as ranges from sequence `a` using the new functions and operators. A sequence of sequences of integers `f` is instatiated with some values and shown that the operators (with syntax `a[b..c]`) are equivalent to the function calls. Finally, a sequence `h` is instantiated which is a sequence of instances of the class `Object` and it is asserted that the operations also work on sequences of objects.

```
class Range {

  void main() {
    seq<int> a = seq<int> {1, 4, 5, 7, 8};

    seq<int> b = range(a, 1, 3);
    assert b == seq<int> {4, 5};

    seq<int> c = take(a, 1);
    assert c == seq<int> { 1 };

    seq<int> d = drop(a, 2);
    assert d == seq<int> { 5, 7, 8 };

    seq<int> e = a[1 .. 4];
    assert e == seq<int> {4, 5, 7};

    seq<int> g = a[2..];
    assert g[2..] == seq<int> {5, 7, 8};

    seq<seq<int>> f = seq<seq<int>> {seq<int>{1}, seq<int>{2}, seq<int>{3}, seq<int>{4}};
    assert f[0..|f|][0..|f|] == range(range(f, 0, |f|), 0, 4);

    Object x = new Object(1);
    Object y = new Object(4);
    Object z = new Object(6);
    seq<Object> h = seq<Object> {x, y, z};
    assert h[..1] == seq<Object> {x};
  }
}

class Object {
  int var;

  Object (int v) {
    var = v;
  }
}
```

Listing 7.14: Example of using the `range`, `take` and `drop` functions/operators

7.4 Removing Values from Sequences by Index

In this section, the `remove` operation is introduced. The implementation of this operation is similar to the `range` operation, therefore this section is less detailed by excluding the discussion on implementation choices.

Description and syntax

Remove

Syntax:
`remove(a, b)`

Arguments:
a - A sequence.
b - An integer index within the range of sequence `a`.

Description:
Construct a new sequence using the elements of sequence `a` excluding the element at index `c`.

Returns:
A new sequence with all values from sequence `a` except the element at index `b`

Implementation

This feature is implemented using function generation. Both variants of syntactic sugar are not an option since it is not simple to express in one (or a few) COL expression. Viper does not support this feature natively, so mapping to the back end is not an option. Defining a new domain for this functionality is more involved than using function generation.

For this operation, only a single syntax has been defined. An intuitive syntax using the minus operator was considered, however this was discarded to avoid confusion with similar `remove` operations in other languages where instead of providing an index to remove, an element is provided which is removed from the sequence. For example, for an integer sequence `a` with values `[1, 2, 3, 4]`, the syntax `a - 2` could both be interpreted as removing the second index or removing the value 2.

Since the syntax is a function syntax, no additions need to be made to the ANTLR grammar file. A new `StandardOperator` is defined named `Remove` (see Listing 7.15) and in `PVLtoCOL` the syntax is matched and transformed into a COL expression with operator `Remove` (see Listing 7.16).

```
public enum StandardOperator {
  ..., Remove(2), ...
}
```

Listing 7.15: The enum class `StandardOperator` with operators `Remove`

```
if (match(ctx,"remove",tuple)){
  return create.expression(StandardOperator.Remove,args);
}
```

Listing 7.16: Matching the syntax of the `remove`

No additional rewrites are needed on the `remove` operation, thus the next step is generating the function. This task is again performed by the rewrite pass `RewriteSequenceFunctions` which has also been used for the `range` operation in the previous section. Again, this pass has two purposes, firstly identify which functions need to be generated and secondly generate the identified functions. In the first step, the tree is visited and all COL expression with operator `Remove` are rewritten to a COL function invokation (see Listing 7.17).

To invoke a function, the name and arguments of the function are required. The name of the function to invoke consists of two parts, a predefined part and a part specific to the function. In the case of the operation `remove`, the predefined part is chosen to be *remove_ by_ index_* and the specific part is the type of the sequence. The type of the sequence and the name of the function to generate are again stored in a map to be used in the next step.

```
override def visit(operator: OperatorExpression): Unit = {
  operator.operator match {
    case StandardOperator.Remove =>
    val sequenceType = operator.arg(0).getType
    result = create.invokation(null, null, RewriteSequenceFunctions.getRemoveFunction(sequenceType),
        ↪ rewrite(operator.args.toArray):_*)
  }
}
```

Listing 7.17: Rewriting a COL expression with operator `Remove` to a COL function invokation

The next step is to generate the invoked functions. The map built in the previous step contains all information that is required to generate the `remove` function, that being the type of the sequence the `remove` function has to operate on and the name of the function. Before continuing with implementation in detail, let us first look at the function to generate.

Listing 7.18 shows the PVL equivalent of the generated `remove_by_index_` function. As can be seen, there is one precondition and three postconditions. The precondition states that the given index should be valid and the postconditions state that the result should be the elements from the original sequence excluding the removed element.

```
requires 0 <= i && i < |sequence|;
ensures |\result| == |sequence| - 1;
ensures (forall int j; 0 <= j && j < i; \result[j] == sequence[j]);
ensures (forall int j; i <= j && j < |\result|; \result[j] == sequence[j + 1]) ;
seq<T> remove_by_index_T(seq<T> sequence, int i);
```

Listing 7.18: PVL equivalent of the generated `remove_by_index_` function

Now that we know what needs to be defined, let us look at how it is defined. Generating the function is performed by the Scala method `removeFromSequenceByIndex` in the `RewriteSequenceFunctions` class. Similar to generating the function for the `range` operation, a `ContractBuilder` is defined by which a COL contract can be easily constructed. After the contract has been defined, a COL function declaration is created with the proper return type, contract, function name and arguments.

Simplified versions of the definition of one precondition and one postcondition can be seen in Listing 7.19 corresponding to the first and second conditions in Listing 7.18. The result of this Scala method is a bodyless COL function expressing the `remove` operation and this function is added to the COL AST as a static function.

```
def removeFromSequenceByIndex(sequenceType: Type, functionName: String): ASTNode = {
  val contract = new ContractBuilder
  val result = create.reserved_name(ASTReserved.Result, sequenceType)

  val sequence; // The original sequence
  val index; // The index to remove

  // Require the index to be valid in the given sequence
  contract.requires(
    and(
      lte(0, index),
      less(index, size(sequence))
    )
  )

  // Ensure |result| to be equal to |original sequence| - 1
  contract.ensures(
    eq(
      size(result),
      minus(
        size(name(sequence)),
        create.constant(1)
      )
    )
  )

  val declaration = create.function_decl(sequenceType, contract.getContract, functionName,
      ↪ functionArguments.toArray, null)
  declaration.setStatic(true)
  declaration
}
```

Listing 7.19: Simplified definition of a pre and postcondition of the `range` operation.

Small example

See Listing 7.20. A sequence of integers `a` is instantiated with some values. For each of the following sequences, a value at some index is removed and it is asserted the value has been removed until the sequence is empty. Afterwards, a sequence of Edge objects is instantiated and it is asserted that the remove function also works on objects.

```
class RemoveValues {

  void main() {
    seq<int> a = seq<int>{1, 2, 3, 4, 5};
    seq<int> b = remove(a, 2);
    assert b == seq<int> {1, 2, 4, 5};

    seq<int> c = remove(b, 0);
    assert c == seq<int> {2, 4, 5};

    seq<int> d = remove(c, 2);
    assert d == seq<int> {2, 4};

    seq<int> f = remove(d, 1);
    assert f == seq<int> {2};

    seq<int> g = remove(f, 0);
    assert empty(g);

    Edge e1 = new Edge(0, 1);
    Edge e2 = new Edge(1, 2);
    Edge e3 = new Edge(2, 0);
    seq<Edge> es = seq<Edge> {e1, e2, e3};
    assert remove(es, 1) == seq<Edge> {e1, e3};
  }
}

class Edge {
  int source;
  int target;

  Edge (int s, int t) {
    source = s;
    target = t;
  }
}
```

Listing 7.20: Example of using the `remove` function

7.5 Simple Collection Constructors

Description and syntax

The current syntax for declaring an empty sequence of type T is `seq<T> {}` and to initialize the sequence with some values the syntax `seq<T> {val1, val2, val3}` is used. The syntax for sets and bags are equivalent.

This syntax makes it difficult to write readable code when using it multiple times in a single function such as the two PVL functions in Listing 7.21. To solve this, a simpler syntax is needed for sequences, sets and bags.

```
pure static seq<int> update(seq<int> xs, int i, int v) =
  0 < i ? seq<int> { head(xs) } + update(tail(xs), i - 1, v) : seq<int> { v } + tail(xs);

pure static seq<seq<int>> Place(seq<seq<int>> stacks, int card) =
  empty(stacks) ?
    seq<seq<int>> { seq<int>{ card } } :
    (head(head(stacks)) >= card ?
      seq<seq<int>> { card :: head(stacks) } + tail(stacks):
      seq<seq<int>> { head(stacks) } + Place(tail(stacks), card)
    )
;
```

Listing 7.21: Examples of PVL methods using sequence constructors

We decided to implement square bracket notation for sequences based on the suggestion made in the VerCors Axiomatic Data Types documentation [28]. This notation/syntax is also used by languages such as Python and Haskell. For sets, curly brackets are suggested and for bags curly brackets preceded by a `b` are suggested.

Since the type of the constructed collection is not given, it has to be inferred. Type inference is deducing the type of an expression based on the context. In this case, the context is mainly the values provided to initialize the collection. The topic of type inference and its scope for this feature is discussed in the next section. Let us first introduce simple notation.

Simple Empty Collection Constructor

Syntax:

`<openingbracket> t:T <closingbracket>`

Arguments:

T - The type of the new collection.

Description:

Construct a empty collection with type `T`.

Returns:

A empty collection with type `T`.

Simple Initialized Collection Constructor

Syntax:
`<openingbracket>a, b, c<closingbracket>`

Arguments:
a - A value of the new collection.
b - A value of the new collection.
c - A value of the new collection.

Description:
Construct a new collection with the type of the given value(s). The values have to be of the same type. The syntax definition shows, as an example, 3 values. The minimum amount of values is 1 with theoretically no upper bound.

Returns:
A new collection with the type of the given value(s).

For example, the syntax for a sequence with values 1 to 3 is `[1,2,3`, for sets that would be `{1,2,3}` and for bags that would be `b{1,2,3}`. For an empty integer sequence is `[t:int]`, for an empty integer set it is `{t:int}` and for an empty integer bag it is `b{t:int}`.

Type inference

A variety of languages such as Java 7+, Kotlin, Python, Scala, Haskell and more have some level of type inference baked into the language. In general, we can say that every language infers the type of an expression using some form of a type system where for example a comparison with an equals operator == is deduced to be of type boolean and the addition of two integers is again an integer. There are more complex cases where the type of an expression is not obvious and this is where type inference in programming languages is implemented.

Let us look at how some languages use type inferences at different scopes, starting with Java. There are two places where Java uses type inference where the type of an expression is not obvious. The first place is with diamond operators in assignments (for Java 7+). Let us first look at the example in Listing 7.22. The first line instantiates a list of strings `someListOfStrings` to an empty list where the type of the value (i.e. the right side) and the target (i.e. the left side) can be immediately inferred to be lists of string. Semantically, the second line is equal to the first, however the type of the value is not explicitly stated. The user could either explicitly tell Java what the type is in the diamond operator as in the first line or the user can tell Java to infer the type of the value by leaving the diamond operator empty as in the second line. Java deduces the type of the value from the target which is a list of strings.

```
List<String> someListOfStrings = new ArrayList<String>();
List<String> anotherListOfStrings = new ArrayList<>();
```

Listing 7.22: Type inference of Java for the diamond operator

The second case where Java has type inference is Java 10's `var` keyword. The `var` keyword can be used to declare a local variable without explicitly stating the type. Let us look at the example in Listing 7.23. The first line is what we have seen above where the type can be immediately inferred to be a list of objects. Since the type of the value in the assignment can be inferred directly, the type of the target can be replaced by the keyword `var`. By using this keyword, Java knows to infer the type from the context in which it is used.

```java
List<Object> objects = new ArrayList<Object>();
var objects2 = new ArrayList<Object>();
```

Listing 7.23: Type inference of Java 10+ for the `var` keyword

The type system of languages such as Python and Haskell are entirely based on type inference. The type of a variable or function does not need to be explicitly written down anywhere. Haskell has a gradual type system, meaning that the type of a function can either be stated explicitly or implicitly, in the latter case the type is inferred at compile time or by the interpreter. Languages such as Python have type checking during runtime, meaning that if some types are incompatible the program fails at runtime.

Although it is not required to specify the type of a variable or function, Haskell does have a way to specify a type baked into the language (see Listing 7.24) and Python 3.5+ has a library for type hinting which can be used by third party type checkers and code linters (see Listing 7.25).

```haskell
increment :: Int -> Int
increment x = x + 1
```

Listing 7.24: Type declaration in Haskell

```python
# Without type hints
def scale1(x):
return x + 1

# With type hints
def scale2(x: Int) -> Int:
return x + 1
```

Listing 7.25: Type hints in Python 3.5+

Implementation

To implement type inference for collection constructors in VerCors, the scope of type inference needs to be defined. The scope of type inference is set to the scope of the constructor, meaning that only the values supplied in the constructor are used. Information about the context such as if it is used in an assignment or a method call is not available. Having type inference at a larger scope, for example an assignment such as with Java 10+'s `var` keyword, requires more involved changes to the architecture of VerCors. For programming languages, it makes sense to implement type inference at a larger scope, however for a verification language without a runtime the benefit is not as apparent.

Recall that there are two simple constructors; a constructor for an empty collection where the type is supplied and a constructor for an initialized collection where the values are supplied. The reason for this distinction is directly related to the scope of type inference. In the case where values are supplied the type can be inferred from those values. In the case where there are no values to infer the type from, the type has to be supplied.

For the rest of the section, the focus of the discussion is put on sequences to keep the listings readable. The steps for sets and bags are equivalent.

Although this feature might be dissimilar to other operations discussed above, the implementation generally follows the same steps as transformed syntactic sugar: match the syntax to transform it into a COL expression and define a pass to perform some action or transformation. Let us start with the syntax defined in the ANTLR file. Before the implementation of this feature, the ADT constructors for sequences, sets and bags were a part of the `atomExpression` rule which is the building block of PVL expressions. Instead of adding new cases to the rule `atomExpression`, a new rule was defined named `collectionConstructors` which contains all different constructors for ADTs (see Listing 7.26). In the future, if a new constructor for any ADT needs to be added, it can simply be added to the `collectionConstructors` rule.

```
atomExpression
  : collectionConstructors
;

collectionConstructors
  : CONTAINER '<' type '>' values
  | '[' arguments ']'
  | '[t:' type ']'
;

CONTAINER : 'seq' | 'set' | 'bag' ;
```

Listing 7.26: The ANTLR grammar rules `atomExpression` and `collectionConstructors`

Since a new rule is defined in the syntax, the parser will have a new method called `visitCollectionConstructors` in which the syntax of the constructors is matched. Using the `match` method, the new sequence constructor syntax is matched (see Listing 7.27). If a type is provided, an empty sequence is defined with the given type. If values are provided, then a sequence is initialized with the given values and variable type `INFER_ADT_TYPE`.

```
if (match(ctx, "[t:", type_expr, "]")) {
  Type t=checkType(convert(ctx,1));
  return create.struct_value(create.primitive_type(PrimitiveSort.Sequence, t), null);
}
if (match(ctx, "[", null, "]")) {
  ASTNode[] args = getValues((ParserRuleContext)ctx.children.get(1));
  return create.struct_value(create.primitive_type(PrimitiveSort.Sequence, create.type_variable(
      ↪ InferADTTypes.typeVariableName())),null,args);
}
```

Listing 7.27: Matching the new constructors in `PVLtoCOL`

Type checking

During type checking, the variable type named `INFER_ADT_TYPE` is matched. The values are type checked and all non-null types (of the values) are collected into a set. Now there are three possibilities for the length of that set:

- **The length is 0.** In this case, all values have their type set to `null`. This is only the case when the type of none of the elements are set which could be an implementation error in the type checker. This error should, in theory, never be reached in a release. If it does happen, a descriptive error message is shown.
- **The length is 1.** In this case, all values have the same type and thus the type of the sequence can be inferred to be of the same type.

- **The length is more than 1.** In this case, the sequence cannot be constructed resulting in a fail. This case also does not work with the old constructor syntax simply because a sequence cannot contain elements of multiple types.

Types in COL

The variable type named `INFER_ADT_TYPE` is modeled by `TypeVariable`. `TypeVariable` is not the only type in VerCors. `TypeVariable` is one of the implementations of the abstract class `Type` and every `ASTNode` in a COL AST has a `Type`.

When choosing the correct `Type` to be used for type inference, different types were considered. In the current version of VerCors, there are seven implementations of `Type` which are `PrimitiveType`, `ClassType`, `FunctionType`, `RecordType`, `TupleType`, `TypeExpression` and `TypeVariable`. Let us go over these different types, discussing their function and how they are used within VerCors.

- `PrimitiveType`: Models supported types. Used when any supported type (such as integers or sequences) are created.
- `ClassType`: Models the type with a class.
- `FunctionType`: Models the type of a function as a list of types for the arguments and a result type. Used for functions and methods.
- `RecordType`: Models a mapping from variable names to their types. Used in tests.
- `TupleType`: Models a tuple of type (e.g (integer, boolean)). Used by some operators to keep track of the types they operate on.
- `TypeExpression`: Models an expression of types, similar to what we have seen with `OperatorExpression`. `TypeExpression` takes an operators of type `TypeOperator` and an array of `Types`. Used to express that some type `T` extends some type `G`.
- `TypeVariable`: Models a Type with a variable name.

The three viable options are `PrimitiveType`, `ClassType` and `TypeVariable` with `TypeVariable` being the best option. A `PrimitiveType` is not used since the inferred type `INFER_ADT_TYPE` is not a supported type, it is a temporary type. A `ClassType` is not used since using a `ClassType` would imply that a class with that specific name would be created. This creates unnecessary overhead.

Type of `StructValue`

A `StructValue` is an AST node that models a structured value (such as a sequence or bag). `StructValue` has two types: the type given when creating it and the type inherited by `ASTNode`. The difference between the two is that the type inherited from `ASTNode` is omitted after a rewrite pass and the type given at creation is not. This means that the variable type `INFER_ADT_TYPE` is still the type of the collection.

A rewrite pass `InferADTTypes` is defined to properly set the both types of the collection (see Listing 7.28). Four conditions have to be met to infer the type:

1. First, the `StrucValue` has to be a sequence, set or bag.
2. Second, the collection should not be empty.
3. Third, the type should be a `TypeVariable`.
4. Lastly, the name of the `TypeVariable` should match `INFER_ADT_TYPE`.

When all of these conditions are met, a new `StructValue` is created with the values of the *old* collection and the inferred type. All types should be known after this pass or in other words there should be no type `INFER_ADT_TYPE` in the COL AST.

```
class InferADTTypes(source: ProgramUnit) extends AbstractRewriter(source, true) {
  override def visit(v: StructValue): Unit = {
    if((v.`type`.isPrimitive(PrimitiveSort.Sequence) || v.`type`.isPrimitive(PrimitiveSort.Set) || v.`
        ↪ type`.isPrimitive(PrimitiveSort.Bag)) &&
      v.`type`.args.nonEmpty &&
      v.`type`.firstarg.isInstanceOf[TypeVariable] &&
      v.`type`.firstarg.asInstanceOf[TypeVariable].name == InferADTTypes.typeVariableName
      ) {
        // If the inference succeeded in the type checker, then the type should be v.getType
        result = create.struct_value(create.primitive_type(v.`type`.asInstanceOf[PrimitiveType].sort, v.
            ↪ getType.firstarg), null, v.values: _*)
    } else {
      super.visit(v)
    }
  }
}
```

Listing 7.28: The rewrite pass `InferADTTypes`

Limitation

The feature in its current form is relatively simple. For future reference, one of the possible difficulties is discussed.

If inheritance is going to be supported in PVL, both type checking and type inference will have to be reimplemented. The current type check and type inference are simple since the type system is not complex. When inheritance is supported, the type checker will have to find the closest common superclass of all values. For example, if we have a class `A` extending `C` and a class `B` extending `C`, then the closest common superclass is class `C` and thus the sequence will be of type `C`. When inferring the type, a similar approach must be taken.

Small example

See Listing 7.29. In this example, two sequences `a` and `b` are instantiated using the old syntax and two equivalent sequences `c` and `d` are instantiated using the new syntax and it is asserted that the new syntax behaves the same as the old syntax. An empty sequence of `Edge` objects in instantiated and it is asserted that an initialized sequence with some values is equivalent to an empty sequence with those values prepended to it.

```
class SimpleCollectionConstructors {
  void main() {
    seq<int> a = seq<int> {};
    seq<int> b = seq<int> {1, 5, 7, 9, 2};

    seq<int> c = [t:int];
    seq<int> d = [1, 5, 7, 9, 2];

    assert a == c && b == d;

    Edge e1 = new Edge(0, 1);
    Edge e2 = new Edge(1, 2);
    seq<Edge> es = [t:Edge];

    assert [e1, e2] == e1::e2::[t:Edge];

    set<int> f = set<int> {};
    set<int> g = set<int> {1, 5, 7, 9, 2};

    set<int> h = {t:int};
    set<int> i = {1, 5, 7, 9, 2};

    assert f == h && g == i;

    bag<int> j = bag<int> {};
    bag<int> k = bag<int> {1, 1, 5, 7, 9, 2};

    bag<int> l = {t:int};
    bag<int> m = b{1, 1, 5, 7, 9, 2};

    assert j == l && k == m;
  }
}

class Edge {
  int source;
  int target;

  Edge(int s, int t) {
    source = s;
    target = t;
  }
}
```

Listing 7.29: Example of using the simple sequence constructor

7.6 Subset Notation

Description and syntax

The `subset` operator is defined as follows:

SubSet

Syntax:
`a < b`

Arguments:
a - A set or bag.
b - A set or bag.

Description:
Check if the argument `a` is a proper subset of argument `b`. The types of `a` and `b` have to match.

Returns:
`true` if `a` is a proper subset of `b`.

SubSetEq

Syntax:
`a <= b`

Arguments:
a - A set or bag.
b - A set or bag.

Description:
Check if the argument `a` is a subset of argument `b`. The types of `a` and `b` have to match.

Returns:
`true` if `a` is a subset of `b`.

Implementation

Silver has a `subset` operator for sets and bags, so the subset operator is directly mapped to the back end. The syntax already exists to compare integers, so nothing is added to the ANTLR grammar and `PVLtoCOL` rewrites the $<=$ and $<$ operators to the `StandardOperators` `LTE` and `LT` respectively.

The `LTE` and `LT` operators are rewritten in a new rewrite pass `ADTOperatorRewriter` (see Listing 7.30). This separates the behavior of the operators for numeric types and for sets/bags. The operators are rewritten to expressions with new `StandardOperators` `SubSetEq` and `SubSet`.

A mapping of the `StandardOperators` to Silver expressions is defined in the `SilverExpressionMap` (see Listing 7.31). The `SubSetEq` operator is mapped directly to the Silver subset operator. The `SubSet` operator is mapped to the expression `argument1 subset argument2` $\wedge$ `|argument1| < |argument2|`[4]. The Silver expression (as part of the Silver project) for the subset operator is named `AnySetSubset` and using the `SilverExpressionFactory` this expression is retrieved (see Listing 7.32).

[4]An Isabelle proof for this rewrite rule is given in Appendix B.

```
public class ADTOperatorRewriter extends AbstractRewriter {
  @Override
  public void visit(OperatorExpression e) {
    switch (e.operator()) {
      case LTE: case LT:
      if (e.arg(0).getType().isPrimitive(PrimitiveSort.Set) ||
        e.arg(0).getType().isPrimitive(PrimitiveSort.Bag)) {
          StandardOperator op = (e.operator().equals(StandardOperator.LT)) ? StandardOperator.SubSet :
               ↪ StandardOperator.SubSetEq;
          ASTNode e1 = e.arg(0).apply(this);
          ASTNode e2 = e.arg(1).apply(this);
          result = create.expression(op, e1, e2);
        } else {
          super.visit(e);
        }
    }
  }
}
```

Listing 7.30: Rewriting the `LTE` and `LT` operators for sets and bags

```
public class SilverExpressionMap implements ASTMapping {
  public Exp map(OperatorExpression e) {
    switch(e.operator()){
      ...
      case SubSet: create.and(o, create.any_set_subset(o, e1, e2), create.lt(o, create.size(o, e1),
           ↪ create.size(o, e2)));
      case SubSetEq: return create.any_set_subset(o, e1, e2);
      ...
    }
  }
}
```

Listing 7.31: The mapping of `SubSet` and `SubSetEq` in `SilverExpressionMap`

```
class SilverExpressionFactory {
  override def any_set_subset(o:O,e1:Exp,e2:Exp):Exp = add(AnySetSubset(e1,e2)_,o)
}
```

Listing 7.32: The Silver subset expression in `SilverExpressionFactory`

Small example

See Listing 7.33. In this example, two sets `a` and `b` and two bags `c` and `d` are initialized with some values. It is asserted that a set is a subset of itself and that a smaller set/bag is a proper subset of a larger set/bag.

```
class SubSet {
  void main() {
    set<int> a = set<int> {1, 5, 7, 8, 6, 1, 4, 8, 6, 3};
    set<int> b = set<int> {1, 5, 7};
    bag<int> c = bag<int> {1,5,7,4,9,6,3,2,4,5};
    bag<int> d = bag<int> {4,6,2,4,5};

    assert b <= b && b < a && d < c;
  }
}
```

Listing 7.33: Example of using the subset operator

Chapter 8

Set Comprehension

This chapter answers the questions for RQ4 on set comprehension. Section 8.1 answers RQ4.1, showing the definition of set comprehension with its syntax. Section 8.2 answers RQ4.2 by discussing the encoding of set comprehension in Viper. Next, the implementation is discussed followed by a discussion on limitations and design choices based on the limitations.

The implementation of set comprehension (as described in this chapter) can be found in commit `ce03982` which can be browsed at https://github.com/OmerSakar/vercors/tree/ce03982b25b13d693a4b6fd2dda0fee62de96ba3.

8.1 Description and Syntax

Set Comprehension

Syntax:

```
set<T> { main | (U var (<- collection)?)+; selector}
```

Arguments:

T - The type of the resulting set.
main - The expression to be part of the set (of type `T`).
selector - The condition to include an element.
U var - A variable `var` of type `U`
collection - A sequence, set or bag.

Description:

Set comprehension is building a set based on generators. All variables `var` are quantified over and have a possible domain `collection` (i.e. the value of `var` is from the collection `collection`). The expression `main` is the element to include in the set only if the expression `selector` holds.

Returns:

A set with expressions of the form `main` based on the variable(s) `var` on the condition that `selector` holds.

Since the syntax is complex, consider the example in Listing 8.1. We have two integers `x` and `y` from sequences with values 1 to 5. We add the expression `x+y` to the resulting set iff `x` equals `y`.

```
set<int> {x+y | int x <- {1, 2, 3, 4, 5}, int y <- {1, 2, 3, 4, 5}; x == y}
```

Listing 8.1: Example of set comprehension

8.2 Design

The example in Listing 8.1 can be read as follows: for all integer x in the set $\{1, 2, 3, 4, 5\}$ and integers y in the set $\{1, 2, 3, 4, 5\}$, $x == y$ iff $x + y$ is in the resulting set. From this alternative interpretation, the set comprehension could be seen as a set with a universal quantifier describing its elements.

To translate this approach to Viper, there must be a construct that Viper supports to get such a set. There is no special syntax to creating a set out of nothing, however there are bodyless functions. A function can return a set with the postcondition of the function describing the elements from that set.

In the design we differentiate between three cases:

- Quantifying over integers with a finite domain
- Quantifying over objects with a finite domain
- Quantifying over integers with an infinite domain

Quantifying over integers with a finite domain

Listing 8.1 shows an example of quantifying over integers with a finite domain. The Viper encoding of this Listing is shown in Listing 8.2. The domain of the integers `x` and `y` (i.e. the range of values `x` and `y` can have) are supplied through the two arguments of the function `domainOfX` and `domainOfY`. The post condition describes the elements of the resulting set in almost the same way as the alternative interpretation above: For all integers `x` and `y`, if they are in their respective domain, then $x == y$ iff $x + y$ is in the resulting set (where the special keyword `result` refers to the result of the function).

```
function setComprehension(domainOfX: Set[Int], domainOfY: Set[Int]): Set[Int]
   ensures forall x: Int, y: Int :: (x in domainOfX && y in domainOfY) ==> (x==y) <==> (x+y in result)
```

Listing 8.2: The Viper encoding of Listing 8.1

Quantifying over objects with a finite domain

Set comprehension on objects is similar to the previous case with one additional step. As mentioned in the background, classes in PVL are encoded as `Refs` in Viper. These references have permission on fields that are declared at the top level. When using a collection of objects in set comprehension, read permissions on the fields of that class are needed as a minimum.

Listing 8.3 shows an example of set comprehension over objects. We have a class `Edge` with two integer fields `a` and `b` and a constructor initializing the fields returning read permission on both fields. In the method `main` of the `SetComp` class, two `Edge` objects are instantiated after which a set of `Edges` `a` is initialized using set comprehension. The two initialized `Edge` objects are quantified over and added to the set iff the field `a` is larger than 2.

The Viper encoding of Listing 8.3 is shown in Listing 8.4. The two fields `a` and `b` (prepended with `Edge_`) are declared at top-level. The precondition states that all `Refs` (i.e. `Edge` objects) in the argument of the function must have wildcard permissions or in other words read permissions at a minimum.

```
class SetComp {
  void main() {
    Edge e1 = new Edge(3, 2);
    Edge e2 = new Edge(1, 2);
    set<Edge> es = set<Edge> { e | Edge e <- set<Edge> {e1, e2}; e.a > 2 };
  }
}

class Edge {
  int a;
  int b;

  ensures Perm(a, read) ** Perm(b, read);
  Edge(int c, int d) {
    a = c;
    b = d;
  }
}
```

Listing 8.3: Example of set comprehension over objects

```
field Edge_a: Int
field Edge_b: Int

function vct_set_comprehension_Set_Edge_(domainOfE: Set[Ref]): Set[Ref]
    requires (forall e: Ref :: (e in domainOfE) ==> acc(e.Edge_a, wildcard) && acc(e.Edge_b, wildcard))
    ensures (forall e: Ref :: (e in domainOfE) ==> 2 < e.Edge_a == (e in result))
```

Listing 8.4: The Viper encoding of Listing 8.3

Quantifying over integers with an infinite domain

For integers, it is also possible to quantify over them with a (theoretical) infinite domain. Suppose that we want all positive integers divisible by 2 (see Listing 8.5).

```
  set<int> {x | int x; x > 0 && x % 2 == 0}
```

Listing 8.5: Example of set comprehension over integers with an infinite domain

The Viper encoding of Listing 8.5 is shown in Listing 8.6. The function returns a set of integers containing all integers (from the infinite domain of all possible integers) where the condition holds.

Since the domain of integers is infinite, the resulting set can also be (in theory) infinite. In Chapter 2, it was stated that sets in Viper are finite. These two seemingly conflicting observations can be combined with the following description of the resulting set; The resulting set is not infinite, however it models an infinite set.

```
function setComprehension(): Set[Int]
    ensures (forall x: Int :: (0 < x && x % 2 == 0) == (x in result))
```

Listing 8.6: The Viper encoding of Listing 8.5

8.3 Implementation

As stated above, a bodyless function can be used to get a set with the desired properties. Given that set comprehension is not part of Viper and that it is a new feature to VerCors, function generation seems like the only viable option.

The approach to implement set comprehension using function generation is slightly different from the general approach explained in Section 6.3. The steps for the general approach are as follows:

1. The syntax related to the functionality needs to be defined.
2. A new operator is defined in enum class `StandardOperator`.
3. The syntax is matched in `PVLtoCOL` and transformed into a COL expression with the newly defined operator and its arguments.
4. A rewrite pass is defined (or an existing one is used) that generates the related function.

The steps that are taken for this specific case (which differ in steps 2 and 3) are:

1. The syntax related to the functionality needs to be defined.
2. A new binder is defined in enum class `Binder`.
3. The syntax is matched in `PVLtoCOL` and transformed into a COL `BindingExpression` AST node with the newly defined binder and its arguments.
4. A rewrite pass is defined (or an existing one is used) that generates the related function.

There are two reasons to choose a `BindingExpression` over an `OperatorExpression` (in the general approach). First, a `StandardOperator` has a constant number of arguments. Set comprehension needs to keep track of the variables, selector expression and main expression and the relation between them. A constant number of arguments is not suited to model set comprehension. Second, set comprehension is very similar to a universal quantifier which is modeled by a `BindingExpression`. This has the added benefit that the logic for `BindingExpressions` is already present in VerCors and is suitable for this use case as well, for example the scoping of the bounded variables for the selector and main expressions.

The `BindingExpression` AST node models an expression that binds fresh variables to an expression. In the case of the universal/existential quantifier, the selector (or guard) and the main expression are bound by the variables which are quantified over. Listing 8.7 shows a universal quantifier with the different parts underlined.

```
\forall(int x, int y; x >= 0 && x < 5 && y >= 0 && y < 5; x+y >= 0 && x+y <= 8);


          Variables                   Selector                   Main
```

Listing 8.7: An example of a binding expression with the universal quantifier as its binder

The AST node `SetComprehension`

Besides the variables, selector and main expression modeled in `BindingExpression`, set comprehension needs to keep track of the domain of the quantified variables. This is modeled by a new Java class `SetComprehension` which extends `BindingExpression`.

Listing 8.8 shows the new Java class `SetComprehension`. `SetComprehension` has a Java map of the quantified variable names (in the form of `NameExpressions`) to their domain (in the form of `ASTNodes`). The constructor has the following arguments:

1. `result_type`: The result type of the expression itself. The expression in this case is set comprehension and the result type is a set with elements of some type.
2. `decls`: The quantified variables. The declaration of these new variables are modeled as `DeclarationStatement`.
3. `selector`: The selector expression.
4. `main`: The main expression. The type of this expression matches the type of the elements of the resulting set.
5. `variables`: A map of variable names to their domain.

The first four arguments are passed to the constructor of `BindingExpression` along with the enum value `Binder.SetComp`. `Binder` is an enum class used by `BindingExpression` to signify what kind of binding expression it is (e.g. forall and exists). For set comprehension, the value `SetComp` is added to `Binder` (see Listing 8.9) and passed to the constructor of `BindingExpression`.

```java
public class SetComprehension extends BindingExpression {

  public Map<NameExpression, ASTNode> variables;

  public SetComprehension(Type result_type, DeclarationStatement[] decls, ASTNode selector, ASTNode main
      ↪ , Map<NameExpression, ASTNode> variables) {
    super(Binder.SetComp, result_type, decls, new ASTNode[0][], selector, main);
    this.variables = variables;
  }
}
```

Listing 8.8: The Java class `SetComprehension`

```java
  public enum Binder {
    ...,
    SetComp
  }
```

Listing 8.9: The Java enum `Binder`

Syntax definition

The syntax for set comprehension is added in the form of three ANTLR grammar rules. Listing 8.10 shows these three rules. The rule `collectionConstructor` is the main rule and it is used to define constructors for collections which are natively supported (i.e. sets, bags and collection). The lexer rule `CONTAINER` contains the token *set*. The grammar rule `setCompSelectors` and `setCompSelector` define the syntax to quantify over variables (e.g. `int x <- [0, 1, 2]`). The reason for separating these rules from `collectionConstructors` is for easier parsing since some quantified variables (of type integer) do not have a domain.

Listing 8.11 shows how the syntax is matched in `PVLtoCOL`. The `match` method is used to match the syntax and the AST node factory named `create` is used to create a new instance of `SetComprehension` as described above. The two methods `getVariableDecls` and `getVarBounds` are used to go over the quantified variables and return the declared variables and a map of variables to their domain respectively.

```
setCompSelector :
    type identifier ('<-' (identifier | collectionConstructor))?
;
setCompSelectors :
    setCompSelector (',' setCompSelector)*
;

collectionConstructor :
    CONTAINER '<' type '>' '{' expr '|' setCompSelectors ';' expr '}'
;
```

Listing 8.10: The ANTLR grammar rules of the set comprehension

```
if (match(ctx,"set","<",null,">","{",null,"|",null,";",null,"}")) {
  // The type of the resulting set elements
    Type t=checkType(convert(ctx,2));

    // Get all variables which are quantified over
    DeclarationStatement[] variables = getVariableDecls((ParserRuleContext)ctx.getChild(7));
    // Get all variables which are quantified over including their bounds
    Map<NameExpression, ASTNode> varBounds = getVarBounds((ParserRuleContext)ctx.getChild(7));

  return create.setComp(
      create.primitive_type(PrimitiveSort.Set, t), // The result type is a set with elements of type t
      convert(ctx,9), // The selector expression
      convert(ctx,5), // The main expression
      varBounds,
      variables
    );
}
```

Listing 8.11: Matching the set comprehension syntax in PVLtoCOL

Generating the function

The function is generated by the method generateSetComprehensionFunction in a new rewrite pass GenerateADTFunctions. This method gets the SetComprehension object and the name of the function to generate as its arguments. The method generates the function in six steps:

1. Create local variables to keep track fo the result type, ContractBuilder and the result keyword.
2. Collect all field accesses from the main and selector expressions.
3. For each field access, acquire wildcard permission (i.e. at least read permission) for those fields from the quantified variables in a precondition.
4. Create an argument for each quantified variable with a domain.
5. Generate the postcondition according to the following pseudo-code template: For all quantified variables with a domain, if those variables are within their domain it implies that the quantified variable is in the result set iff the selector holds.
6. Create a function declaration with the result type, contract, name and arguments.

The second step is performed by a RecursiveVisitor. Previously, implementations of the abstract class RecursiveVisitor were used as visitor passes. For this use case, a tree visitor FieldAccessCollector is created to walk the main and selector expressions (see Listing 8.12).

A list is made to store `Dereference` which is the ASTNode representing a field access. The corresponding `visit` method is overwritten and the ASTNode `e` (corresponding to the field access) is stored in the list. This list is then used in step 3.

```
public class FieldAccessCollector extends RecursiveVisitor<Object> {
  private List<Dereference> fieldAccesses = new ArrayList<>();

  @Override
  public void visit(Dereference e) {
    fieldAccesses.add(e);
  }
}
```

Listing 8.12: The `FieldAccessCollector` class

The steps 4 to 6 are equivalent to the steps taken in Section 7.3.

8.4 Limitations and Design Choices

8.4.1 Difficulties with proving simple properties

Consider the three cases in Listing 8.13. In all three cases a set `a` is constructed using set comprehension and it is asserted that $\ldots + \ldots$ is in the set:

- **Case 1:** Set `a` consists of all integers between 0 and 5 (inclusive). The assertion checks if $1 + 1$ is within the set. This assertion succeeds.
- **Case 2:** Set `a` consists of all $x + x$ where the integer `x` is between 0 and 5 (inclusive). The assertion checks if $1 + 1$ is within the set. This assertion fails.
- **Case 3:** Set `a` consists of all $x + x$ where the integer `x` is between 0 and 5 (inclusive). The assertion checks if $j + j$ is within the set where integer `j` is an argument of the method and is required to be within the bounds. This assertion succeeds.

```
class SetComp {

  void case1() {
    set<int> a = set<int> { x | int x; x >= 0 && x <= 5 };
    assert 1+1 in a; // Verifies
  }

  void case2() {
    set<int> a = set<int> { x+x | int x; x >= 0 && x <= 5 };
    assert 1+1 in a; // Fails
  }

  requires 0 <= j && j < 5;
  void case3(int j) {
    set<int> a = set<int> { x+x | int x; x >= 0 && x <= 5 };
    assert j+j in a; // Verifies
  }
}
```

Listing 8.13: Three different usecases of set comprehension

When the expression returned by set comprehension is the identity function as with case 1, it can be asserted that a certain value is in the set or is not in the set. When a non-identity function is used as with cases 2 and 3, it becomes more difficult for the SMT solver to solve the problem.

The difficulty is caused by the simplification of arithmetic expressions, in case 2 that is `1+1`. The assertion in case 2 gets simplified by either Viper or Z3 to `2 in a`. Z3 cannot match the simplified expression `2` against `x+x` and fails in verifying the problem. Case 3 is a similar case with an integer argument `j` and the reason that the assertion in case 3 succeeds is that j does not have a concrete value and thus cannot be simplified further.

The limitation in case 2 can be overcome by using a function to wrap around the returned expression. Listing 8.14 shows case 2 rewritten to use a function call. A new function `plus` is defined taking two integer arguments and returning the sum. The expression `x+x` is replaced with a function call. Instead of asserting `1+1 in a` it now has to be asserted that `plus(1,1) in a` and the same holds for `j+j`.

```
class SetComp {
  requires 0 <= j && j < 5;
  void case4(int j) {
    set<int> a = set<int> {SetComp.plus(x, x) | int x; x >= 0 && x <= 5 };
    assert plus(1, 1) in a; // Verifies
    assert plus(j, j) in a; // Verifies
  }

  pure static int plus(int a, int b) = a+b;
}
```

Listing 8.14: An equivalent method `case4` to the method `case2`

For any non-identity expression as its main expression, it is recommended to wrap the expression in a function and have a call to the function as the return expression. In the example above, the non-identity expression `x+x` is wrapped in the `plus` function. With a single function call, Viper can generate the correct triggers.

8.4.2 Scoping issues

Using a field of the current class or a local variable is not possible in the main and selector expressions. This limitation is put by the fact that the scope of the generated function is entirely new.

This limitation can be worked around relatively easily by defining another quantified variable with as its domain a sequence with the local variable/class field. Listing 8.15 shows two methods showing the limitation and the workaround. The method `wrong` fails due to the variable `b` being used in the main expression of the set comprehension. The method `right` works around this problem by defining a new quantified integer `c` with its domain a singleton sequence with `b`.

```
void wrong() {
  int b = 0;
  set<int> a = set<int> {x+b | int x; x >= 0};
}

void right() {
  int b = 0;
  set<int> a = set<int> {x+c | int x, int c <- [b]; x >= 0};
}
```

Listing 8.15: Scoping of Set Comprehension

The alternative solution is to generate arguments for each variable that is out of scope. This solution has a major disadvantage compared to the workaround above. Finding all variables used in the main/selector expression that is also out of scope is difficult. This would require keeping track of which variables are in scope and out of scope for this specific function which is a non-trivial problem.

8.4.3 Classes must have a domain

Quantified variables of the class type must have a domain (as mentioned in the design). This limitation comes from the fact that PVL classes are encoded as Silver `Refs` with the write permissions to the class fields.

As a reminder, `Refs` have all the declared fields, however they can only access those for which permission is acquired. This implies that when we reason about all `Refs` that have a certain set of permissions, we are reasoning about a larger set of `Refs` than the set of all instances of our PVL class. It is due to this reason that instances of classes have to be bound by a domain.

8.5 Examples using Set Comprehension

Listing 8.16 shows examples from the previous section which create three different sets using set comprehension.

```
class SetComp {
  void main() {
    set<int> c = set<int> {x+y | int x <- {1, 2, 3, 4, 5}, int y <- {1, 2, 3, 4, 5}; x == y};
    set<int> d = set<int> {x | int x; x > 0 && x % 2 == 0};

    Edge e1 = new Edge(3, 2);
    Edge e2 = new Edge(1, 2);
    set<Edge> a = set<Edge> { e | Edge e <- set<Edge> {e1, e2}; e.a > 2 };
  }
}

class Edge {
  int a;
  int b;

  ensures Perm(a, read) ** Perm(b, read);
  Edge(int c, int d) {
    a = c;
    b = d;
  }
}
```

Listing 8.16: An example using set comprehension in PVL

Chapter 9

Maps

This chapter answers the questions for RQ5 on maps. Section 9.1 answers RQ5.1, showing the definition of a map with its syntax. Section 9.2 continues by answering both RQ5.2 and RQ5.3, introducing the Dafny map axiomatization with its functions. Next, the implementation is discussed followed by a discussion on triggers chosen for the axioms in the `Map` domain.

The implementation of maps (as described in this chapter) can be found in commit `0899e5d` which can be browsed at https://github.com/OmerSakar/vercors/tree/0899e5da5498ca1ee3b49b9951c284e88e943944.

9.1 Description and Syntax

Similar to the discussion in Section 2.2, there is no a single definition for a map. The definition of a map, implemented as part of this thesis, follows the same line as the sequences, sets and bags in VerCors[1]. We define maps in VerCors as *an unordered, finite, immutable collection of key/value pairs with unique keys.*

Function/operators on maps are introduced below in the form of tables. Some functions have been grouped since their descriptions are very similar.

Map Constructor

Syntax:

```
map <K,V> {a -> b, c -> d}
```

Arguments:

K - The type of the keys.
V - The type of the values.
a, c - Keys of type `K`.
b, d - Values of type `V`.

Description:

Constructs a new map that maps elements of type `K` to elements of type `V`. The map is initialized with key/value pairs using the syntax `a -> b` to variable `a` of type `K` to variable `b` to type `V`. If no mappings are defined, an empty map is returned.

Returns:

A new map initialized with the given value(s).

[1] As a reminder, the ADTs in VerCors were finite and immutable collections.

Keys, values and items

Syntax:
`keys(m), values(m), items(m)`

Arguments:
m - A map with keys of type `K` and values of type `V`.

Description:
Gets the key set, value set or item set respectively of map `m`. The key set has elements of type `K`, the value set has elements of type `V` and the item set has elements of type `Tuple[K,V]` (introduced in the next Section).

Returns:
Return the key set, value set or item set respectively of map `m`.

Add a key/value pair

Syntax:
`buildMap(m, k, v)` or `m ++ (k,v)`

Arguments:
m - A map with keys of type `K` and values of type `V`.
k - A key of type `K`.
v - A value of type `V`.

Description:
Adds the key/value pair `(k,v)` to the map `m`. If the key already exists, update the value of the key.

Returns:
Returns a map with the new key/value pair.

Remove a key/value pair

Syntax:
`removeFromMap(m, k)`

Arguments:
m - A map with keys of type `K` and values of type `V`.
k - A key of type `K`.

Description:
Removes the key `k` and its associated value from the map `m`.

Returns:
Returns a map without the key `k`.

Get a value from a key

Syntax:
`getFromMap(m, k)` or `m[k]`

Arguments:
m - A map with keys of type `K` and values of type `V`.
k - A key of type `K`.

Description:
Gets the value mapped by the key `k` from the map `m`.

Returns:
Returns the value corresponding to the key `k` from the map `m`.

Cardinality

Syntax:
`cardMap(m)` or `|m|`

Arguments:
m - A map.

Description:
Gets the cardinality/size of map `m` which corresponds to the number of keys in the key set.

Returns:
Returns the cardinality of the map `m`.

Equals

Syntax:
`equalsMap(m1, m2)` or `m1 == m2`

Arguments:
m1 - A map with keys of type `K` and values of type `V`.
m2 - A map with keys of type `K` and values of type `V`.

Description:
Checks if two maps are equal. Two maps are equal iff the key sets are equivalent and the keys map to the same values.

Returns:
Returns `true` if the two maps are equivalent, else `false`.

Disjoint

Syntax:
`disjointMap(m1, m2)`

Arguments:
m1 - A map with keys of type `K` and values of type `V`.
m2 - A map with keys of type `K` and values of type `V`.

Description:
Check if two maps are disjointed. Two maps are disjoint iff no key is in both the key set of `m1` and the key set of `m2`.

Returns:
Returns `true` if the two maps are disjoint, else `false`.

9.2 The Silver Equivalent of Dafny's `Map` Axiomatization

Instead of reinventing the wheel, we base our map axiomatization on that of the Dafny tool. The axiomatization comes from the file `DafnyPrelude.bpl` [31] (from now on referred to as `DafnyPrelude`). Among axiomatizations for sequences, sets and multisets, two axiomatizations can be found for a map: `Map` and `IMap`.

`Map` models a finite map and `IMap` models a map with (possibly) infinitely many key/value pairs. There is one key difference between `Map` and `IMap` that is the cardinality of a map is undefined for `IMap`. Both the function for cardinality and axioms to express its behavior are undefined in `IMap`. The axiomatizations for `Map` are used since we defined a map to be finite.

Instead of explaining every axiom, we discuss the different functions and show some examples of axioms. The behavior of these functions has been described in Section 9.1. The full axiomatization in Viper/Silver can be found in Appendix C. It can be assumed that the Viper axioms in the following sections are part of the domain `VCTMap` with type parameters `[K,V]` and functions are prefixed with `vctmap_` unless specified otherwise.

Tuples

The axiomatization of `Map` uses tuples to keep track of the key/value pairs in the map. Tuples are defined as a new type using a domain (see Listing 9.1). The domain consists of a constructor `vcttuple_tuple` and two destructors `vcttuple_fst` and `vcttuple_snd` that return the first and second elements in the tuple respectively.

```
domain VCTTuple[F,S] {
    function vcttuple_tuple(f:F, s:S): VCTTuple[F,S]
    function vcttuple_fst(t:VCTTuple[F,S]): F
    function vcttuple_snd(t:VCTTuple[F,S]): S

    axiom vctTupleFstAx {
        forall f1:F, s1:S :: vcttuple_fst(vcttuple_tuple(f1,s1)) == f1
    }

    axiom vctTupleSndAx {
        forall f1:F, s1:S :: vcttuple_snd(vcttuple_tuple(f1,s1)) == s1
    }
}
```

Listing 9.1: Axiomatization of `Tuple`

Modeling a map

There are four functions that together model the map: `keys`, `values`, `items` and `get` (see Listing 9.2). The first three functions keep track of the keys, values and pairs respectively and the `get` function keeps track of the mapping. The axiom `vctMapValuesAx` states that a value `v1` is in the set of values of `m1` iff there exists a key `k1` that is both in the keyset of `m1` and maps to `v1`. The axiom `vctMapItemsKeysAx` reads similarly.

```
function vctmap_keys(m:VCTMap[K,V]): Set[K]
function vctmap_values(m: VCTMap[K,V]): Set[V]
function vctmap_items(m: VCTMap[K,V]): Set[VCTTuple[K,V]]
function vctmap_get(m:VCTMap[K,V], k: K): V

axiom vctMapValuesAx {
  forall v1: V, m1: VCTMap[K,V] ::
    v1 in vctmap_values(m1) == (exists k1: K :: k1 in vctmap_keys(m1) && vctmap_get(m1, k1) == v1)
}

axiom vctMapItemsKeysAx {
  forall t1: VCTTuple[K,V], m1: VCTMap[K,V] ::
    (t1 in vctmap_items(m1)) <==>
    (vcttuple_fst(t1) in vctmap_keys(m1) && vctmap_get(m1, vcttuple_fst(t1)) == vcttuple_snd(t1))
}
```

Listing 9.2: The four functions modeling a map

The Dafny axiomatization is written in the Boogie programming language and uses Boogie maps. When translating the Dafny axioms into Silver, these Boogie maps need to be translated as well. The type of a Boogie map is written as `[U]V` where `U` is the type of the key and `V` is the type of the value. The function `get` is defined to behave like a map. For example, the axiom `vctMapItemsKeysAx` expresses that if a key/value pair `(a,b)` is in the map, the function `get` returns the value `b` given the key `a`.

Map constructors

The functions `empty` and `build` construct maps (see Listing 9.3). Starting from an `empty` map, key/value pairs can be added using the `build` function[2]. The axioms `vctMapEmptyKeyAx` and `vctMapEmptyValueAx` express the relation between an empty map and the functions `keys` and `values`. The axiom `vctMapBuildAx0` expresses that when a key/value pair `(k1,v1)` is added to the map using the `build` function, the key `k1` is in the key set and that the function `get` with `k1` maps to the given value `v1`.

```
function vctmap_empty(): VCTMap[K,V]
function vctmap_build(m: VCTMap[K,V], k: K, v: V): VCTMap[K,V]

axiom vctMapEmptyKeyAx {
  forall k1: K :: !(k1 in vctmap_keys(vctmap_empty())) && |vctmap_keys(vctmap_empty())| == 0
}

axiom vctMapEmptyValueAx {
  forall v1: V :: !(v1 in vctmap_values(vctmap_empty())) && |vctmap_values(vctmap_empty())| == 0
}

axiom vctMapBuildAx0 {
  forall k1: K, v1: V, m1: VCTMap[K,V] ::
    k1 in vctmap_keys(vctmap_build(m1, k1, v1)) && vctmap_get(vctmap_build(m1, k1, v1), k1) == v1
}
```

Listing 9.3: Constructors for a map

[2]These two constructors are similar to sequence constructors `Nil` and `Cons` where the former is an empty sequence and the latter is a sequence with a head and a tail.

Map destructor

In addition to the functions in DafnyPrelude, functions/methods/operators on maps in different programming languages were considered to be added as an extension. The Dafny axioms cover most basic functions except for removing a key/value pair from a map.

Listing 9.4 shows the function remove that has been added to support this operation. Six axioms define the behavior of the remove function in relation to the other functions (see Listing 9.4). The axioms can be read as follows:

- vctMapRemoveAx1: Key k is removed from map m, k is not in the key set of m.
- vctMapRemoveAx2: If a key k is a key of map m, then the value of k is not the same value after removing k.
- vctMapRemoveAx3: For any two distinct keys k1 and k2, if k1 is removed from the map, then k2 is still a key of m and maps to the same value.
- vctMapRemoveAx4: If a key k is a key of map m, then removing k decreases the cardinality of m by one.
- vctMapRemoveAx5: If a key k is not a key of map m, then removing k does not have an effect on the cardinality of m.
- vctMapRemoveAx6: If a key k is not a key of map m, then removing k results in the same map m.

```
function vctmap_remove(m: VCTMap[K,V], k: K): VCTMap[K,V]

axiom vctMapRemoveAx1 {
  forall m: VCTMap[K,V], k: K :: !(k in vctmap_keys(vctmap_remove(m, k)))
}

axiom vctMapRemoveAx2 {
  forall m: VCTMap[K, V], k: K ::
  (k in vctmap_keys(m)) ==> vctmap_get(vctmap_remove(m, k), k) != vctmap_get(m, k)
}

axiom vctMapRemoveAx3 {
  (forall m: VCTMap[K, V], k1: K, k2: K ::
  k1 != k2 ==>
  (k2 in vctmap_keys(vctmap_remove(m, k1)) == k2 in vctmap_keys(m) &&
  vctmap_get(vctmap_remove(m, k1), k2) == vctmap_get(m, k2)))
}

axiom vctMapRemoveAx4 {
  forall m: VCTMap[K,V], k: K ::
  (k in vctmap_keys(m)) ==> vctmap_card(m)-1 == vctmap_card(vctmap_remove(m, k))
}

axiom vctMapRemoveAx5 {
  forall m: VCTMap[K,V], k: K ::
  !(k in vctmap_keys(m)) ==> vctmap_card(m) == vctmap_card(vctmap_remove(m, k))
}

axiom vctMapRemoveAx6 {
  (forall m: VCTMap[K, V], k: K ::
  !((k in vctmap_keys(m))) ==> vctmap_equals(m, vctmap_remove(m, k)))
}
```

Listing 9.4: The remove function and its axioms

Functions on maps

In addition to the functions above, there are 3 functions defined on maps as defined in Section 9.1 (see Listing 9.5):

- `card`: The cardinality of the map. The cardinality of the map is defined to be the cardinality of the key set of the map (as expressed in the axiom `vctMapCardAx2`).
- `equals`: Check if two maps are equivalent. Two maps are equivalent iff their keysets are equivalent and the keys map to the same value in both maps (as expressed in the axiom `vctMapEqualsAx1`).
- `disjoint`: Check if two maps are disjoint. Two maps are disjointed if for all keys they are not in the keyset of one of the maps (as expressed in the axiom `vctMapDisjointAx1`).

```
function vctmap_card(m:VCTMap[K,V]): Int
axiom vctMapCardAx2 {
  forall m1: VCTMap[K,V] :: {vctmap_card(m1)} vctmap_card(m1) == |vctmap_keys(m1)|
}

function vctmap_equals(m1: VCTMap[K,V], m2: VCTMap[K,V]): Bool
axiom vctMapEqualsAx1 {
  forall m1: VCTMap[K,V], m2: VCTMap[K,V] :: {vctmap_equals(m1, m2)} vctmap_equals(m1, m2) <==>
  (
    vctmap_keys(m1) == vctmap_keys(m2)
    &&
    forall k: K :: k in vctmap_keys(m1) ==> vctmap_get(m1, k) == vctmap_get(m2, k)
  )
}

function vctmap_disjoint(m1: VCTMap[K,V], m2: VCTMap[K,V]): Bool
axiom vctMapDisjointAx1 {
  forall m1: VCTMap[K,V], m2: VCTMap[K,V] :: {vctmap_disjoint(m1, m2)} vctmap_disjoint(m1, m2) <==> (
      ↪ forall k: K :: {k in vctmap_keys(m1)} {k in vctmap_keys(m2)} !(k in vctmap_keys(m1)) || !(k in
      ↪  vctmap_keys(m2)))
}
```

Listing 9.5: The functions `card`, `equals` and `disjoint`

9.3 Implementation

With the Dafny axiomatization of a map, it follows that this feature is implemented using a domain. The concrete implementation steps (based on the general approach) are as follows:

1. Add the syntax as defined above.
2. Add the value `Map` to the `PrimitiveSort` enum.
3. Define `StandardOperators` where necessary.
4. Match the syntax in `PVLtoCOL` and transform it into COL.
5. Add the Silver domain to the file `prelude.sil` and add the VCTMap domain to the COL AST.
6. Transform the new operators into invokations of the domain functions.

The syntax

The ANTLR grammar is extended with the map constructor and the other syntax that is not already part of VerCors. Listing 9.6 shows the ANTLR grammar rule for the map constructor. The function syntax (as defined in Section 9.1) is added to the class `PVLSyntax`.

```
nonTargetUnit:
'map' '<' type ',' type '>' mapValues
;

mapValues : '{' ( | expr '->' expr (',' expr '->' expr)*) '}';
```

Listing 9.6: The ANTLR grammar rules for the map constructor

Map encoding in COL

Maps are encoded in COL as `StructValues`. As mentioned in Section 7.5, the AST node `StructValue` models structured values such as sequences, sets, bags, boxes, and tuples. The type of a `StructValue` depends on the type given during creation. For example, a sequence is encoded as a `StructValue` with a `PrimitiveType` of sort `Sequence`. Maps are encoded similarly by adding a new sort `Map` to the enum `PrimitiveSort`. Now we create the type of the map by defining a `PrimitiveType` with sort `Map` and create a map by defining a `StructValue` with the previously mentioned type.

Defining `StandardOperators` for the domain functions

There are ten functions in the `VCTMap` domain. Nine of these functions need a `StandardOperator` associated with them. The exception is the `empty` function which is used by the constructor syntax defined above. Listing 9.1 shows a mapping of the defined `StandardOperators` to the domain functions.

• `MapKeySet`	⟶	`vctmap_keys`
• `MapValueSet`	⟶	`vctmap_values`
• `MapItemSet`	⟶	`vctmap_items`
• `MapGetByKey`	⟶	`vctmap_get`
• `MapBuild`	⟶	`vctmap_build`
• `MapRemoveKey`	⟶	`vctmap_remove`
• `MapCardinality`	⟶	`vctmap_card`
• `MapEquality`	⟶	`vctmap_equals`
• `MapDisjoint`	⟶	`vctmap_disjoint`

Figure 9.1: `StandardOperators` and their respective domain functions

PVLtoCOL

The function syntax is handled automatically since it was added to PVLSyntax. The syntax for the constructor is matched and transformed into a StructValue (see Listing 9.7). The key/value pairs are matched by the method convert_pairs which loops over all the pairs and makes an array of ASTNodes. This array has to have an even length where the even indices point to keys and odd indices point to values.

```java
if(match(ctx, "map", "<", null, ",", null, ">",null)) {
  Type t1 = checkType(convert(ctx,2)); // Get the type of the keys
  Type t2 = checkType(convert(ctx,4)); // Get the type of the values
  ASTNode[] pairs = convert_pairs(ctx.getChild(6), "{",",","->","}");
  if (pairs.length %2 != 0 || Arrays.stream(pairs).anyMatch(Objects::isNull)) {
    Fail("Values␣of␣map␣are␣not␣pairs");
  }
  return create.struct_value(create.primitive_type(PrimitiveSort.Map, t1, t2), null, pairs);
}
```

Listing 9.7: Matching the constructor syntax

Adding the domains to VerCors

The full axiomatization in Appendix C is added (without the comments) to the file prelude.sil. In SilverClassReduction, the StandardOperators are rewritten as invokations of their respective functions. The StructValue with sort Map is also rewritten (see Listing 9.8). The map is initially empty (i.e. an invokation of the empty function). If there are key/value pairs in the map, they get added to the empty map by invoking the build function[3]. For example, if we would construct a map with $1 \to 2$ and $2 \to 3$, then this map would be constructed by build(build(empty(), 1, 2), 2, 3).

```java
public class SilverClassReduction extends AbstractRewriter {
    @Override
  public void visit(StructValue v) {
    if (v.type().isPrimitive(PrimitiveSort.Map)) {
      Type resultType = rewrite(v.type());
      ASTNode map = create.invokation(resultType,null,"vctmap_empty");
      for (int i=0; i < v.valuesArray().length; i+=2) {
        map = create.invokation(resultType, null, "vctmap_build", map, v.valuesArray()[i], v.valuesArray
            ↪ ()[i+1]);
      }
      result = map;
    }
  }
}
```

Listing 9.8: Rewriting a StructValue to the constructors in the VCTMap domain

[3]This resembles how sequences are axiomatically defined as Nil and Cons. Sequences are initially empty (i.e. the Nil constructor) and values are added to this empty sequence by using the Cons constructor.

9.4 Evaluation of the Implementation

We evaluate the implementation and the chosen triggers through a set of representative examples that should verify and by a visual inspection of the instantiation graphs of those examples in Axiom Profiler. The examples used for the evaluation can be found in Listing 9.12 and Appendix D[4]. The former is an example written for this thesis and the latter is based on an example from the Dafny GitHub repository.

9.4.1 The chosen triggers

Viper uses heuristics to generate possible triggers. These triggers are sufficient for examples with a few candidate trigger sets. As the examples become larger/more complex, more trigger sets are generated. An example is shown in Listing 9.9. Here we see the `vctMapBuild1Dot5Ax` axiom and the trigger sets generated by Viper as comments above[5]. Z3 tries to match any of these trigger sets, thereby slowing down the verification effort. For our running examples with no (explicitly defined) triggers, Z3 times out at 100s.

```
// Generated triggers
// {vctmap_build(m1,k1,v1), k2 in vctmap_keys(m1)}
// {vctmap_build(m1,k1,v1), vctmap_get(m1,k2)}
// {vctmap_keys(vctmap_build(m1,k1,v1)), k2 in vctmap_keys(m1)}
// {vctmap_keys(vctmap_build(m1,k1,v1)), vctmap_get(vctmap_build(m1,k1,v1),k2)}
// {vctmap_keys(vctmap_build(m1,k1,v1)), vctmap_get(m1,k2)}
// {k2 in vctmap_keys(vctmap_build(m1,k1,v1))}
// {vctmap_keys(m1), vctmap_build(m1,k1,v1), vctmap_get(m1,k2)}
// {vctmap_keys(m1), vctmap_get(vctmap_build(m1,k1,v1),k2)}
// {k2 in vctmap_keys(m1), vctmap_build(m1,k1,v1)}
// {k2 in vctmap_keys(m1), vctmap_get(vctmap_build(m1,k1,v1),k2)}
// {vctmap_get(vctmap_build(m1,k1,v1),k2)}
axiom vctMapBuild1Dot5Ax {
  forall k1: K, k2: K, v1: V, m1: VCTMap[K,V] ::
    (k1 != k2 ==>
      (
        (k2 in vctmap_keys(vctmap_build(m1, k1, v1))) == (k2 in vctmap_keys(m1))
      &&
        vctmap_get(vctmap_build(m1, k1, v1), k2) == vctmap_get(m1, k2)
      )
    )
}
```

Listing 9.9: The axiom `vctMapBuild1Dot5Ax`

It is also possible that the heuristics of Viper change in the future. This implies that some examples that did not work due to bad trigger choices, might work with the new heuristics or vice versa. In other words, choosing good triggers is important to both speed up the verification effort and keep the behavior consistent over time.

With these points in mind, it has been decided to explicitly define the trigger sets for the `VCTMap` domain axioms. The trigger sets are chosen according to the following strategy:

1. **Start off with the triggers in the `DafnyPrelude`**: These axiomatizations have matured over time since they are used by Dafny and other tools that follow `DafnyPrelude` closely (for example Silicon and Carbon).

[4]The second example has been moved to the appendix for readability.
[5]For an overview of the generated triggers for each axiom in `VCTMap`, see Appendix C.

2. **Use examples that should verify**: These are the two examples mentioned above.
3. **Consider the triggers generated by Viper**: For the cases that do not verify, we try to identify which axioms are related to it and if the triggers are sufficient. We consider the trigger sets generated by Viper and try to make the examples verify.

The first step is relatively simple, however some changes were made w.r.t. the triggers in `DafnyPrelude`. These changes can be summarized as follows:

- `No triggers for existential quantifier`: the existential quantifier in the `DafnyPrelude` have triggers defined for them. Triggers are used to guide the SMT solver to instantiate the body of a universal quantifier where needed. For existential quantifiers, triggers do not have any effect. The syntax for triggers on existential quantifiers is allowed by different tools such as Viper and Z3, however these are omitted for evaluation. For this reason, the triggers for the existential quantifiers are omitted.
- `Adding trigger sets`: Some trigger sets were added that could be used in a verification effort. For example, for the axiom `vctMapCardAx2` the only trigger is size of the key set of a map `m` (i.e. `|keys(m)|`). Since there is a function `card` that is defined to be the size of the map, `card(m)` is added as an extra trigger set.

9.4.2 Instantiation graphs

With these triggers, the two examples verify. With the examples working, we use Axiom Profiler to inspect the instantiation graph. This inspection gives us insight into how Z3 solves the problem and what terms are instantiated. Given that the examples do verify we do not expect an infinite matching loop.

The instantiation graphs for all methods can be found in Appendix E. A quick visual inspection of these graphs already tells us that not all cases are solved using triggers. We discuss these graphs in two groups: cases that are solved by trigger instantiation and cases that are solved by other strategies.

Trigger instantiation

The instantiation graphs are part of this group if they have colored nodes. The colored nodes are instantiations of the body of a quantifier based on its trigger. The graphs for methods `m`, `m2`, `m4`, `m6`, `m7`, `m9` and the `main` method in Listing 9.12 are part of this group.

We take the `main` method in Listing 9.12 as an example to evaluate the triggers since its instantiation graph is the most complex (see Figure 9.2). We can immediately see that there are paths with multiple purple nodes which points to a matching loop. The purple nodes represent the axiom `vctMapBuildAx1` (see Listing 9.10).

This matching loop is a consequence of the recursive nature of the map constructor. The map is initially empty and the key/value pair is added one by one. For example `map<int, boolean>{1 -> true, 2 -> false}` is translated as `build(build(empty(), 1, true), 2, false)`. A trigger of the form `build(m1, k1, v1)` matches the entire expression as well as `build(empty(), 1, true)`. Following the path from the top purple node to the last purple node we see that the top purple node matches the entire map and that the last purple node matches the inner-most `build` function. In other words, the map starts as the complete expression and is reduced to the base value which is the empty map. Since the expression is reduced to a base value, we conclude that the matching loop is finite.

The same holds about the instantiation graphs of the other methods in this group. There is only one matching loop when a concrete map is constructed.

```
axiom vctMapBuildAx1 {
  forall k1: K, k2: K, v1: V, m1: VCTMap[K, V] ::
    {k2 in vctmap_keys(vctmap_build(m1, k1, v1))}
    {vctmap_get(vctmap_build(m1, k1, v1), k2), vctmap_get(m1, k2)}
    k1 == k2 ==>
      k2 in vctmap_keys(vctmap_build(m1, k1, v1)) &&
      vctmap_get(vctmap_build(m1, k1, v1), k2) == v1
}
```

Listing 9.10: The axiom `vctMapBuildAx1` in the `VCTMap` domain

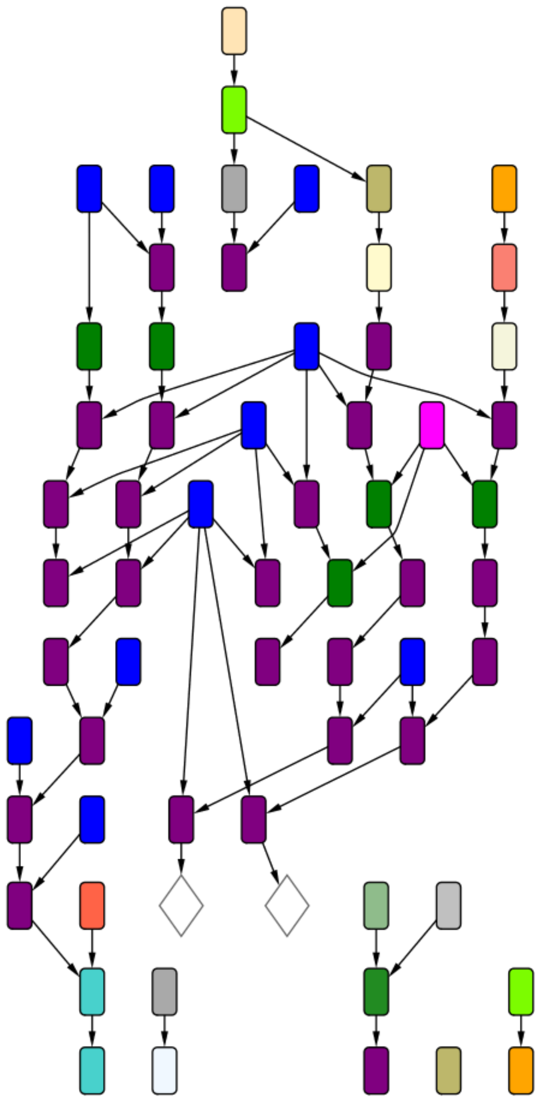


Figure 9.2: The instantiation graph for the `main` method in Listing 9.12

Other strategies

The remaining instantiation graphs are those for methods `m3`, `m5` and `m8`. All three of these graphs have no colored nodes. From this, we conclude that no triggers are instantiated. Given that the methods verify, there must be some other strategy Z3 uses.

The three methods have in common that they work with integers. Let us take the method `m3` and its instantiation graph as an example (see Listing 9.11 and Figure 9.3). The method has a map argument `a` and in the precondition it states that for all integers `i` between 0 and 100, that `i` maps to `(i * i)`.

The problem is solved in two steps. First, the problem is reduced by Silicon by introducing a fresh integer `i'` and the body of the quantifier is asserted with `i'`. Second, Z3's arithmetic solver and basic theory solver verify the problem without instantiating a term using a trigger.

We can see this behavior in the SMT2 files generated by Silicon/Viper and the instantiation graphs in Axiom Profiler. The SMT2 files generated by Viper show that a new integer is declared (for example `i@9@08`) and that the body of the quantifier is asserted or in other words it is added to the pool of knowledge.

The instantiation graph shows several diamond-shaped nodes. These nodes correspond to instantiations from either the arithmetic solver or the basic theory solver. As mentioned before, Axiom Profiler shows only a part of the instantiation graph. The entire graph consists of 366 diamond nodes. By going through these nodes, we see that the following terms are part of the knowledge pool:

- $0 <= 20$
- $20 < 100$
- $0 <= i'$
- $i' < 100$
- $vctmap\_get(a, i') == i' \times i'$
- $20 \times 20 == 400$

With these terms in the knowledge pool, Z3 is able to prove that the assertion $vctmap\_get(a, 20) == 400$ holds without instantiation based on triggers.

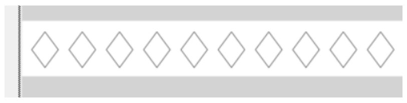


Figure 9.3: The instantiation graph for the `m3` method

```
requires (\forall int i; 0 <= i && i < 100; i in keysMap(a) && a[i] == i*i);
void m3(map<int, int> a) {
  assert a[20] == 400;
}
```

Listing 9.11: The method `m3` in PVL

9.5 Examples using Maps

The example in Listing 9.12 creates four maps named m1 to m4. The different function are used on these maps with both the function syntax and the alternative syntax.

```
class MapsInPVL {
  void main() {
    map<int, boolean> m1 = map<int,boolean>{1 -> true, 2 -> true, 0 -> false};

    assert m1 == map<int,boolean>{1 -> true, 2 -> true, 0 -> false, 1 -> false, 1 -> true};
    assert m1[1] && !m1[0];
    assert !isEmpty(m1);
    assert |m1 ++ (1, false)| == 3;
    assert getFromMap(m1, 1);
    assert |m1| == 3;
    assert getFromMap(map<int,boolean>{1 -> true, 2 -> true, 0 -> false}, 1);

    m1 = buildMap(m1, 1, true);
    assert !disjointMap(m1, map<int,boolean>{1 -> true});
    assert disjointMap(m1, map<int,boolean>{3 -> false, 4 -> true});

    set<tuple<int,boolean>> items = itemsMap(m1);
    assert tuple<int,boolean>{1, true} in items;


    map<int, boolean> m2 = map<int,boolean>{};
    m2 = buildMap(m2, 1, true);
    assert equalsMap(map<int,boolean> {1 -> true}, m2);

    assert removeFromMap(map<int, boolean> {0 -> true}, 0) == map<int,boolean>{};

    assert isEmpty(keysMap(map<int, boolean>{}));
    assert keysMap(map<int, boolean>{123 -> false}) == {123};
    assert valuesMap(map<int, boolean>{123 -> false}) == {false};

    assert valuesMap(map<int, boolean>{123 -> false, 2 -> false, 84368 -> true}) == {false, true};
    assert keysMap(map<int, boolean>{123 -> false, 2 -> false, 84368 -> true}) == {123, 2, 84368};


    map<int, boolean> m3 = map<int,boolean>{1 -> true, 2 -> true, 0 -> false};
    map<int, boolean> m4 = removeFromMap(m3, 1);

    assert m3[2];
    assert 0 in keysMap(m4);
    assert false in valuesMap(m4);
    assert m4[2];
    assert true in valuesMap(m4);
    assert removeFromMap(buildMap(map<int,boolean>{}, 0, true), 0) == map<int,boolean>{};
  }
}
```

Listing 9.12: An example using maps in PVL

Chapter 10

Generics

This chapter answers the questions for RQ6 on generic classes and functions. Section 10.1 answers RQ6.1, showing the definition of generic classes and generic functions, their scope and syntax. Section 10.2 answers RQ6.3 by going into detail on the design and translation of generics into Silver. Sections 10.3.2 and Section 10.3.4 answer RQ6.2 by discussing type checking for generic classes and generic functions respectively.

The implementation of generics (as described in this chapter) can be found in commit `b835bc7` which can be browsed at `https://github.com/OmerSakar/vercors/tree/b835bc72575d263ba3376a6c02522fbb0d9c4106`.

10.1 Description and Syntax

The syntax for generics is an extension of the current syntax for class types, constructors and function signatures. The extension allows for type parameters to be defined, (e.g. `MyGenericClass<int,boolean>` to represent a concrete instance of `MyGenericClass` with `T` $\leftarrow$ `int` and `R` $\leftarrow$ `boolean`).

Generic Class

Syntax:

```
class MyGenericClass<T,R> {...}
```

Arguments:

T - A type parameter
R - A type parameter

Description:

Defines a generic class with type parameters `T` and `R`. These type parameters can be used as field types or in functions. A generic class requires at least one type parameter.

Generic function

Syntax:

```
pure <T,R> returnType myGenericFunc(args) = exp;
```

Arguments:

T - A type parameter
R - A type parameter
returnType - The return type of the function. This type can be of type `T` or `R`
args - The arguments of the function
exp - The body of the function

Description:

Defines a generic function with type parameters `T` and `R`. These type parameters can be used as argument types or the return type. A generic function requires at least one type parameter.

10.2 Design

10.2.1 Monomorphizing generic classes and generic functions

Encoding generics into Viper has been done by Matthias Erdin [32]. His Master's thesis focuses on generics, type states and traits in the programming language Rust and implements the verification of these Rust constructs in the formal verification tool Prusti [33]. The general idea is to monomorphize structs (comparable to a class without fields in PVL) and generic functions in Rust. Monomorphization in this context is creating a copy of the generic class/function and substituting the type parameters of those classes/functions with their concrete instances. This idea is extended by also monomorphizing pure functions in the generic class.

As an alternative to monomorphization, an approach was considered to encode generics using domains. This approach seems the most straightforward since domains have type parameters and thus do not have to be monomorphized. However, this approach was abandoned due to the difficulty of translating a function in an imperative language into a set of axioms defined on that function. For example, a function must require permissions on fields of classes that it uses those fields (e.g. the field of an argument). This cannot be expressed using a domain axiom as field accesses and permissions are not allowed.

A generic class is monomorphized by the following steps:

1. Search for all concrete instantiations of the generic class and keep track of a mapping of the type parameters to the concrete types. For example, the statement `new MyGenericClass<boolean, int>` results in a mapping of `T` $\leftarrow$ `boolean` and `R` $\leftarrow$ `int`.
2. Generate a new class (based on the generic class) for each mapping found in the previous step.
3. Rename the constructor to match the generated class if there is a constructor.

A generic function outside of a generic class is monomorphized similarly. The major difference is in how the mapping for the type parameters is determined. For generic classes the mapping of the type parameters is provided by the user in the syntax, for instance for `MyGenericClass<boolean, int>` the first type parameter is mapped to a `boolean` as provided by the user. However, the user does not provide a type in a function invokation. Consider the pure function `pure <T,K>`

`T myFunc(T arg1, T arg2, K arg3) = arg1`. This function has to be called with the first two arguments of the same type. The algorithm used to determine the mapping is discussed in Section 10.3.4.
To summarize, a generic function is monomorphized by the following steps:

1. Find all instances of the function.
2. Determine the mapping for the type parameters based on the arguments of the function.
3. Generate a new function (based on the generic function) for each mapping found in the previous step.

10.2.2 The verification of the generic class

In addition to verifying the monomorphized classes, the generic class can also be verified. This is achieved by introducing the type parameters as actual types. This idea is realized in Silver by introducing a domain for each type parameter. This step is sufficient to support the verification of generic classes since domains are types in Silver.

10.2.3 Concrete example

All these ideas combined form the concept of generics in PVL. Listing 10.1 shows a concrete case with its Silver equivalent in Listing 10.2[1]. The Silver code is relatively long as a consequence of the monomorphization. It consists of several parts:

- Two domains `L` and `R` for the type parameters.
- Fields for the generic class (e.g. `MyGenericClass_myField3`).
- Fields for the concrete instances (e.g. `MyGenericClass_Boolean_Integer_myField2`).
- The functions of the generic class (e.g. `MyGenericClass_getL` with a return type `L`).
- The functions of the concrete instances (e.g. `MyGenericClass_Boolean_Integer_getL` with a return type `Bool`).
- The rest of the methods, functions and fields, in this case the only additional method is the `main` method of `NormalClass`.

Some of the functions have an argument `diz` (as generated by VerCors) which refers to the current object similar to Java's `this` or Python's `self`.

[1]The constructors of the concrete instances have been omitted to simplify the example.

```
class MyGenericClass<L,R> {
  int myField1;
  L myField2;
  R myField3;

  requires Perm(myField2, write);
  pure L getL() = myField2;

  requires 0 <= i && i < |xs|;
  ensures |\result| <= |xs|;
  ensures (\forall int k; 0 <= k && k < |\result|; \result[k] == xs[k]);
  pure seq<R> take(seq<R> xs, int i) = xs[..i];
}

class NormalClass {
  void main() {
    MyGenericClass<boolean,int> mac2 = new MyGenericClass<boolean,int>();
    mac2.myField2 = true;
    boolean LofMac2 = mac2.getL();
    assert mac2.getL();
  }
}
```

Listing 10.1: An example of a generic class

```
domain L {} // Type parameter L
domain R {} // Type parameter R

// The fields of the generic class
field MyGenericClass_myField1: Int
field MyGenericClass_myField2: L
field MyGenericClass_myField3: R
// The fields of the concrete instance
field MyGenericClass_Boolean_Integer_myField1: Int
field MyGenericClass_Boolean_Integer_myField2: Bool
field MyGenericClass_Boolean_Integer_myField3: Int


/////////////////////////////////////
// Functions of the generic class //
/////////////////////////////////////
function MyGenericClass_getL(diz: Ref): L
requires acc(diz.MyGenericClass_myField2, write)
{ diz.MyGenericClass_myField2 }

function MyGenericClass_take_Sequence_R_Integer(diz: Ref, xs: Seq[R], i: Int): Seq[R]
requires 0 <= i && i < |xs|
ensures |result| <= |xs|
ensures (forall k: Int :: 0 <= k && k < |result| ==> result[k] == xs[k])
{ xs[..i] }

/////////////////////////////////////////
// Functions of the concrete instances //
/////////////////////////////////////////
function MyGenericClass_Boolean_Integer_getL(diz: Ref): Bool
requires acc(diz.MyGenericClass_Boolean_Integer_myField2, write)
{ diz.MyGenericClass_Boolean_Integer_myField2 }

function MyGenericClass_Boolean_Integer_take_Sequence_Integer_Integer(diz: Ref, xs: Seq[Int], i: Int):
     ↪ Seq[Int]
requires 0 <= i && i < |xs|
ensures |result| <= |xs|
ensures (forall k: Int :: 0 <= k && k < |result| ==> result[k] == xs[k])
{ xs[..i] }

///////////////////////////////////////
// The main function of NormalClasss //
///////////////////////////////////////
method NormalClass_main(diz: Ref)
requires diz != null
{
  var mac2_1: Ref
  var LofMac2_2: Bool

  mac2_1 := MyGenericClass_Boolean_Integer_MyGenericClass_Boolean_Integer()
  mac2_1.MyGenericClass_Boolean_Integer_myField2 := true
  LofMac2_2 := MyGenericClass_Boolean_Integer_getL(mac2_1)
  assert MyGenericClass_Boolean_Integer_getL(mac2_1)
}
```

Listing 10.2: The Silver equivalent of Listing 10.1

10.3 Implementation

The implementation of generics is relatively complex (compared to the other features discussed in this thesis) and combines multiple approaches described in Chapter 6 although there are similarities with function generation. Instead of generating functions, concrete classes are generated by copying the generic class.

The implementation is explained in three parts: a common part (Section 10.3.1), generic classes (Section 10.3.2) and generic functions (Section 10.3.3). The reason for this division is that the implementation of generic classes and functions differs enough that they are best explained separately.

10.3.1 Common part of the implementation

The generic class is encoded as `ASTNodes` of the subclass `ASTClass` with the type parameters as `TypeVariables` (with the same name). `ASTClass` has a field `kind` representing the kind of class. Generic classes are of kind `Abstract` and normal classes are of kind `Plain`. It is assumed that a class of kind `Abstract` has type parameters.

Generic functions are encoded as `ASTNodes` of the subclass `Method` with the type parameters as `TypeVariables` (with the same name). `Methods` also have a kind, however there is no special kind for a generic function. Since we only support pure generic functions, generic functions are of kind `Pure` and have at least one type parameter.

The type parameters should be introduced as new types, however this cannot be achieved by simply introducing a new class with the same name at this point. This is because the scope of these type parameters is the class or function. To introduce the type parameters as actual types in a certain scope, the Java class `ASTFrame` is used. `ASTFrame` keeps track of a stack of loaded classes and methods. Upon entering an `ASTClass` in the COL AST, that class is pushed on the stack and upon leaving the class is popped off the stack. The same holds for methods.

This behavior is extended by pushing an empty `ASTClass` onto the stack of classes for each of the type parameters (with the same name). Upon leaving, those classes are popped from the stack.

The type parameters are eventually represented in Silver as domains. This translation is done by the rewrite pass `SilverClassReduction` (as used in the approach for supporting domains). This pass is preceded by several type checks. From these type checks, we can conclude that a type parameter is not used outside its scope. This means that we can introduce the type parameters as actual classes at this point. Therefore, `SilverClassReduction` introduces a new, empty `ASTClass` of kind `Abstract` to the list of classes for each type parameter. These `Abstract` `ASTClasses` are translated into domains during the final step.

10.3.2 Generic classes

Generic classes are monomorphized by a new rewrite pass `MonomorphizeGenericClass`. This class has three tasks. The first is rewriting all constructors and class types to match the generated class by appending the concrete types to the name of the class. For example, if the generic type is `GenericType<int, myClass>`, the generated type is rewritten to `GenericType_int_myClass`.

The second task is to find all concrete instances of the generic type. The `visit` method with argument type `ASTClass` is overwritten to visit all normal classes using a new visitor pass `GenericsScanner`. `GenericsScanner` overwrites the `visit` method with argument `ClassType`. If this class type uses type parameters, then the mapping is kept of the `ClassType` to a list of

Types in a (Java) map. The list is the mapping of type parameters to concrete types. After all mappings are collected by the pass GenericsScanner, the pass MonomorphizeGenericClass continues.

The third task is to generate the concrete instances of the generic class. The rewriteAll method of MonomorphizeGenericClass is overwritten[2]. After performing the tasks above, the new rewrite pass GenerateGenericClassInstance is used for each mapping found. The input for this pass is the generic class to monomorphize and the mapping of type parameters to concrete types. Each instance of a type parameter is rewritten to its concrete type. The output of this pass is a class with no instances of the type parameters. The name of this class is generated by appending the concrete types to the name of the class. This class is then added to the list of classes in the program.

10.3.3 Generic functions

Generic functions are monomorphized by a new rewrite pass MonomorphizeGenericFunctions in a similar way. This pass has two tasks. The first task is to find all invokations of generic functions. A function invokation is encoded as a MethodInvokation with a reference to the invoked method/function. This reference is set during type checking. How it is determined which generic function corresponds to a function invokation is discussed in Section 10.3.4.

The second task is to generate the concrete instances of the generic function. The rewriteAll method of the MonomorphizeGenericFunctions is overwritten. After performing the task above, the new rewrite pass GenerateGenericFunctionInstance is used for each mapping found. The input for this pass is the generic function to monomorphize and the mapping of type parameters to concrete types. Each instance of a type parameter is rewritten to its concrete type. The output of this pass is a function with no instances of the type parameters. This function is then added to the same ASTClass as the generic function.

10.3.4 Determine mapping for a generic function.

During type checking and generating the concrete instances for a generic function, it is necessary to determine a mapping of type parameters (of a generic function) to concrete types based on the function invokation and the arguments of a generic function. We treat the type of the arguments of a generic function and the corresponding type of the argument of an invokation as trees and traverse those trees simultaneously in post order (i.e. children first). The types at each level should be the same unless that type is a type variable. When a type variable is encountered, we save a mapping of the type parameter to the concrete type in a (Java) map.

For example, if we need to match the type `Edge<T, seq<R>>` to an instance of type `Edge<int,seq<boolean>>`, we traverse both types simultaneously as shown in Figure 10.1. In the first step we determine that T is mapped to an integer and in the second step we determine that R is mapped to boolean. This continues until both types are compared.

If this process ends successfully, the result is a mapping for each type parameter. This process can fail in two ways; Either the type does not match or there are two (concrete) mappings for the same type parameter. For example, a function with the signature `myFunc(seq<T> a, seq<T> b)` cannot be invoked with a a sequence of integers and b a sequence of booleans since T is mapped to two different types (integers and booleans).

[2]This is similar to how functions are generated after identifying which functions to generate.

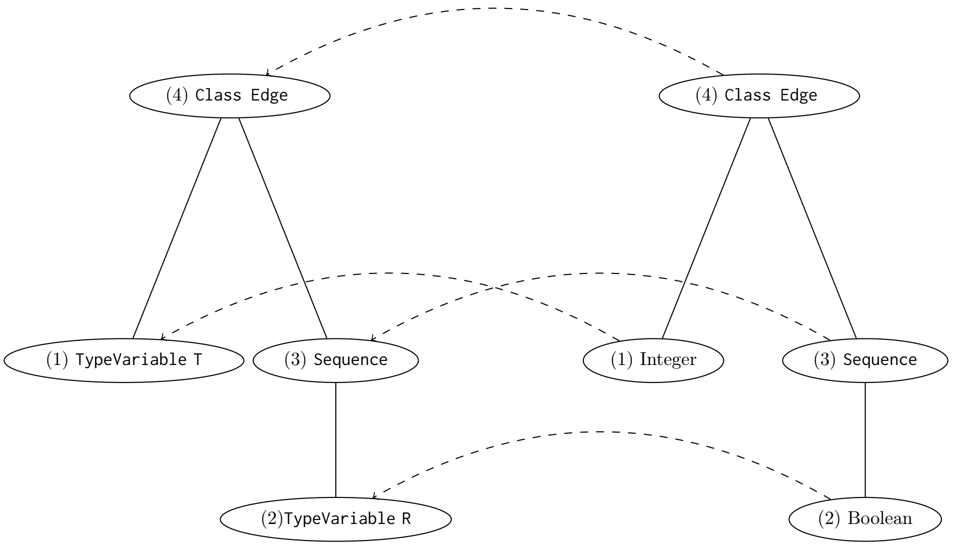


Figure 10.1: Mapping the type of the argument of a generic function (left) to the type of the argument of an invokation of a generic function (right)

10.4 Limitations

10.4.1 No support for generic methods

Methods in a generic class or generic methods are currently not supported. This limitation stems from the fact that methods are not pure and combined with generics they give rise to permission issues. Consider the PVL example in Listing 10.3. We have a class `MyObj` with a field `n` where the permissions for `n` is stored in a lock invariant. We have class `GenericClass` with a type parameter with a method `foo` that takes an argument of type `T`.

We get an instance of `MyObj`, lock on that instance to get the permissions for the field `n`, set `n` to an arbitrary value and unlock to release the permissions. We call the `foo` method on an instance of `GenericClass` and fail to assert that the value of `n` has not changed. This is due to the usage of the lock as `foo` can acquire the lock and change the value of `n`. This could be solved by ensuring (in the postcondition of `foo`) that the value of `n` has not changed, however that postcondition cannot be stated for the generic method since the type parameter `T` has no fields.

It is possible to support generic methods by supporting type parameters with fields (discussed in Section 10.5). The problem above can be solved this way by having a postcondition on the `foo` method stating that the field `T.n` has not changed.

```
class MyObj {
  int n;

  resource lock_invariant() = Perm(this.n, 1);

  MyObj() {}
}
class GenericClass<T> {
  void foo(T bar) {}
}
class NormalClass {
  void main() {
    GenericClass<MyObj> genericObj = new GenericClass<MyObj>();
    MyObj myObj = new MyObj();
    lock myObj;
    myObj.n = 2;
    unlock myObj;
    genericObj.foo(myObj);
    lock myObj;
    assert myObj.n == 2;
    unlock myObj;
  }
}
```

Listing 10.3: An example of a permission issue in a generic method

10.4.2 No static functions in generic classes

Static functions in generic classes are not allowed. This decision is based on the fact that the type parameters of the generic class are not visible to the function, this is similar to Java.

10.5 Future Improvements

This section goes into possible future improvements. These improvements were originally planned, however due to their complexity and time constraints they are not part of this thesis.

An extension on the type parameters

Currently, type parameters cannot have fields. This has the consequence that no permissions can be inhaled or exhaled on those fields. This limits the functionality that can be expressed in the functions.

This can be improved by allowing Java-like interfaces for type parameters. These interfaces state which fields the type parameter has. During the passes, it can be checked if the class in the concrete case matches the interface. It is also possible to allow inheritance for type parameters. For example, if some type parameter `T` has to be a subclass of a superclass `SuperClass`, we can say that `T extends SuperClass` to allow only subclasses of `SuperClass`.

In Section 10.2, it was stated that type parameters are translated into Silver as empty domains. This idea can be extended by generating uninterpreted functions that return the value of the field.

The encoding of these improvements is possible by combining the notion of abstract predicate families to encode the type parameters with all its fields and encoding generic types as uninterpreted constants and have axioms describe the hierarchy of the classes (as OpenJML does).

However, these topics need more research for a definitive solution (both in the design part as in the implementation in VerCors).

Allow mapping type parameters to other mapping types

Type parameters have to be mapped to concrete cases (e.g. using a generic class inside of another generic class). The difficulty of this task is that for any type parameter you need to find all possible mappings. Listing 10.4 shows a simple example of the complexity. We have two generic classes `A` and `B` with both two type parameters. The class `Normal` has two fields of type `B` with concrete instances for the type parameters. `B` has a field of type `A`. This should result in two monomorphized instances of `B` (for the two mappings) and two monomorphized instances for `A`.

This example is for a (relatively) simple hierarchy where the class `Normal` uses class `B` which uses class `A`. When other common mechanisms such as inheritance are introduced, this problem becomes complex. Again, this topic needs more research for a definitive solution (both in its design as in its implementation in VerCors).

```
class A<T,R>{}

class B<Q,S>{
  A<Q,S> f;
}

class Normal {
  B<int,boolean> b1;
  B<boolean,int> b2;
}
```

Listing 10.4: An example of the complexity of type parameters instantiated by other type parameters

10.6 Examples using Generics

The example in Listing 10.5 shows an example where generic classes and functions are used. We have a class `NormalClass` with its `main` method. This method instantiates an instance of the abstract class `MyAbstractClass`. Next, a sequence is constructed with values from 1 to 6. The first 2 elements are taken from these sequences using two functions `take` and `Take`. `take` is a generic function and `Take` is an instance of this function (taken from the VerCors examples directory). Using the same arguments, it is shown that the concrete instance of `take` and `Take` are equivalent.

```
class MyAbstractClass<L,R> {
  int myField1;
  L myField2;
  R myField3;

  requires Perm(myField2, write);
  pure L getL() = myField2;

}

class NormalClass {
  requires 0 <= i && i <= |xs|;
  ensures (0 < i) ==> (\result == seq<R> {head(xs)} + take(tail(xs),i-1));
  ensures !(0 < i) ==> (\result == seq<R>{});
  pure static <R> seq<R> take(seq<R> xs, int i) =
      0 < i ? seq<R> {head(xs)} + take(tail(xs),i-1) : seq<R>{};


  requires 0 <= n && n <= |xs|;
  ensures (0 < n) ==> (\result == seq<int> { head(xs) } + Take(tail(xs), n - 1));
  ensures !(0 < n) ==> (\result == seq<int> {});
  static pure seq<int> Take(seq<int> xs, int n) =
      0 < n ? seq<int> { head(xs) } + Take(tail(xs), n - 1) : seq<int> { };

  void main() {
    MyAbstractClass<boolean,int> mac2 = new MyAbstractClass<boolean,int>();
    mac2.myField2 = true;
    boolean LofMac2 = mac2.getL();
    assert mac2.getL();

    seq<int> xs = seq<int> {1, 2, 3, 4, 5, 6};
    int i = 2;
    assert NormalClass.take(xs, i) == NormalClass.Take(xs, i);
  }
}
```

Listing 10.5: An example using a generic class and generic function in PVL

Chapter 11

Conclusion

This chapter concludes this thesis. We started with two general questions RQ1 on what features regarding ADTs were desired by users and RQ2 on how to support those features (see Figure 11.1). RQ1 has been answered by conducting a survey and consulting different documentation on VerCors resulting in the following list of features to support:

1. Appending and prepending values to sequences.
2. Taking ranges/subsequences from a sequence.
3. Removing elements from a sequence.
4. Simple syntax for sequence, set and bag creation.
5. Subset notation for sets and bags.
6. Set comprehension.
7. Maps with basic operations.
8. Generic classes and functions.

RQ2 has been answered by presenting an implementation-level view of VerCors in Chapter 5 and identifying the following five (general) approaches to implement a feature in Chapter 6:

1. Pure syntactic sugar
2. Transformed syntactic sugar
3. Mapping directly to back end
4. Function generation
5. Domains

RQ2 has been specified in Chapter 4 based on the answer of RQ1 resulting in RQ3 to RQ6 (see Figure 11.1). RQ3 covers the first five features. These features have been grouped since they are (relatively) simple in their design and implementation. Their definition and implementation are discussed with a focus on their implementation in Chapter 7.

The other three features are more complex in design and discussed separately with a focus on their design. Chapter 8 answers RQ4 by defining set comprehension (RQ4.1) and discussing its encoding using function generation (RQ4.2). Chapter 9 answers RQ5 by defining maps (RQ5.1). A list of the supported functions is presented based on the Dafny map axiomatization, thereby answering RQ5.2 and RQ5.3. Chapter 10 answers RQ6 by defining generic classes and functions

(RQ5.1). It is discussed how generic classes and functions are monomorphized, thereby answering RQ6.3 and how type checking works for generics with a focus on finding the generic function corresponding to its invokation, thereby answering RQ6.2.

RQ1 What ADTs or functionality on ADTs is desired from VerCors/PVL?

RQ2 How can the architecture of VerCors and the back end Viper support the functionality of RQ1?

RQ3 For features number 1 to 5:

- **RQ3.1** What is the definition of the functionality?
- **RQ3.2** Which of the five approaches in Chapter 6 can be applied to implement the functionality?

RQ4 For set comprehension:

- **RQ4.1** What is the definition of set comprehension?
- **RQ4.2** How is set comprehension encoded into Viper?

RQ5 For maps:

- **RQ5.1** What is the definition of a map?
- **RQ5.2** What operations are defined on maps?
- **RQ5.3** How can Viper Domains be used to implement a map?

RQ6 For generic classes/functions:

- **RQ6.1** What is the definition of generic classes/functions?
- **RQ6.2** How should type checking work for generics classes/functions?
- **RQ6.3** How does the verification of generic classes/functions work?

Figure 11.1: All research questions

These features were all implemented in a fork of VerCors. This fork can be found at https://github.com/OmerSakar/vercors with links to the specific features at the start of their respective chapters. Not all features are currently in the latest release of VerCors yet. The features that are currently part of VerCors are features 1, 3 and 4. Features 5, 6, 7 and 8 are planned to be merged in the near future once their implementation is reviewed and accepted. There is one exception which is feature 2 (taking ranges from sequences). This feature was implemented by another developer in parallel to this thesis. That implementation used the Viper range operators instead of our approach (of generating a function). A small discussion was held and it was decided that the former implementation would be merged into VerCors, due to its simplicity.

11.1 Summary of Contributions

Altogether, this thesis contributes in two ways. First, documenting the process of implementing features and identifying general approaches to implement a feature in VerCors. Second, the result of those processes are the implemented features/the encodings of the ADTs in VerCors.

The documentation of the process should help both future VerCors developers in implementing features into VerCors specifically and developers of other tools that use Viper as its back end. The implemented features should help the user (both experts and new users) to write more expressive code, especially set comprehension, maps and generic classes and function.

To sum up the contributions:

- A detailed explanation of the VerCors architecture from an implementation-level view (Chapter 5).
- A detailed explanation on general approaches to implement a feature in VerCors (Chapter 6).
- Concrete examples of implemented features in VerCors (Chapter 7).
- A discussion of the encoding of set comprehension in Viper and the behavior of Z3 in solving problems using set comprehension (Chapter 8).
- A discussion of the map axiomatizations in Viper (based on the Dafny axiomatization), an evaluation of the chosen triggers and the behavior of Z3 in solving problems using maps (Chapter 9).
- A discussion on the encoding of generics in Viper with a basis that can be used by future work (Chapter 10).

Implemented features

In addition to the contributions mentioned above, the actual implementations of the features in VerCors are also contributions. All features implemented for this thesis are listed below:

- The `isEmpty` function that checks if its argument is empty (Section 7.1). The argument can be a sequence, set, bag or map and can be extended for other ADTs by implementing the `size` operator for those ADTs. This feature is implemented using *pure syntactic sugar*.
- The `append` and `prepend` operators (Section 7.2). These operators respectively append and prepend a value to a sequence. This feature is implemented using *transformed syntactic sugar*.
- The `take`, `drop` and `range` functions and their respectively operators (Section 7.3). These functions take a range of elements from a sequence where `take` takes the first `n` elements, `drop` drops the first `n` elements and `range` takes a range from a lowerbound to an upperbound. Not merged into VerCors as explained above. This feature is implemented using *function generation*.
- The `remove` function that removes an element from a sequence by its index (Section 7.4). This feature is implemented using *function generation*.
- Simple constructors for sequences, sets and bags which do not need a user-provided type if it has values (Section 7.5). This feature is implemented using *transformed syntactic sugar*.
- The <= and < operators defined on sets and bags as a subset notation (Section 7.6). This feature is implemented by *mapping directly to the back end operator*.

- A construct for set comprehension (Chapter 8). Set comprehension is modeled using a bodyless function returning a set with the specified elements. This feature is implemented using *function generation*.
- A map ADT with basic operations (Chapter 9). The axiomatization is based on the Dafny map axiomatization written in BPL (Boogie Programming Language) and are translated into Silver (see Appendix C). This feature is implemented using a *domain*.
- Generic classes and functions (Chapter 10). Generic classes and functions are monomorphized (i.e. copied and the type parameters substituted with the concrete types). This idea was taken from the Master's thesis of Matthias Erdin [32]. This idea was extended to also verify the generic classes and functions themselves (instead of only verifying the monomorphized versions) by introducing the type parameters as actual classes. This feature is implemented using a combination of approaches.

11.2 Future Work

Besides the future improvements mentioned in Section 10.5, there are three more ways to follow up on this thesis. The first is a user study to evaluate the results of this thesis (as mentioned in Chapter 1). This user study can either compare a version of VerCors with the new features to an older version. VerCors could also be compared to other tools that have a similar feature set. With the current set of new features, the user study would not be enough for a master's thesis. It could be enough if it is combined with a larger set of features.

The second future work would be supporting higher-order functions. There were several features requested in the survey that can be defined using higher-order functions, such as min/max functions and sorting functions. It would be interesting to support these. Both examples require their arguments (either sequences, sets and bags) to have an order. In VerCors, integers are ordered, however self-defined types do not have an order. This order could be defined using a higher-order function. It was considered to implement this feature as part of this thesis, however the estimated time for this feature did not fit in the planning.

It was found later that Benjamin Weber has tackled this problem in his Master's thesis [34] and implemented his solution in Nagini, an automated verifier for concurrent Python programs that uses Viper as its back end [35]. It would be interesting to see if the same idea can be implemented in VerCors to support lambda functions in Java.

The third future work is a proof of the soundness of the map axiomatization in Appendix C. The map axiomatization has been based on Dafny's map axiomatization. It has been used by different tools and for this thesis, it is assumed to be sound. However, no proof could be found whether the set of axioms was sound. This could be proven using a theorem prover such as Isabelle/HOL.

Appendix A

Survey Results

The results can be found in Table A.1[1]. To anonymize the results and have a readable overview, the results are summarized. To fit the results into a single table, the questions have been separated from the table. The questions are repeated below and their numbers correspond to the question numbers in Table A.1.

Questions

1. Have you used PVL before?
2. Which of the currently supported ADTs have you used before?
3. Were there any auxiliary functions you wrote for those ADTs which could be used in general?
4. Was there ever a point where you were looking for a specific ADT which was not supported? Did you instead model the problem using a supported ADT? If so, could you explain what you wanted to model and how you solved it?
5. What new functionality/ADT would help you with your software verification in PVL?

[1]A single result has been omitted since the result did not have any usable information. The person did not use PVL and for those who did not use PVL, the question was phrased differently to *What is a must for any verification tool*. The answer listed features that were already part of PVL.

Question 1	Question 2	Question 3	Question 4	Question 5
Yes	Sequences	A get function for sequences since accessing an element by its index is not possible in some constructs.	Always use sequences and sequences of sequences.	See previous question.
Yes	Sequences, Bags	Summing over sequences, min and max functions over bags	I'm always thinking about simple structured data in terms of sum (aka union-types), product and recursion. Product and recursion are there (I model it by making an object, Java-style) and sum can be hacked by using null values (and the invariant that at most one of the fields is non-null).	Higher-order functions: a map function, a fold function, a traverse function (generalizes the map and fold functions, although less important if map and fold exit).
Yes	Sequences, Sets	Checking if a sequence is a permutation of another.	Maps (used an array instead)	Maps and subset notation would be helpful.
Yes	Sequences, Bags, Sets	See the VerCors example directory	Pairs/Tuples. Modelled using sequences. Mappings (for example, the maps of Scala).	Custom ADT's such as in Viper. Pairs, Triples or Tuples
Yes	Sequences	-	Pointers	Maps could be useful
Yes	Sequences	No	A sequence with per element permissions, implemented using a domain.	Sequence with per element permissions. A way to specify a method is run by multiple threads. Functionality to simulate OpenCL Barriers.
Yes	Sequences, Bags, Sets	Comparison of all objects in a seq/bag/set with another object.	No	Simple sequence, set and bag constructors. Maps.
Yes	Sequences, Bags, Sets	No.	-	Futures and histories have to be specified in a process algebra. Ability to use (a subset of) mcrl2 and importing mcrl2 ADT
Yes	Sequences	-	-	Multidimensional arrays

Table A.1: The results of the survey

Appendix B

Isabelle Proof for the Proper Subset Operator

The automated theorem prover Isabelle has been used to show that the rewrite rule for the proper `SubSet` operator is correct. Listing B.1 shows the proof. `xs` and `ys` are sets and since sets in Viper are finite, it is assumed that they are finite by using the `finite` function. Then it is shown that $(|xs| < |ys| \land xs \subseteq ys) \implies xs \subset ys$.

```
theory SubSet
imports Main
begin

lemma SubSetinSubSetEq:
    "(finite xs ∧ finite ys ∧ card xs < card ys ∧ xs ⊆ ys) ==> xs ⊂ ys"
by blast

end
```

Listing B.1: The new proper subset expressed with existing Viper operators

Appendix C

Axiomatization of Maps

The entire Dafny `Map` axiomatization has been translated into Viper/Silver (see Listing C.2 below). The specific version of the `DafnyPrelude.bpl` file can be found at `https://github.com/dafny-lang/dafny/blob/14d0af1074f12981a32b43313fefe3cc2483d4e8/Binaries/DafnyPrelude.bpl`. All functions in `DafnyPrelude.bpl` have been translated into Silver with the exception of the function `Glue` (see Listing C.1). This function is related to a Boogie map and a type `Ty` which is used internally by Dafny. This function is omitted from our translation since it serves no purpose for the `Map` domain itself.

For future reference, the Boogie code corresponding to the Silver code has been added as a comment above each function/axiom. Also, the triggers that are generated by Viper have been added as a comment. These triggers have been generated by Viper using the VSCode plugin (version `2.2.2`).

```
function Map#Glue<U, V>([U] bool, [U]V, Ty): Map U V;
axiom (forall<U, V> a: [U] bool, b:[U]V, t:Ty ::
   { Map#Domain(Map#Glue(a, b, t)) }
   Map#Domain(Map#Glue(a, b, t)) == a);
axiom (forall<U, V> a: [U] bool, b:[U]V, t:Ty ::
   { Map#Elements(Map#Glue(a, b, t)) }
   Map#Elements(Map#Glue(a, b, t)) == b);
axiom (forall<U, V> a: [U] bool, b:[U]V, t:Ty ::
   { $Is(Map#Glue(a, b, t), t) }
   $Is(Map#Glue(a, b, t), t));
```

Listing C.1: The `Map#Glue` function

Listing C.2: The Silver translation of the `Map` axiomatization in `DafnyPrelude.bpl`

```
domain VCTTuple[F,S] {
  function vcttuple_tuple(f:F, s:S): VCTTuple[F,S]

// function _System.Tuple2._0(DatatypeType) : Box;
  function vcttuple_fst(t:VCTTuple[F,S]): F

// function _System.Tuple2._1(DatatypeType) : Box;
  function vcttuple_snd(t:VCTTuple[F,S]): S

  axiom vctTupleFstAx {
    forall f1:F, s1:S :: {vcttuple_tuple(f1,s1)} vcttuple_fst(vcttuple_tuple(f1,s1)) == f1
  }

  axiom vctTupleSndAx {
    forall f1:F, s1:S :: {vcttuple_tuple(f1,s1)} vcttuple_snd(vcttuple_tuple(f1,s1)) == s1
  }
}

/////////////////////////// DAFNY ///////////////////////////
// type Map U V;
domain VCTMap[K,V] {

/////////////////////////////////////////////////////
/////////// AXIOMS THAT ARE NOT INCLUDED ///////////
/////////////////////////////////////////////////////
// function Map#Glue<U, V>([U] bool, [U]V, Ty): Map U V;
// axiom (forall<U, V> a: [U] bool, b:[U]V, t:Ty ::
// { Map#Domain(Map#Glue(a, b, t)) }
// Map#Domain(Map#Glue(a, b, t)) == a);
// axiom (forall<U, V> a: [U] bool, b:[U]V, t:Ty ::
// { Map#Elements(Map#Glue(a, b, t)) }
// Map#Elements(Map#Glue(a, b, t)) == b);
// axiom (forall<U, V> a: [U] bool, b:[U]V, t:Ty ::
// { $Is(Map#Glue(a, b, t), t) }
// $Is(Map#Glue(a, b, t), t));
/////////////////////////////////////////////////////
/////////////////////////////////////////////////////
/////////////////////////////////////////////////////

// function Map#Domain<U,V>(Map U V) : Set U;
  function vctmap_keys(m:VCTMap[K,V]): Set[K]

// function Map#Card<U,V>(Map U V) : int;
  function vctmap_card(m:VCTMap[K,V]): Int

// function Map#Values<U,V>(Map U V) : Set V;
  function vctmap_values(m: VCTMap[K,V]): Set[V]

// function Map#Elements<U,V>(Map U V) : [U]V;
  function vctmap_get(m:VCTMap[K,V], k: K): V

// function Map#Items<U,V>(Map U V) : Set Box;
  function vctmap_items(m: VCTMap[K,V]): Set[VCTTuple[K,V]]

// function Map#Empty<U, V>(): Map U V;
  function vctmap_empty(): VCTMap[K,V]

// function Map#Build<U, V>(Map U V, U, V): Map U V;
  function vctmap_build(m: VCTMap[K,V], k: K, v: V): VCTMap[K,V]

// function Map#Equal<U, V>(Map U V, Map U V): bool;
  function vctmap_equals(m1: VCTMap[K,V], m2: VCTMap[K,V]): Bool

// function Map#Disjoint<U, V>(Map U V, Map U V): bool;
  function vctmap_disjoint(m1: VCTMap[K,V], m2: VCTMap[K,V]): Bool


///////////////// Trigger generated by Viper //////////////////
// {vctmap_card(m1)}
/////////////////////////// DAFNY ///////////////////////////
// axiom (forall<U,V> m: Map U V :: { Map#Card(m) } 0 <= Map#Card(m));
  axiom vctMapCardAx1 {
    forall m1: VCTMap[K,V] :: {vctmap_card(m1)} vctmap_card(m1) >= 0
  }

```

```
///////////////// Trigger generated by Viper /////////////////
// {vctmap_card(m1)}
// {|vctmap_keys(m1)|}
/////////////////////////// DAFNY ///////////////////////////
// axiom (forall<U,V> m: Map U V :: { Set#Card(Map#Domain(m)) }
// Set#Card(Map#Domain(m)) == Map#Card(m));
  axiom vctMapCardAx2 {
    forall m1: VCTMap[K,V] :: {vctmap_card(m1)} {|vctmap_keys(m1)|}
    vctmap_card(m1) == |vctmap_keys(m1)|
  }

///////////////// Trigger generated by Viper /////////////////
// {v1 in vctmap_values(m1)}
/////////////////////////// DAFNY ///////////////////////////
// axiom (forall<U,V> m: Map U V, v: V :: { Map#Values(m)[v] }
// Map#Values(m)[v] ==
//  (exists u: U :: { Map#Domain(m)[u] } { Map#Elements(m)[u] }
//  Map#Domain(m)[u] &&
// v == Map#Elements(m)[u]));
  axiom vctMapValuesAx {
    forall v1: V, m1: VCTMap[K,V] :: {(v1 in vctmap_values(m1))}
    v1 in vctmap_values(m1) == (exists k1: K :: k1 in vctmap_keys(m1) && vctmap_get(m1, k1) == v1)
  }

///////////////// Trigger generated by Viper /////////////////
// {|vctmap_items(m1)|}
// {vctmap_card(m1)}
/////////////////////////// DAFNY ///////////////////////////
// axiom (forall<U,V> m: Map U V :: { Set#Card(Map#Items(m)) }
// Set#Card(Map#Items(m)) == Map#Card(m));
  axiom vctMapItemsSizeAx {
    forall m1: VCTMap[K,V] :: {vctmap_card(m1)} {|vctmap_items(m1)|}
      |vctmap_items(m1)| == vctmap_card(m1)
  }

///////////////// Trigger generated by Viper /////////////////
// {t1 in vctmap_items(m1)}
/////////////////////////// DAFNY ///////////////////////////
// axiom (forall m: Map Box Box, item: Box :: { Map#Items(m)[item] }
// Map#Items(m)[item] <==>
// Map#Domain(m)[_System.Tuple2._0($Unbox(item))] &&
// Map#Elements(m)[_System.Tuple2._0($Unbox(item))] == _System.Tuple2._1($Unbox(item)));
  axiom vctMapItemsKeysAx {
    forall t1: VCTTuple[K,V], m1: VCTMap[K,V] ::
      {vctmap_get(m1, vcttuple_fst(t1))} {(t1 in vctmap_items(m1))}
      (t1 in vctmap_items(m1)) <==>
      (vcttuple_fst(t1) in vctmap_keys(m1) && vctmap_get(m1, vcttuple_fst(t1)) == vcttuple_snd(t1))
  }

///////////////// Trigger generated by Viper /////////////////
// {k1 in vctmap_keys(vctmap_empty())}
/////////////////////////// DAFNY ///////////////////////////
// axiom (forall<U, V> u: U ::
// { Map#Domain(Map#Empty(): Map U V)[u] }
// !Map#Domain(Map#Empty(): Map U V)[u]);
  axiom vctMapEmptyKeyAx{
    forall k1: K :: {(k1 in vctmap_keys(vctmap_empty()))}
      !(k1 in vctmap_keys(vctmap_empty())) && |vctmap_keys(vctmap_empty())| == 0
  }

///////////////// Trigger generated by Viper /////////////////
// {v1 in vctmap_values(vctmap_empty())}
```

```
/////////////////////////// DAFNY ///////////////////////////
  axiom vctMapEmptyValueAx{
    forall v1: V :: {(v1 in vctmap_values(vctmap_empty()))}
      !(v1 in vctmap_values(vctmap_empty())) && |vctmap_values(vctmap_empty())| == 0
  }

//////////////// Trigger generated by Viper /////////////////
// {vctmap_card(m1)}
/////////////////////////// DAFNY ///////////////////////////
// Chosen trigger: vctmap_card(m1)
  axiom vctMapEmptyCardAx1 {
    forall m1: VCTMap[K,V] :: {vctmap_card(m1)} vctmap_card(m1) == 0 <==> m1 == vctmap_empty()
  }

//////////////// Trigger generated by Viper /////////////////
// {vctmap_card(m1)}
// {vctmap_keys(m1)}
/////////////////////////// DAFNY ///////////////////////////
// axiom (forall<U, V> m: Map U V :: { Map#Card(m) }
// (Map#Card(m) == 0 <==> m == Map#Empty()) &&
// (Map#Card(m) != 0 ==> (exists x: U :: Map#Domain(m)[x])));
  axiom vctMapEmptyCardAx2 {
    forall m1: VCTMap[K,V] :: {vctmap_card(m1)}
      vctmap_card(m1) != 0 ==> (exists k1: K :: k1 in vctmap_keys(m1))
  }

//////////////// Trigger generated by Viper /////////////////
// {k1 in vctmap_keys(vctmap_build(m1,k1,v1))}
// {vctmap_get(vctmap_build(m1,k1,v1),k1)}
/////////////////////////// DAFNY ///////////////////////////
// axiom (forall<U, V> m: Map U V, u: U, v: V ::
// { Map#Domain(Map#Build(m, u, v))[u] } { Map#Elements(Map#Build(m, u, v))[u] }
// Map#Domain(Map#Build(m, u, v))[u] && Map#Elements(Map#Build(m, u, v))[u] == v);
  axiom vctMapBuildAx0 {
    forall k1: K, v1: V, m1: VCTMap[K,V] :: {vctmap_build(m1, k1, v1)}
      k1 in vctmap_keys(vctmap_build(m1, k1, v1)) && vctmap_get(vctmap_build(m1, k1, v1), k1) == v1
  }

//////////////// Trigger generated by Viper /////////////////
// {vctmap_keys(vctmap_build(m1,k1,v1)), vctmap_get(vctmap_build(m1,k1,v1), k2)}
// {k2 in vctmap_keys(vctmap_build(m1,k1,v1))}
// {vctmap_get(vctmap_build(m1,k1,v1),k2)}
/////////////////////////// DAFNY ///////////////////////////
// axiom (forall<U, V> m: Map U V, u: U, u': U, v: V ::
// { Map#Domain(Map#Build(m, u, v))[u'] } { Map#Elements(Map#Build(m, u, v))[u'] }
// (u' == u ==> Map#Domain(Map#Build(m, u, v))[u'] &&
// Map#Elements(Map#Build(m, u, v))[u'] == v) &&
// (u' != u ==> Map#Domain(Map#Build(m, u, v))[u'] == Map#Domain(m)[u'] &&
// Map#Elements(Map#Build(m, u, v))[u'] == Map#Elements(m)[u']));
  axiom vctMapBuildAx1 {
    forall k1: K, k2: K, v1: V, m1: VCTMap[K,V] ::
    {k2 in vctmap_keys(vctmap_build(m1, k1, v1))}
    {vctmap_get(vctmap_build(m1, k1, v1), k2), vctmap_get(m1, k2)}
    (k1 == k2 ==>
      (
        k2 in vctmap_keys(vctmap_build(m1, k1, v1))
       &&

        vctmap_get(vctmap_build(m1, k1, v1), k2) == v1
      )
    )
  }
```

```
///////////////// Trigger generated by Viper //////////////////
// {vctmap_build(m1,k1,v1), k2 in vctmap_keys(m1)}
// {vctmap_build(m1,k1,v1), vctmap_get(m1,k2)}
// {vctmap_keys(vctmap_build(m1,k1,v1)), k2 in vctmap_keys(m1)}
// {vctmap_keys(vctmap_build(m1,k1,v1)), vctmap_get(vctmap_build(m1,k1,v1),k2)}
// {vctmap_keys(vctmap_build(m1,k1,v1)), vctmap_get(m1,k2)}
// {k2 in vctmap_keys(vctmap_build(m1,k1,v1))}
// {vctmap_keys(m1), vctmap_build(m1,k1,v1), vctmap_get(m1,k2)}
// {vctmap_keys(m1), vctmap_get(vctmap_build(m1,k1,v1),k2)}
// {k2 in vctmap_keys(m1), vctmap_build(m1,k1,v1)}
// {k2 in vctmap_keys(m1), vctmap_get(vctmap_build(m1,k1,v1),k2)}
// {vctmap_get(vctmap_build(m1,k1,v1),k2)}
//////////////////////////// DAFNY ////////////////////////////
  axiom vctMapBuild1Dot5Ax {
  forall k1: K, k2: K, v1: V, m1: VCTMap[K,V] ::
  {k2 in vctmap_keys(vctmap_build(m1, k1, v1))}
  {vctmap_get(vctmap_build(m1, k1, v1), k1), vctmap_get(m1, k2)}
  (k1 != k2 ==>
      (
        (k2 in vctmap_keys(vctmap_build(m1, k1, v1))) == (k2 in vctmap_keys(m1))
        &&
        vctmap_get(vctmap_build(m1, k1, v1), k2) == vctmap_get(m1, k2)
      )
    )
  }

///////////////// Trigger generated by Viper //////////////////
// {vctmap_keys(m1), vctmap_card(vctmap_build(m1,k1,v1))}
// {k1 in vctmap_keys(m1), vctmap_build(m1,k1,v1)}
// {k1 in vctmap_keys(m1), vctmap_card(vctmap_build(m1,k1,v1))}
// {vctmap_card(vctmap_build(m1,k1,v1))}
//////////////////////////// DAFNY ////////////////////////////
// axiom (forall<U, V> m: Map U V, u: U, v: V :: { Map#Card(Map#Build(m, u, v)) }
// Map#Domain(m)[u] ==> Map#Card(Map#Build(m, u, v)) == Map#Card(m));
  axiom vctMapBuildAx2 {
    forall k1: K, v1: V, m1: VCTMap[K,V] :: {vctmap_card(vctmap_build(m1, k1, v1))}
    (k1 in vctmap_keys(m1)) ==> (vctmap_card(vctmap_build(m1, k1, v1)) == vctmap_card(m1))
  }

///////////////// Trigger generated by Viper //////////////////
// {vctmap_keys(m1), vctmap_card(vctmap_build(m1,k1,v1))}
// {k1 in vctmap_keys(m1), vctmap_build(m1,k1,v1)}
// {k1 in vctmap_keys(m1), vctmap_card(vctmap_build(m1,k1,v1))}
// {vctmap_card(vctmap_build(m1,k1,v1))}
//////////////////////////// DAFNY ////////////////////////////
// axiom (forall<U, V> m: Map U V, u: U, v: V :: { Map#Card(Map#Build(m, u, v)) }
// !Map#Domain(m)[u] ==> Map#Card(Map#Build(m, u, v)) == Map#Card(m) + 1);
  axiom vctMapBuildAx3 {
    forall k1: K, v1: V, m1: VCTMap[K,V] :: {vctmap_card(vctmap_build(m1, k1, v1))}
      !(k1 in vctmap_keys(m1)) ==> (vctmap_card(vctmap_build(m1, k1, v1)) == vctmap_card(m1)+1)
  }

///////////////// Trigger generated by Viper //////////////////
// {vctmap_equals(m1,m2)}
// {vctmap_keys(m1), vctmap_keys(m2)}
// {vctmap_keys(m2), vctmap_keys(m1)}
//////////////////////////// DAFNY ////////////////////////////
// axiom (forall<U, V> m: Map U V, m': Map U V::
// { Map#Equal(m, m') }
// Map#Equal(m, m') <==> (forall u : U :: Map#Domain(m)[u] == Map#Domain(m')[u]) &&
// (forall u : U :: Map#Domain(m)[u] ==> Map#Elements(m)[u] == Map#Elements(m')[u]));
  axiom vctMapEqualsAx1 {
    forall m1: VCTMap[K,V], m2: VCTMap[K,V] :: {vctmap_equals(m1, m2)}
    vctmap_equals(m1, m2) <==>
    (
      (
        vctmap_keys(m1) == vctmap_keys(m2)
      ) &&
      (
        forall k: K :: k in vctmap_keys(m1) ==> vctmap_get(m1, k) == vctmap_get(m2, k)
      )

    )
  }

//////////////// Trigger generated by Viper /////////////////
// {vctmap_equals(m1,m2)}
/////////////////////////// DAFNY ///////////////////////////
// // extensionality
// axiom (forall<U, V> m: Map U V, m': Map U V::
// { Map#Equal(m, m') }
// Map#Equal(m, m') ==> m == m');
  axiom vctMapEqualsAx2 {
    forall m1: VCTMap[K,V], m2: VCTMap[K,V] :: {vctmap_equals(m1, m2)}
      vctmap_equals(m1, m2) <==> (m1 == m2)
  }

//////////////// Trigger generated by Viper /////////////////
// {vctmap_disjoint(m1,m2)}
// {vctmap_keys(m1), vctmap_keys(m2)}
// {vctmap_keys(m2), vctmap_keys(m1)}
/////////////////////////// DAFNY ///////////////////////////
// axiom (forall<U, V> m: Map U V, m': Map U V ::
// { Map#Disjoint(m, m') }
// Map#Disjoint(m, m') <==> (forall o: U :: {Map#Domain(m)[o]} {Map#Domain(m')[o]} !Map#Domain(m)[o] ||
     ↪ !Map#Domain(m')[o]));
  axiom vctMapDisjointAx1 {
    forall m1: VCTMap[K,V], m2: VCTMap[K,V] :: {vctmap_disjoint(m1, m2)}
      vctmap_disjoint(m1, m2) <==>
      (forall k: K :: {k in vctmap_keys(m1)} {k in vctmap_keys(m2)} !(k in vctmap_keys(m1)) || !(k in
          ↪ vctmap_keys(m2)))
  }


// OWN FUNCTION AND AXIOMS
  function vctmap_remove(m: VCTMap[K,V], k: K): VCTMap[K,V]

//////////////// Trigger generated by Viper /////////////////
// {k in vctmap_keys(vctmap_remove(m,k))}
  axiom vctMapRemoveAx1 {
    forall m: VCTMap[K,V], k: K :: {vctmap_remove(m, k)} !(k in vctmap_keys(vctmap_remove(m, k)))
  }

//////////////// Trigger generated by Viper /////////////////
// {vctmap_keys(m), vctmap_remove(m,k)}
// {vctmap_keys(m), vctmap_get(vctmap_remove(m,k), k)}
// {vctmap_keys(m), vctmap_get(m,k)}
// {k in vctmap_keys(m)}
// {vctmap_get(vctmap_remove(m,k), k)}
// {vctmap_get(m,k)}
  axiom vctMapRemoveAx2 {
    forall m: VCTMap[K, V], k: K :: {vctmap_remove(m, k)}
      (k in vctmap_keys(m)) ==> vctmap_get(vctmap_remove(m, k), k) != vctmap_get(m, k)
  }
```

```

///////////////// Trigger generated by Viper /////////////////
// {vctmap_remove(m,k1), k2 in vctmap_keys(m)}
// {vctmap_remove(m,k1), vctmap_get(m,k2)}
// {vctmap_keys(vctmap_remove(m,k1)), k2 in vctmap_keys(m)}
// {vctmap_keys(vctmap_remove(m,k1)), vctmap_get(vctmap_remove(m,k1),k2)}
// {vctmap_keys(vctmap_remove(m,k1)), vctmap_get(m,k2)}
// {k2 in vctmap_keys(vctmap_remove(m,k1))}
// {vctmap_keys(m), vctmap_remove(m,k1), vctmap_get(m,k2)}
// {vctmap_keys(m), vctmap_get(vctmap_remove(m,k1),k2)}
// {k2 in vctmap_keys(m), vctmap_remove(m,k1)}
// {k2 in vctmap_keys(m), vctmap_get(vctmap_remove(m,k1),k2)}
// {vctmap_get(vctmap_remove(m,k1),k2)}
  axiom vctMapRemoveAx3 {
    (forall m: VCTMap[K, V], k1: K, k2: K ::
    {(k2 in vctmap_keys(vctmap_remove(m, k1)))}
    {vctmap_get(vctmap_remove(m, k1), k2)}
      k1 != k2 ==>
        (k2 in vctmap_keys(vctmap_remove(m, k1)) == k2 in vctmap_keys(m) &&
        vctmap_get(vctmap_remove(m, k1), k2) == vctmap_get(m, k2)))
  }

///////////////// Trigger generated by Viper /////////////////
// {vctmap_keys(m), vctmap_remove(m,k)}
// {vctmap_keys(m), vctmap_card(vctmap_remove(m,k))}
// {k in vctmap_keys(m)}
// {vctmap_card(m), vctmap_remove(m,k)}
// {vctmap_card(m), vctmap_card(vctmap_remove(m,k))}
// {vctmap_card(vctmap_remove(m,k))}
  axiom vctMapRemoveAx4 {
    forall m: VCTMap[K,V], k: K :: {vctmap_remove(m, k)}
      (k in vctmap_keys(m)) ==> vctmap_card(m)-1 == vctmap_card(vctmap_remove(m, k))
  }

///////////////// Trigger generated by Viper /////////////////
// {vctmap_keys(m), vctmap_remove(m,k)}
// {vctmap_keys(m), vctmap_card(vctmap_remove(m,k))}
// {k in vctmap_keys(m)}
// {vctmap_card(m), vctmap_remove(m,k)}
// {vctmap_card(m), vctmap_card(vctmap_remove(m,k))}
// {vctmap_card(vctmap_remove(m,k))}
  axiom vctMapRemoveAx5 {
    forall m: VCTMap[K,V], k: K :: {vctmap_remove(m, k)}
      !(k in vctmap_keys(m)) ==> vctmap_card(m) == vctmap_card(vctmap_remove(m, k))
  }

///////////////// Trigger generated by Viper /////////////////
// {vctmap_keys(m), vctmap_remove(m,k)}
// {vctmap_keys(m), vctmap_card(vctmap_remove(m,k))}
// {k in vctmap_keys(m)}
// {vctmap_card(m), vctmap_remove(m,k)}
// {vctmap_card(m), vctmap_card(vctmap_remove(m,k))}
// {vctmap_card(vctmap_remove(m,k))}
  axiom vctMapRemoveAx6 {
      (forall m: VCTMap[K, V], k: K :: {vctmap_remove(m, k)}
        !((k in vctmap_keys(m))) ==> vctmap_equals(m, vctmap_remove(m, k)))
  }
}
```

Appendix D

A Large Example using Maps

Listing D.1 shows an example using maps. This example is based on a test in the Dafny repository and is translated to PVL. The file can be found at `https://github.com/dafny-lang/dafny/blob/213ed90c75bee4f30638c60060dd7ac5971b82a0/Test/dafny0/Maps.dfy`.

Listing D.1: An example based on a test in the Dafny repository

```
class Maps {

    void main() {
        map<int,int> m = map<int,int> {2 -> 3};
        assert 2 in keysMap(m);
        assert !(3 in keysMap(m));
        assert m[2] == 3;
        assert disjointMap(m, map<int, int> {3 -> 3});
        assert map<int, int> {2 -> 4} == map<int, int> {2 -> 4};
        assert equalsMap(m ++ (7, 1), m ++ (2, 3) ++ (7, 1));
        assert equalsMap(m, m ++ (2, 3));
    }

    void m() {
        map<int, int> a = map<int, int> {2 -> 3};
        map<int, int> b = map<int, int> {3 -> 2};
        assert a[b[3]] == 3;
    }

    requires (\forall int i; 0 <= i && i < 100; i in keysMap(a) && i in keysMap(b) && a[i] != b[i]);
    void m2(map<int, boolean> a, map<int, boolean> b) {
        assert (\forall int i; 0 <= i  && i < 100; a[i] || b[i]);
    }

    requires (\forall int i; 0 <= i && i < 100; i in keysMap(a) && a[i] == i*i);
    void m3(map<int, int> a) {
        assert a[20] == 400;
    }

    void m4() {
        map<int, int> a = map<int, int> {3 -> 9};
        if (a[4] == 4) {
            m();
        }
    }

    requires 20 in keysMap(a);
    void m5(map<int, int> a) {
        assert a[20] <= 0 || 0 < a[20];
    }

    void m6() {
        map<int,int> a = map<int,int> {3 -> 9};
        assert map<int, int> {2 -> 4} == map<int, int> {2 -> 4};
        assert a ++ (3, 5) == map<int, int> {3 -> 5};
        assert a ++ (2, 5) == map<int, int> {2 -> 5, 3 -> 9};
        assert a ++ (2, 5) == map<int, int> {2 -> 6, 3 -> 9, 2 -> 5};
    }

    void m7() {
        map<int,int> a = map<int,int> {1 -> 1, 2 -> 4, 3 -> 9};
        assert (\forall int i; i in keysMap(a); a[i] == i*i);
        assert !(0 in keysMap(a));
        assert 1 in keysMap(a);
        assert 2 in keysMap(a);
        assert 3 in keysMap(a);
        assert (\forall int i; i < 1 || i > 3; !(i in keysMap(a)));
    }

    void m8() {
        map<int,int> a = map<int,int> {};
        assert (\forall int i; true; !(i in keysMap(a)));
        int i = 0;
        int n = 100;


        loop_invariant 0 <= i && i <= n;
        loop_invariant (\forall int j; j in keysMap(a); a[j] == j*j);
        loop_invariant (\forall int k; true; (0 <= k && k < i) == k in keysMap(a));
        while (i < n) {
          a = a ++ (i, (i*i));
          i = i + 1;
        }

        assert disjointMap(a, map<int, int>{-1 -> 2});
        m3(a);
    }

    void m9() {
        map<int,int> a = map<int,int> {};
        map<int,int> b = map<int,int> {};
        assert disjointMap(a, b);

        b = map<int,int> {2 -> 3, 4 -> 2, 5 -> -6, 6 -> 7};
        assert disjointMap(a, b);
        assert !disjointMap(b, map<int,int> {6 -> 3});
    }

    void m10()
    {
        map<int,int> a = map<int,int> {};
        map<int,int> b = map<int,int> {};
        assert disjointMap(a, b);

        b = map<int,int> {2 -> 3, 4 -> 2, 5 -> -6, 6 -> 7};
        assert disjointMap(a, b);

        a = map<int,int> {3 -> 3, 1 -> 2, 9 -> -6, 8 -> 7};
        assert disjointMap(a, b);
    }
}
```

Appendix E

Instantiation Graphs for the Methods in Listing D.1

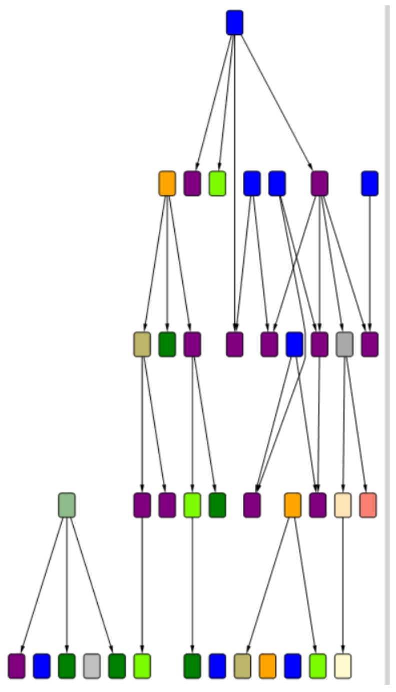


Figure E.1: The instantiation graph for the `main` method

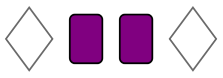


Figure E.2: The instantiation graph for the `m` method

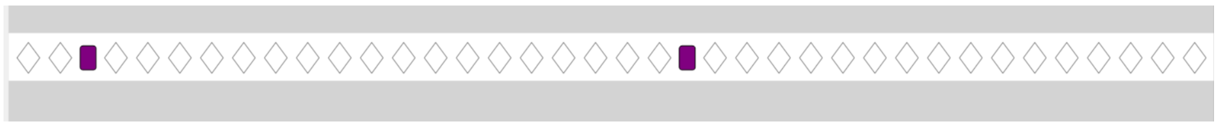


Figure E.3: The instantiation graph for the `m2` method

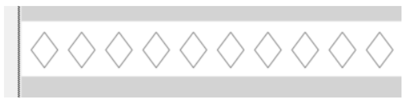


Figure E.4: The instantiation graph for the `m3` method

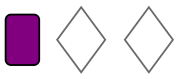


Figure E.5: The instantiation graph for the `m4` method

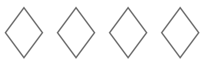


Figure E.6: The instantiation graph for the `m5` method

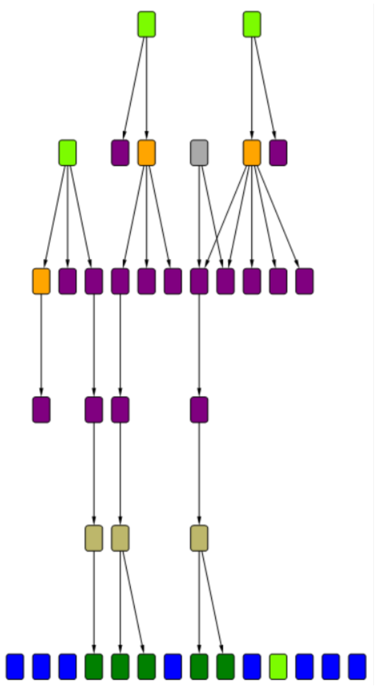


Figure E.7: The instantiation graph for the `m6` method

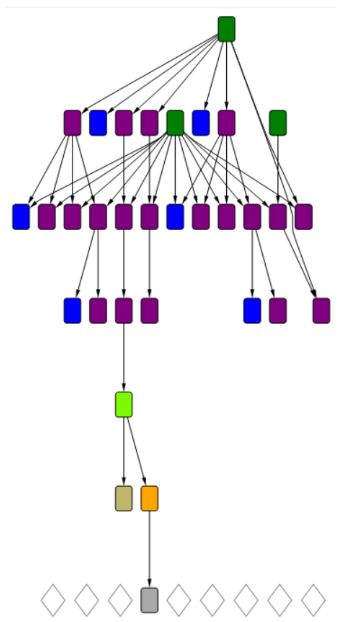


Figure E.8: The instantiation graph for the `m7` method

Figure E.9: The instantiation graph for the `m8` method

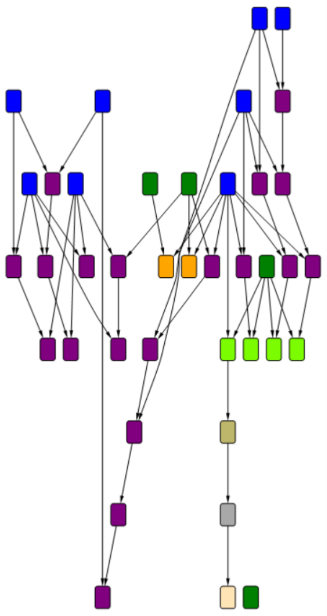


Figure E.10: The instantiation graph for the `m9` method

Bibliography

[1] The webpage of Dafny. https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/. Accessed: 16-04-2020.

[2] The webpage of GPUVerify. http://multicore.doc.ic.ac.uk/tools/GPUVerify/. Accessed: 16-04-2020.

[3] The webpage of Frama-C. https://frama-c.com/. Accessed: 16-04-2020.

[4] The webpage of VerCors. https://vercors.ewi.utwente.nl/. Accessed: 16-04-2020.

[5] The webpage of VeriFast. https://people.cs.kuleuven.be/~bart.jacobs/verifast/. Accessed: 17-04-2020.

[6] The webpage of Boogie. https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/. Accessed: 17-04-2020.

[7] Stefan Blom and Marieke Huisman. The VerCors tool for verification of concurrent programs. In *International Symposium on Formal Methods*, pages 127–131. Springer, 2014.

[8] The main page on Viper. https://www.pm.inf.ethz.ch/research/viper.html. Accessed: 18-09-2019.

[9] The VerCors Verifier. https://utwente-fmt.github.io/vercors/. Accessed: 18-09-2019.

[10] Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. The VerCors project: Setting up basecamp. In *Proceedings of the sixth workshop on Programming languages meets program verification*, pages 71–82. ACM, 2012.

[11] Wytse Hendrikus Marinus Oortwijn. *Deductive techniques for model-based concurrency verification*. PhD thesis, University of Twente, Netherlands, 12 2019.

[12] Sebastiaan JC Joosten, Wytse Oortwijn, Mohsen Safari, and Marieke Huisman. An exercise in verifying sequential programs with VerCors. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 40–45. ACM, 2018.

[13] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, Lecture Notes in Computer Science, pages 102–110. Springer, 2017.

[14] Stefan Blom and Marieke Huisman. *Witnessing the elimination of magic wands*. Number TR-CTIT-13-22 in CTIT Technical Report Series. Centre for Telematics and Information Technology (CTIT), Netherlands, 11 2013.

[15] The Github page of ANTLR. https://github.com/antlr/antlr4#antlr-v4. Accessed: 18-09-2019.

[16] The GitHub page of VerCors. https://github.com/utwente-fmt/vercors#vercors-verification-toolset. Accessed: 19-09-2019.

[17] The PVL Syntax page. https://github.com/utwente-fmt/vercors/wiki/PVL-Syntax. Accessed: 15-04-2020.

[18] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.

[19] The Silicon BitBucket page. https://bitbucket.org/viperproject/silicon/. Accessed: 29-12-2019.

[20] The Carbon BitBucket page. https://bitbucket.org/viperproject/carbon/. Accessed: 29-12-2019.

[21] The Silicon ADT axiomatizations. https://bitbucket.org/viperproject/silicon/src/e96d2ce8fdf5be1bddb47aae0d114c2f9d0d7d8f/src/main/resources/dafny_axioms/?at=default. Accessed: 31-12-2019.

[22] The Carbon ADT axiomatizations. https://bitbucket.org/viperproject/carbon/src/1221bd1decd5bf4f23ed5c0c6a9b8d48eb2245b1/src/main/scala/viper/carbon/modules/impls/sequence_axioms/?at=default. Accessed: 31-12-2019.

[23] K Rustan M Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *International Conference on Computer Aided Verification*, pages 361–381. Springer, 2016.

[24] David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[25] Michał Moskal. Programming with triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 20–29, 2009.

[26] The Viper tutorial. http://viper.ethz.ch/tutorial/. Accessed: 18-09-2019.

[27] Nils Becker, Peter Müller, and Alexander J Summers. The axiom profiler: understanding and debugging smt quantifier instantiations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–116. Springer, 2019.

[28] The VerCors ADT documentation. https://github.com/utwente-fmt/vercors/wiki/Axiomatic-Data-Types#future-enhancements-1. Accessed: 18-09-2019.

[29] The VerCors Examples directory. https://github.com/utwente-fmt/vercors/tree/master/examples. Accessed: 23-10-2019.

[30] Justin Pombrio, Shriram Krishnamurthi, and Mitchell Wand. Inferring scope through syntactic sugar. *Proceedings of the ACM on Programming Languages*, 1(ICFP):44, 2017.

[31] Dafny Axiomatizations for different collections. https://github.com/dafny-lang/dafny/blob/master/Binaries/DafnyPrelude.bpl. Accessed: 26-02-2020.

[32] Matthias Erdin, Vytautas Astrauskas, and Federico Poli. Verification of rust generics, typestates, and traits. 2018.

[33] The formal verifier Prusti. https://www.pm.inf.ethz.ch/research/prusti.html. Accessed: 30-03-2019.

[34] Benjamin WEBER, Peter MÜLLER, Arshavir TER-GABRIELYAN, and Marco EILERS. Automating modular reasoning about higher-order functions. 2017.

[35] The webpage of Nagini. https://www.pm.inf.ethz.ch/research/nagini.html. Accessed: 20-04-2020.